Set-Constrained Delivery Broadcast: Definition, Abstraction Power, and Computability Limits

Damien Imbs, Achour Mostefaoui, Matthieu Perrin, Michel Raynal

Set-Constrained Delivery Broadcast: Definition, Abstraction Power, and Computability Limits

Damien Imbs°, Achour Mostéfaoui†, Matthieu Perrin◇, Michel Raynal⋆,‡

°LIF, Université Aix-Marseille, 13288 Marseille, France
†LINA, Université de Nantes, 44322 Nantes, France
◇IMDEA Software Institute, 28223 Pozuelo de Alarcón, Madrid, Spain
⋆Institut Universitaire de France
‡IRISA, Université de Rennes, 35042 Rennes, France

June 15, 2017

Abstract

This paper introduces a new communication abstraction, called *Set-Constrained Delivery Broadcast* (SCD-broadcast), whose aim is to provide its users with an appropriate abstraction level when they have to implement *objects* or *distributed tasks* in an asynchronous message-passing system prone to process crash failures. This abstraction allows each process to broadcast messages and deliver a sequence of sets of messages in such a way that, if a process delivers a set of messages including a message m and later delivers a set of messages including a message m', no process delivers first a set of messages including m' and later a set of message including m.

After having presented an algorithm implementing SCD-broadcast, the paper investigates its programming power and its computability limits. On the "power" side it presents SCD-broadcast-based algorithms, which are both simple and efficient, building objects (such as snapshot and conflict-free replicated data), and distributed tasks. On the "computability limits" side it shows that SCD-broadcast and read/write registers are computationally equivalent.

Keywords: Abstraction, Asynchronous system, Communication abstraction, Communication pattern, Conflict-free replicated data type, Design simplicity, Distributed task, Lattice agreement, Linearizability, Message-passing system, Process crash, Read/write atomic register, Snapshot object.

1 Introduction

Programming abstractions Informatics is a science of abstractions, and a main difficulty consists in providing users with a "desired level of abstraction and generality – one that is broad enough to encompass interesting new situations, yet specific enough to address the crucial issues" as expressed in [18]. When considering sequential computing, functional programming and object-oriented programming are well-know examples of what means "desired level of abstraction and generality".

In the context of asynchronous distributed systems where the computing entities (processes) communicate –at the basic level– by sending and receiving messages through an underlying communication network, and where some of them can experience failures, a main issue consists in finding appropriate communication-oriented abstractions, where the meaning of the term "appropriate" is related to the problems we intend to solve. Solving a problem at the send/receive abstraction level is similar to the writing of a program in a low-level programming language. Programmers must be provided with abstractions that allow them to concentrate on the problem they solve and not on the specific features of the underlying system. This is not new. Since a long time, high level programming languages have proved the benefit of this approach. From a synchronization point of view, this approach is the one promoted in *software transactional memory* [33], whose aims is to allow programmers to focus on the synchronization needed to solve their problems and not on the way this synchronization must be implemented (see the textbooks [19, 29]).

If we consider specific coordination/cooperation problems, "matchings" between problems and specific communication abstractions are known. One of the most famous examples concerns the consensus problem whose solution rests on the *total order broadcast* abstraction[1]. Another "matching" example is the *causal message delivery* broadcast abstraction [11, 31], which allows for a very simple implementation of a causal read/write memory [2].

Aim of the paper The aim of this paper is to introduce and investigate a high level communication abstraction which allows for simple and efficient implementations of concurrent objects and distributed tasks, in the context of asynchronous message-passing systems prone to process crash failures. The concurrent objects in which we are interested are defined by a sequential specification [20] (e.g., a queue). Differently, a task extends to the distributed context the notion of a function [10, 27]. It is defined by a mapping from a set of input vectors to a set of output vectors, whose sizes are the number of processes. An input vector I defines the input value $I[i]$ of each process p_i, and, similarly, an output vector O defines the output $O[j]$ of each process p_j. Agreement problems such as consensus and k-set agreement are distributed tasks. What makes difficult the implementation of a task is the fact that each process knows only its input, and, due to net effect of asynchrony and process failures, no process can distinguish if another process is very slow or crashed. The difficulty is actually an impossibility for consensus [17], even in a system in which at most one process may crash.

Content of the paper: a broadcast abstraction The SCD-broadcast communication abstraction proposed in the paper allows a process to broadcast messages, and to deliver sets of messages (instead of a single message) in such a way that, if a process p_i delivers a message set ms containing a message m, and later delivers a message set ms' containing a message m', then no process p_j can deliver first a set containing m' and later another set containing m. Let us notice that p_j is not prevented from delivering m and m' in the same set. Moreover, SCD-broadcast imposes no constraint on the order in which a process must process the messages it receives in a given message set.

[1]Total order broadcast is also called *atomic broadcast*. Actually, total order broadcast and consensus have been shown to be computationally equivalent [12]. A more general result is presented in [21], where is introduced a communication abstraction which "captures" the k-set agreement problem [13, 30] (consensus is 1-set agreement).

After having introduced SCD-broadcast, the paper presents an implementation of it in asynchronous systems where a minority of processes may crash. This assumption is actually a necessary and sufficient condition to cope with the net effect of asynchrony and process failures (see below). Assuming an upper bound Δ on message transfer delays, and zero processing time, an invocation of SCD-broadcast is upper bounded by 2Δ time units, and $O(n^2)$ protocol messages (messages generated by the implementation algorithm).

Content of the paper: implementing objects and tasks Then, the paper addresses two fundamental issues of SCD-broadcast: its abstraction power and its computability limits. As far as its abstraction power is concerned, i.e., its ability and easiness to implement atomic (linearizable) or sequentially consistent concurrent objects [20, 26] and read/write solvable distributed tasks, the paper presents, on the one side, two algorithms implementing atomic objects (namely a snapshot object [1, 3], and a distributed increasing/decreasing counter), and, on the other side, an algorithm solving the lattice agreement task [6, 16].

The two concurrent objects (snapshot and counter) have been chosen because they are encountered in many applications, and are also good representative of the class of objects identified in [4]. The objects of this class are characterized by the fact that each pair op1 and op2 of their operations either commute (i.e., in any state, executing op1 before op2 is the same as executing op2 before op1, as it is the case for a counter), or any of op1 and op2 can overwrite the other one (e.g., executing op1 before op2 is the same as executing op2 alone). Our implementation of a counter can be adapted for all objects with commutative operations, and our implementation of the snapshot object illustrates how overwriting operations can be obtained directly from the SCD-broadcast abstraction. Concerning these objects, it is also shown that a slight change in the algorithms allows us to obtain implementations (with a smaller cost) in which the consistency condition is weakened from linearizability to sequential consistency [25].

In the case of read/write solvable tasks, SCD-broadcast shows how the concurrency inherent (but hidden) in a task definition can be easily mastered and solved.

A distributed software engineering dimension All the algorithms presented in the paper are based on the same communication pattern. As far as objects are concerned, the way this communication pattern is used brings to light two genericity dimensions of the algorithms implementing them. One is on the variety of objects that, despite their individual features (e.g., snapshot vs counter), have very similar SCD-broadcast-based implementations (actually, they all have the same communication pattern-based structure). The other one is on the consistency condition they have to satisfy (linearizability vs sequential consistency).

Content of the paper: the computability limits of SCD-broadcast The paper also investigates the computability power of the SCD-broadcast abstraction, namely it shows that SCD-broadcast and atomic read/write registers (or equivalently snapshot objects) have the same computability power in asynchronous systems prone to process crash failures. Everything that can be implemented with atomic read/write registers can be implemented with SCD-broadcast, and vice versa.

As read/write registers (or snapshot objects) can be implemented in asynchronous message-passing system where only a minority of processes may crash [5], it follows that the proposed algorithm implementing SCD-broadcast is resilience-optimal in these systems. From a theoretical point of view, this means that the consensus number of SCD-broadcast is 1 (the weakest possible).

Roadmap The paper is composed of 9 sections. Section 2 defines the SCD-broadcast abstraction and the associated communication pattern used in all the algorithms presented in the paper. Section 3 presents a resilience-optimal algorithm implementing SCD-broadcast in asynchronous message-passing systems prone to process crash failures, while Section 4 adopts a distributed software engineering point of view and presents a communication pattern associated with SCD-broadcast. Then, Sections 5-7 present SCD-broadcast-based algorithms for concurrent objects and tasks. Section 8 focuses on the computability limits of SCD-broadcast. Finally, Section 9 concludes the paper.

2 The SCD-broadcast Communication Abstraction

Process model The computing model is composed of a set of n asynchronous sequential processes, denoted p_1, ..., p_n. "Asynchronous" means that each process proceeds at its own speed, which can be arbitrary and always remains unknown to the other processes.

A process may halt prematurely (crash failure), but it executes its local algorithm correctly until it crashes (if it ever does). The model parameter t denotes the maximal number of processes that may crash in a run r. A process that crashes in a run is said to be *faulty* in r. Otherwise, it is *non-faulty*.

Definition of SCD-broadcast The set-constrained broadcast abstraction (SCD-broadcast) provides the processes with two operations, denoted $\mathsf{scd\_broadcast}()$ and $\mathsf{scd\_deliver}()$. The first operation takes a message to broadcast as input parameter. The second one returns a non-empty set of messages to the process that invoked it. Using a classical terminology, when a process invokes $\mathsf{scd\_broadcast}(m)$, we say that it "scd-broadcasts a message m". Similarly, when it invokes $\mathsf{scd\_deliver}()$ and obtains a set of messages ms, we say that it "scd-delivers the set of messages ms". By a slight abuse of language, when we are interested in a message m, we say that a process "scd-delivers the message m" when actually it scd-delivers the message set ms containing m.

SCD-broadcast is defined by the following set of properties, where we assume –without loss of generality– that all the messages that are scd-broadcast are different.

- Validity. If a process scd-delivers a set containing a message m, then m was scd-broadcast by some process.
- Integrity. A message is scd-delivered at most once by each process.
- MS-Ordering. Let p_i be a process that scd-delivers first a message set ms_i and later a message set ms'_i. For any pair of messages $m \in ms_i$ and $m' \in ms'_i$, then no process p_j scd-delivers first a message set ms'_j containing m' and later a message set ms_j containing m.
- Termination-1. If a non-faulty process scd-broadcasts a message m, it terminates its scd-broadcast invocation and scd-delivers a message set containing m.
- Termination-2. If a process scd-delivers a message m, every non-faulty process scd-delivers a message set containing m.

Termination-1 and Termination-2 are classical liveness properties (found for example in Uniform Reliable Broadcast [9, 28]). The other ones are safety properties. Validity and Integrity are classical communication-related properties. The first states that there is neither message creation nor message corruption, while the second states that there is no message duplication.

The MS-Ordering property is new, and characterizes SCD-broadcast. It states that the contents of the sets of messages scd-delivered at any two processes are not totally independent: the sequence of sets scd-delivered at a process p_i and the sequence of sets scd-delivered at a process p_j must be mutually consistent in the sense that a process p_i cannot scd-deliver first $m \in ms_i$ and later $m' \in ms'_i \neq ms_i$, while another process p_j scd-delivers first $m' \in ms'_j$ and later $m \in ms_j \neq ms'_j$. Let us nevertheless observe that if p_i scd-delivers first $m \in ms_i$ and later $m' \in ms'_i$, p_j may scd-deliver m and m' in the same set of messages.

Let us remark that, if the MS-Ordering property is suppressed and messages are scd-delivered one at a time, SCD-broadcast boils down to the well-known *Uniform Reliable Broadcast* abstraction [12, 28].

An example Let m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_8, ... be messages that have been scd-broadcast by different processes. The following scd-deliveries of message sets by p_1, p_2 and p_3 respect the definition of SCD-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 scd-deliveries of the sets including m_2 and m_3, the following scd-deliveries by p_1 and p_2 do not satisfy the MS-broadcast property:

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

A containment property Let ms_i^ℓ be the ℓ-the message set scd-delivered by p_i. Hence, at some time, p_i scd-delivered the sequence of message sets $ms_i^1, \ldots, ms_i^x$. Let $MS_i^x = ms_i^1 \cup \ldots \cup ms_i^x$. The following property follows directly from the MS-Ordering and Termination-2 properties:

- Containment. $\forall\, i, j, x, y$: $(MS_i^x \subseteq MS_j^y) \vee (MS_j^y \subseteq MS_i^x)$.

Partial order on messages created by the message sets The MS-Ordering and Integrity properties establish a partial order on the set of all the messages, defined as follows. Let $\mapsto_i$ be the local message delivery order at process p_i defined as follows: $m \mapsto_i m'$ if p_i scd-delivers the message set containing m before the message set containing m'. As no message is scd-delivered twice, it is easy to see that $\mapsto_i$ is a partial order (locally know by p_i). The reader can check that there is a total order (which remains unknown to the processes) on the whole set of messages, that complies with the partial order $\mapsto = \cup_{1 \leq i \leq n} \mapsto_i$. This is where SCD-broadcast can be seen as a weakening of total order broadcast.

3 An Implementation of SCD-broadcast

This section shows that the SCD-broadcast communication abstraction is not an oracle-like object (oracles allow us to extend our understanding of computing, but cannot be implemented). It describes an implementation of SCD-broadcast in an asynchronous send/receive message-passing system in which any minority of processes may crash. This system model is denoted $\mathcal{CAMP}_{n,t}[t < n/2]$ (where $\mathcal{CAMP}_{n,t}$ stands for "Crash Asynchronous Message-Passing" and $t < n/2$ is its restriction on failures). As $t < n/2$ is the weakest assumption on process failures that allows a read/write register to be built on top of an asynchronous message-passing system [5][2], and SCD-broadcast and read/write registers are computationally equivalent (as shown in the paper), the proposed implementation is optimal from a resilience point of view.

3.1 Underlying communication network

Send/receive asynchronous network Each pair of processes communicate by sending and receiving messages through two uni-directional channels, one in each direction. Hence, the communication network is a complete network: any process p_i can directly send a message to any process p_j (including itself). A process p_i invokes the operation "send TYPE(m) to p_j" to send to p_j the message m, whose type is TYPE. The operation "receive TYPE() from p_j" allows p_i to receive from p_j a message whose type is TYPE.

[2]From the point of view of the maximal number of process crashes that can be tolerated, assuming failures are independent.

Each channel is reliable (no loss, corruption, nor creation of messages), not necessarily first-in/first-out, and asynchronous (while the transit time of each message is finite, there is no upper bound on message transit times) Let us notice that, due to process and message asynchrony, no process can know if another process crashed or is only very slow.

Uniform FIFO-broadcast abstraction To simplify the presentation, and without loss of generality, we consider that the system is equipped with a FIFO-broadcast abstraction. Such an abstraction can be built on top of the previous basic system model without enriching it without additional assumptions (see e.g. [28]). It is defined by the operations $\mathsf{fifo\_broadcast}()$ and $\mathsf{fifo\_deliver}()$, which satisfy the properties of Uniform Reliable Broadcast (Validity, Integrity, Termination 1, and Termination 2), plus the following message ordering property.

- FIFO-Order. For any pair of processes $p_i$and p_j, if p_i fifo-delivers first a message m and later a message m', both from p_j, no process fifo-delivers m' before m.

3.2 Algorithm

This section describes Algorithm 1, which implements SCD-broadcast in $\mathcal{CAMP}_{n,t}[t < n/2]$. From a terminology point of view, an *application message* is a message that has been scd-broadcast by a process, while a *protocol message* is an implementation message generated by the algorithm.

Local variables at a process p_i Each process p_i manages the following local variables.

- $buffer_i$: buffer (initialized empty) where are stored quadruplets containing messages that have been fifo-delivered but not yet scd-delivered in a message set.
- $to\_deliver_i$: set of quadruplets containing messages to be scd-delivered.
- sn_i: local logical clock (initialized to 0), which increases by step 1 and measures the local progress of p_i. Each application message scd-broadcast by p_i is identified by a pair $\langle i, sn\rangle$, where sn is the current value of sn_i.
- $clock_i[1..n]$: array of logical dates; $clock_i[j]$ is the greatest date x such that the application message m identified $\langle x, j\rangle$ has been scd-delivered by p_i.

Content of quadruplet The fields of a quadruplet $qdplt = \langle qdplt.msg, qdplt.sd, qdplt.f, qdplt.cl\rangle$ have the following meaning.

- $qdplt.msg$ contains an application message m,
- $qdplt.sd$ contains the id of the sender of this application message,
- $qdplt.sn$ contains the local date (seq. number) associated with m by its sender. Hence, the pair $\langle qdplt.sd, qdplt.sn\rangle$ is the identity of m.
- $qdplt.cl$ is an array of size n, initialized to $[+\infty, \ldots, +\infty]$. Then, $qdplt.cl[x]$ will contain the sequence number associated with m by p_x when it broadcast $\text{FORWARD}(msg.m, -, -, -, -)$. This last field is crucial in the scd-delivery by the process p_i of a message set containing m.

Protocol message The algorithm uses a single type of protocol message denoted FORWARD(). Such a message is made up of five fields: an associated application message m, and two pairs, each made up of a sequence number and a process identity. The first pair $\langle sd, sn\rangle$ is the identity of the application message, while the second pair $\langle f, sn_f\rangle$ is the local progress (as captured by sn_f) of the forwarder process p_f when it forwarded this protocol message to the other processes by invoking $\mathsf{fifo\_broadcast}$ $\text{FORWARD}(m, sd, sn_{sd}, p_f, sn_f)$ (line 11).

Operation scd_broadcast() When p_i invokes scd_broadcast(m), where m is an application message, it sends the protocol message FORWARD(m, i, sn_i, i, sn_i) to itself (this simplifies the writing of the algorithm), and waits until it has no more message from itself pending in $buffer_i$, which means it has scd-delivered a set containing m.

Uniform fifo-broadcast of a message FORWARD When a process p_i fifo-delivers a protocol message FORWARD$(m, sd, sn_{sd}, f, sn_f)$, it first invokes the internal operation forward$(m, sd, sn_{sd}, f, sn_f)$. In addition to other statements, the first fifo-delivery of such a message by a process p_i entails its participation in the uniform reliable fifo-broadcast of this message (lines 5 and 11). In addition to the invocation of forward(), the fifo-delivery of FORWARD() invokes also try_deliver(), which strives to scd-deliver a message set (lines 4).

operation scd_broadcast(m) **is**
(1) send FORWARD(m, sn_i, i, sn_i, i) to itself;
(2) wait$(\nexists\, qdplt \in buffer_i : qdplt.sd = i)$.

when the message FORWARD$(m, sd, sn_{sd}, f, sn_f)$ **is fifo-delivered do** % from p_f
(3) forward$(m, sd, sn_{sd}, f, sn_f)$;
(4) try_deliver().

procedure forward$(m, sd, sn_{sd}, f, sn_f)$ **is**
(5) **if** $(sn_{sd} > clock_i[sd])$
(6) **then if** $(\exists\, qdplt \in buffer_i : qdplt.sd = sd \wedge qdplt.sn = sn_{sd})$
(7) **then** $qdplt.cl[f] \leftarrow sn_f$
(8) **else** $threshold[1..n] \leftarrow [\infty, \ldots, \infty]$; $threshold[f] \leftarrow sn_f$;
(9) **let** $qdplt \leftarrow \langle m, sd, sn_{sd}, threshold[1..n] \rangle$;
(10) $buffer_i \leftarrow buffer_i \cup \{qdplt\}$;
(11) fifo_broadcast FORWARD$(m, sd, sn_{sd}, i, sn_i)$;
(12) $sn_i \leftarrow sn_i + 1$
(13) **end if**
(14) **end if**.

procedure try_deliver() **is**
(15) **let** $to\_deliver_i \leftarrow \{qdplt \in buffer_i : |\{f : qdplt.cl[f] < \infty\}| > \frac{n}{2}\}$;
(16) **while** $(\exists\, qdplt \in to\_deliver_i, \exists\, qdplt' \in buffer_i \setminus to\_deliver_i : |\{f : qdplt.cl[f] < qdplt'.cl[f]\}| \leq \frac{n}{2})$ **do**
$to\_deliver_i \leftarrow to\_deliver_i \setminus \{qdplt\}$ **end while**;
(17) **if** $(to\_deliver_i \neq \emptyset)$
(18) **then for each** $qdplt \in to\_deliver_i$ **do** $clock_i[qdplt.sd] \leftarrow \max(clock_i[qdplt.sd], qdplt.sn)$ **end for**;
(19) $buffer_i \leftarrow buffer_i \setminus to\_deliver_i$;
(20) $ms \leftarrow \{m : \exists\, qdplt \in to\_deliver_i : qdplt.msg = m\}$; scd_deliver$(ms)$
(21) **end if**.

Algorithm 1: An implementation of SCD-broadcast in $\mathcal{CAMP}_{n,t}[t < n/2]$ (code for p_i)

The core of the algorithm Expressed with the relations $\mapsto_i$, $1 \leq i \leq n$, introduced in Section 2, the main issue of the algorithm is to ensure that, if there are two message m and m' and a process p_i such that $m \mapsto_i m'$, then there is no p_j such that $m' \mapsto_j m$.

To this end, a process p_i is allowed to scd-deliver a message m before a message m' only if it knows that a majority of processes p_j have fifo-delivered a message FORWARD$(m, -, -, -)$ before m'; p_i knows it (i) because it fifo-delivered from p_j a message FORWARD$(m, -, -, -, -)$ but not yet a message FORWARD$(m', -, -, -, -)$, or (ii) because it fifo-delivered from p_j both the messages FORWARD$(m, -, -, -, snm)$ and FORWARD$(m', -, -, -, snm')$ and the sending date smn is smaller than the sending date snm'. The MS-Ordering property follows then from the impossibility that a majority of processes "sees m before m'", while another majority "sees m' before m".

Internal operation forward() This operation can be seen as an enrichment (with the fields f and sn_f) of the reliable fifo-broadcast implemented by the messages FORWARD$(m, sd, sn_{sd}, -, -)$. Considering such a message FORWARD$(m, sd, sn_{sd}, f, sn_f)$, m was scd-broadcast by p_{sd} at its local time sn_{sd}, and relayed by the forwarding process p_f at its local time sn_f. If $sn_{sd} \leq clock_i[sd]$, p_i has already scd-delivered a message set containing m (see lines 18 and 20). If $sn_{sd} > clock_i[sd]$, there are two cases defined by the predicate of line 6.

- There is no quadruplet $qdplt$ in $buffer_i$ such that $qdplt.msg = m$. In this case, p_i creates a quadruplet associated with m, and adds it to $buffer_i$ (lines 8-10). Then, p_i participates in the fifo-broadcast of m (line 11) and records its local progress by increasing sn_i (line 12).
- There is a quadruplet $qdplt$ in $buffer_i$ associated with m, i.e., $qdplt = \langle m, -, -, - \rangle \in buffer_i$. In this case, p_i assigns sn_f to $qdplt.cl[f]$ (line 7), thereby indicating that m was known and forwarded by p_f at its local time sn_f.

Internal operation try_deliver() When it executes try_deliver(), p_i first computes the set $to\_deliver_i$ of the quadruplets $qdplt$ containing application messages m which have been seen by a majority of processes (line 15). From p_i's point of view, a message has been seen by a process p_f if $qdplt.cl[f]$ has been set to a finite value (line 7).

As indicated in a previous paragraph, if a majority of processes received first a message FORWARD carrying m' and later another message FORWARD carrying m, it might be that some process p_j scd-delivered a set containing m' before scd-delivering a set containing m. Therefore, p_i must avoid scd-delivering a set containing m before scd-delivering a set containing m'. This is done at line 16, where p_i withdraws the quadruplet $qdplt$ corresponding to m if it has not enough information to deliver m' (i.e. the corresponding $qdplt'$ is not in $to\_deliver_i$) or it does not have the proof that the situation cannot happen, i.e. no majority of processes saw the message corresponding to $qdplt$ before the message corresponding to $qdplt'$ (this is captured by the predicate $|\{f : qdplt.cl[f] < qdplt'.cl[f]\}| \leq \frac{n}{2}$).

If $to\_deliver_i$ is not empty after it has been purged (lines 16-17), p_i computes a message set to scd-deliver. This set ms contains all the application messages in the quadruplets of $to\_deliver_i$ (line 20). These quadruplets are withdrawn from $buffer_i$ (line 18). Moreover, before this scd-delivery, p_i needs to updates $clock_i[x]$ for all the entries such that $x = qdplt.sd$ where $qdplt \in to\_deliver_i$ (line 18). This update is needed to ensure that the future uses of the predicate of line 17 are correct.

3.3 Cost and proof of correctness

Lemma 1 *If a process scd-delivers a message set containing m, some process invoked* scd_broadcast(m).

Proof If a process p_i scd-delivers a set containing a message m, it previously added into $buffer_i$ a quadruplet $qdplt$ such that $qdplt.msg = m$ (line 10), for which it follows that it fifo-delivered a protocol message FORWARD$(m, -, -, -, -)$. Due to the fifo-validity property, it follows that a process generated the fifo-broadcast of this message, which originated from an invocation of scd_broadcast(m). $\Box_{Lemma\ 1}$

Lemma 2 *No process scd-delivers the same message twice.*

Proof Let us observe that, due to the wait statement at line 2, and the increase of sn_i at line 15 between two successive scd-broadcast by a process p_i, no two application messages can have the same identity $\langle i, sn \rangle$. It follows that there is a single quadruplet $\langle m, i, sn, - \rangle$ that can be added to $buffer_i$, and this is done only once (line 10). Finally, let us observe that this quadruplet is suppressed from $buffer_i$, just before m is scd-delivered (line 19-20), which concludes the proof of the lemma. $\Box_{Lemma\ 2}$

Lemma 3 *If a process p_i executes* fifo_broadcast FORWARD$(m, sd, sn_{sd}, i, sn_i)$ *(i.e., executes line* 19*), each non-faulty process p_j executes once* fifo_broadcast FORWARD$(m, sd, sn_{sd}, j, sn_j)$*.*

Proof First, we prove that p_j broadcasts a message FORWARD$(m, sd, sn_{sd}, j, sn_j)$. As p_i is non-faulty, p_j will eventually receive the message sent by p_i. At that time, if $sn_{sd} > clock_j[sd]$, after the condition on line 6 and whatever its result, $buffer_i$ contains a quadruplet $qdplt$ with $qdplt.sd = sd$ and $qdplt.sn = sn_{sd}$. That $qdplt$ was inserted at line 10 (possibly after the reception of a different message), just before p_j sent a message FORWARD$(m, sd, sn_{sd}, j, sn_j)$ at line 11. Otherwise, $clock_j[sd]$ was incremented on line 18, when validating some $qdplt'$ added to $buffer_j$ after p_j received a (first) message FORWARD$(qdplt'.msg, sd, sn_{sd}, f, clock_f[sd])$ from p_f. Because the messages FORWARD$()$ are fifo-broadcast (hence they are delivered in their sending order), p_{sd} sent message FORWARD$(qdplt.msg, sd, sn_{sd}, sd, sn_{sd})$ before FORWARD$(qdplt'.msg, sd, clock_j[sd], sd, clock_j[sd])$, and all other processes only forward messages, p_j received FORWARD$(qdplt.msg, sd, sn_{sd}, -, -)$ from p_f before the message FORWARD$(qdplt'.msg, sd, clock_j[sd], -, -)$. At that time, $sn_{sd} > clock_j[sd]$, so the previous case applies.

After p_j broadcasts its message FORWARD$(m, sd, sn_{sd}, j, sn_j)$ on line 11, there is a $qdplt \in buffer_j$ with $ts(qdplt) = \langle sd, sn_{sd}\rangle$, until it is removed on line 16 and $clock_j[sd] \geq sn_{sd}$. Therefore, one of the conditions at lines 5 and 6 will stay false for the stamp $ts(qdplt)$ and p_j will never execute line 11 with the same stamp $\langle sd, sn_{sd}\rangle$ later. $\square_{Lemma\ 3}$

Lemma 4 *Let p_i be a process that scd-delivers a set ms_i containing a message m and later scd-delivers a set ms'_i containing a message m'. No process p_j scd-delivers first a set ms'_j containing m' and later a message set ms_j containing m.*

Proof Let us suppose there are two messages m and m' and two processes p_i and p_j such that p_i scd-delivers a set ms_i containing m and later scd-delivers a set ms'_i containing m' and p_j scd-delivers a set ms'_j containing m' and later scd-delivers a set ms_j containing m.

When m is delivered by p_i, there is an element $qdplt \in buffer_i$ such that $qdplt.msg = m$ and because of line 15, p_i has received a message FORWARD$(m, -, -, -, -)$ from more than $\frac{n}{2}$ processes.

- If there is no element $qdplt' \in buffer_i$ such that $qdplt'.msg = m'$, since m' has not been delivered by p_i yet, p_i has not received a message FORWARD$(m', -, -, -, -)$ from any process (lines 10 and 19). Hence, because the communication channels are FIFO, more than $\frac{n}{2}$ processes have sent a message FORWARD$(m, -, -, -, -)$ before sending a message FORWARD$(m', -, -, -, -)$.
- Otherwise, $qdplt' \notin to\_deliver_i$ after line 16. As the communication channels are FIFO, more than half of the processes have sent a message FORWARD$(m, -, -, -, -)$ before a message FORWARD$(m', -, -, -, -)$.

Using the same reasoning, it follows that when m' is delivered by p_j, more than $\frac{n}{2}$ processes have sent a message FORWARD$(m', -, -, -, -)$ before sending a message FORWARD$(m, -, -, -, -)$. There exists a process p_k in the intersection of the two majorities, that has (a) sent FORWARD$(m, -, -, -, -)$ before sending FORWARD$(m', -, -, -, -)$ and (b) sent FORWARD$(m', -, -, -, -)$ before sending a message FORWARD$(m, -, -, -, -)$. However, it follows from Lemma 3 that p_k can send a single message FORWARD$(m', -, -, -, -)$ and a single message FORWARD$(m, -, -, -, -)$, which leads to a contradiction. $\square_{Lemma\ 4}$

Lemma 5 *If a non-faulty process executes* fifo_broadcast FORWARD$(m, sd, sn_{sd}, i, sn_i)$ *(line* 11*), it scd-delivers a message set containing m.*

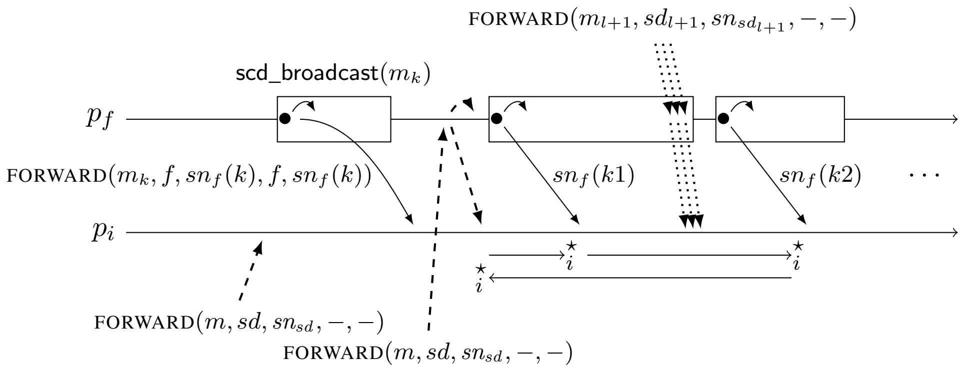

Figure 1: Message pattern introduced in Lemma 5

Proof Let p_i be a non-faulty process. For any pair of messages $qdplt$ and $qdplt'$ ever inserted in $buffer_i$, let $ts = ts(qdplt)$ and $ts' = ts(qdplt')$. Let $\rightarrow_i$ be the dependency relation defined as follows: $ts \rightarrow_i ts' \stackrel{def}{=} |\{j : qdplt'.cl[j] < qdplt.cl[j]\}| \leq \frac{n}{2}$ (i.e. the dependency does not exist if p_i knows that a majority of processes have seen the first update –due to $qdplt'$– before the second –due to $qdplt$). Let $\rightarrow_i^\star$ denote the transitive closure of $\rightarrow_i$.

Let us suppose (by contradiction) that the timestamp $\langle sd, sn_{sd} \rangle$ associated with the message m (carried by the protocol message FORWARD$(m, sd, sn_{sd}, i, sn_i)$ fifo-broadcast by p_i), has an infinity of predecessors according to $\rightarrow_i^\star$. As the number of processes is finite, an infinity of these predecessors have been generated by the same process, let us say p_f. Let $\langle f, sn_f(k) \rangle_{k \in \mathbb{N}}$ be the infinite sequence of the timestamps associated with the invocations of the scd_broadcast() issued by p_f. The situation is depicted by Figure 1.

As p_i is non-faulty, p_f eventually receives a message FORWARD$(m, sd, sn_{sd}, i, sn_i)$, which means p_f broadcast an infinity of messages FORWARD$(m(k), f, sn_f(k), f, sn_f(k))$ after having broadcast the message FORWARD$(m, sd, sn_{sd}, f, sn_f)$. Let $\langle f, sn_f(k1) \rangle$ and $\langle f, sn_f(k2) \rangle$ be the timestamps associated with the next two messages sent by p_f, with $sn_f(k1) < sn_f(k2)$. By hypothesis, we have $\langle f, sn_f(k2) \rangle \rightarrow_i^\star \langle sd, sn_{sd} \rangle$. Moreover, all processes received their first message FORWARD$(m, sd, sn_{sd}, -, -)$ before their first message FORWARD$(m(k), f, sn_f(k), -, -)$, so $\langle sd, sn_{sd} \rangle \rightarrow_i^\star \langle f, sn_f(k1) \rangle$. Let us express the path $\langle f, sn_f(k2) \rangle \rightarrow_i^\star \langle f, sn_f(k1) \rangle$:
$\langle f, sn_f(k2) \rangle = \langle sd'(1), sn'(1) \rangle \rightarrow_i \langle sd'(2), sn'(2) \rangle \rightarrow_i \cdots \rightarrow_i \langle sd(m), sn'(m) \rangle = \langle f, sn_f(k1) \rangle$.

In the time interval starting when p_f sent the message FORWARD$(m(k1), f, sn_f(k1), f, sn_f(k1))$ and finishing when it sent the message FORWARD$(m(k2), f, sn_f(k2), f, sn_f(k2))$, the waiting condition of line 2 became true, so p_f scd-delivered a set containing the message $m(k1)$, and according to Lemma 1, no set containing the message $m(k2)$. Therefore, there is an index l such that process p_f delivered sets containing messages associated with a timestamp $\langle sd'(l), sn'(l) \rangle$ for all $l' > l$ but not for $l' = l$. Because the channels are FIFO and thanks to lines 15 and 16, it means that a majority of processes have sent a message FORWARD$(-, sd'(l+1), sn'(l+1), -, -)$ before a message FORWARD$(-, sd'(l), sn'(l), -, -)$, which contradicts the fact that $\langle sd'(l), sn'(l) \rangle \rightarrow_i \langle sd'(l+1), sn'(l+1) \rangle$.

Let us suppose a non-faulty process p_i has fifo-broadcast a message FORWARD$(m, sd, sn_{sd}, i, sn_i)$ (line 10). It inserted a quadruplet $qdplt$ with timestamp $\langle sd, sn_{sd} \rangle$ on line 9 and by what precedes, $\langle sd, sn_{sd} \rangle$ has a finite number of predecessors $\langle sd_1, sn_1 \rangle, \ldots, \langle sd_l, sn_l \rangle$ according to $\rightarrow_i^\star$. As p_i is non-faulty, according to Lemma 3, it eventually receives a message FORWARD$(-, sd_k, sn_k, -, -)$ for all $1 \leq k \leq l$ and from all non-faulty processes, which are in majority.

Let $pred$ be the set of all quadruplets $qdplt'$ such that $\langle qdplt'.sd, qdplt'.sn \rangle \rightarrow_i^\star \langle sd, sn_{sd} \rangle$. Let us consider the moment when p_i receives the last message $\text{FORWARD}(-, sd_k, sn_k, f, sn_f)$ sent by a correct process p_f. For all $qdplt' \in pred$, either $qdplt'.msg$ has already been delivered or $qdplt'$ is inserted $to\_deliver_i$ on line 15. Moreover, no $qdplt' \in pred$ will be removed from $to\_deliver_i$, on line 16, as the removal condition is the same as the definition of $\rightarrow_i$. In particular for $qdplt' = qdplt$, either m has already been scd-delivered or m is present in $to\_deliver_i$ on line 17 and will be scd-delivered on line 20. $\square_{Lemma\ 5}$

Lemma 6 *If a non-faulty process scd-broadcasts a message m, it scd-delivers a message set containing m.*

Proof If a non-faulty process scd-broadcasts a message m, it previously fifo-broadcast the message $\text{FORWARD}(m, sd, sn_{sd}, i, sn_i)$ at line 11). Then, due to Lemma 5, it scd-delivers a message set containing m. $\square_{Lemma\ 6}$

Lemma 7 *If a process scd-delivers a message m, every non-faulty process scd-delivers a message set containing m.*

Proof Let p_i be a process p_i that scd-delivers a message m. At line 20, there is a quadruplet $qdplt \in to\_deliver_i$ such that $qdplt.msg = m$. At line 15, $qdplt \in buffer_i$, and $qdplt$ was inserted in $buffer_i$ at line 10, just before p_i fifo-broadcast the message $\text{FORWARD}(m, sd, sn_{sd}, i, sn_i)$. By Lemma 3, every non-faulty process p_j sends a message $\text{FORWARD}(m, sd, sn_{sd}, j, sn_j)$, so by Lemma 5, p_j scd-delivers a message set containing m. $\square_{Lemma\ 7}$

Theorem 1 *Algorithm* 1 *implements the* SCD-broadcast *communication abstraction in $\mathcal{CAMP}_{n,t}[t < n/2]$. Moreover, it requires $O(n^2)$ messages per invocation of* scd_broadcast(). *If there is an upper bound Δ on messages transfer delays (and local computation times are equal to zero), each* SCD-broadcast *costs at most 2Δ time units.*

Proof The proof follows from Lemma 1 (Validity), Lemma 2 (Integrity), Lemma 4 (MS-Ordering), Lemma 6 (Termination-1), and Lemma 7 (Termination-2).

The $O(n^2)$ message complexity comes from the fact that, due to the predicates of line 5 and 6, each application message m is forwarded at most once by each process (line 11). The 2Δ follows from the same argument. $\square_{Theorem\ 1}$

The next corollary follows from (i) Theorems 2 and 1, and (ii) the fact that the constraint $(t < n/2)$ is an upper bound on the number of faulty processes to build a read/write register (or snapshot object) [5].

Corollary 1 *Algorithm* 1 *is resiliency optimal.*

4 An SCD-broadcast-based Communication Pattern

All the algorithms implementing concurrent objects and tasks, which are presented in this paper, are based on the same communication pattern, denoted Pattern 1. This pattern involves each process, either as a client (when it invokes an operation), or as a server (when it scd-delivers a message set).

When a process p_i invokes an operation op(), it executes once the lines 1-3 for a task, and 0, 1, or 2 times for an operation on a concurrent object. In this last case, this number of times depends on the consistency condition which is implemented (linearizability [20] or sequential consistency [25]).

All the messages sent by a process p_i are used to synchronize its local data representation of the object, or its local view of the current state of the task. This synchronization is realized by the Boolean

```
operation op() is
      According to the object/task that is implemented, and its consistency condition (if it is an object,
      linearizability vs seq. consistency), execute 0, 1, or 2 times the lines 1-3 where the message type
      TYPE is either a pure synchronization message SYNC or an object/task-dependent message MSG;
(1)   done_i ← false;
(2)   scd_broadcast TYPE(a, b, ..., i);
      a, b, ... are data, and i is the id of the invoking process; a message SYNC carries only the id of its sender;
(3)   wait(done_i);
(4)   According to the states of the local variables, compute a result r; return(r).

when the message set { MSG(..., j_1), ..., MSG(..., j_x)), SYNC(j_{x+1}), ..., SYNC(j_y) } is scd-delivered do
(5)   for each message m = MSG(..., j) do statements specific to the object/task that is implemented end for;
(6)   if ∃ℓ : j_ℓ = i then done_i ← true end if.
```

Pattern 1: Communication pattern (Code for p_i)

$done_i$ and the parameter i carried by every message (lines 1, 3, and 6): p_i is blocked until the message it scd-broadcast just before is scd-delivered. The values carried by a message MSG are related to the object/task that is implemented, and may require local computation.

It appears that the combination of this communication pattern and the properties of SCD-broadcast provides us with a single simple framework that allows for correct implementations of both concurrent objects and tasks.

The next three sections describe algorithms implementing a snapshot object, a counter object, and the lattice agreement task, respectively. All these algorithms consider the system model $\mathcal{CAMP}_{n,t}[\emptyset]$ enriched with the SCD-broadcast communication abstraction, denoted $\mathcal{CAMP}_{n,t}$[SCD-broadcast], and use the previous communication pattern.

5 SCD-broadcast in Action (its Power): Snapshot Object

5.1 Snapshot object

Definition The snapshot object was introduced in [1, 3]. A snapshot object is an array $REG[1..m]$ of atomic read/write registers which provides the processes with two operations, denoted $\mathsf{write}(r, -)$ and $\mathsf{snapshot}()$. The invocation of $\mathsf{write}(r, v)$, where $1 \leq r \leq m$, by a process p_i assigns atomically v to $REG[r]$. The invocation of $\mathsf{snapshot}()$ returns the value of $REG[1..m]$ as if it was executed instantaneously. Hence, in any execution of a snapshot object, its operations $\mathsf{write}()$ and $\mathsf{snapshot}()$ are linearizable.

The underlying atomic registers can be Single-Reader (SR) or Multi-Reader (MR) and Single-Writer (SR) or Multi-Writer (MW). We consider only SWMR and MWMR registers. If the registers are SWMR the snapshot is called SWMR snapshot (and we have then $m = n$). Moreover, we always have $r = i$, when p_i invokes $\mathsf{write}(r, -)$. If the registers are MWMR, the snapshot object is called MWMR.

Implementations based on read/write registers Implementations of both SWMR and MWMR snapshot objects on top of read/write atomic registers have been proposed (e.g., [1, 3, 22, 23]). The "hardness" to build snapshot objects in read/write systems and associated lower bounds are presented in the survey [15]. The best algorithm known to implement an SWMR snapshot requires $O(n \log n)$ read/write on the base SWMR registers for both the $\mathsf{write}()$ and $\mathsf{snapshot}()$ operations [7]. As far as MWMR snapshot objects are concerned, there are implementations where each operation has an $O(n)$ cost[3].

[3]Snapshot objects built in read/write models enriched with operations such as Compare&Swap, or LL/SC, have also been considered, e.g.,[24, 22]. Here we are interested in pure read/write models.

As far as the construction of an SWMR (or MWMR) snapshot object in crash-prone asynchronous message-passing systems where $t < n/2$ is concerned, it is possible to stack two constructions: first an algorithm implementing SWMR (or MWMR) atomic read/write registers (e.g., [5])), and, on top of it, an algorithm implementing an SWMR (or MWMR) snapshot object. This stacking approach provides objects whose operation cost is $O(n^2 \log n)$ messages for SWMR snapshot, and $O(n^2)$ messages for MWMR snapshot. An algorithm based on the same low level communication pattern as the one used in [5], which builds an atomic SWMR snapshot object "directly" (i.e., without stacking algorithms) was recently presented in [14] (the aim of this algorithm is to perform better that the stacking approach in concurrency-free executions).

5.2 An algorithm for atomic MWMR snapshot in $\mathcal{CAMP}_{n,t}$[SCD-broadcast]

Local representation of REG at a process p_i At each register p_i, $REG[1..m]$ is represented by three local variables $reg_i[1..m]$ (data part), plus $tsa_i[1..m]$ and $done_i$ (control part).

- $done_i$ is a Boolean variable.
- $reg_i[1..m]$ contains the current value of $REG[1..m]$, as known by p_i.
- $tsa_i[1..m]$ is an array of timestamps associated with the values stored in $reg_i[1..m]$. A timestamp is a pair made of a local clock value and a process identity. Its initial value is $\langle 0, - \rangle$. The fields associated with $tsa_i[r]$ are denoted $\langle tsa_i[r].date, tsa_i[r].proc \rangle$.

Timestamp-based order relation We consider the classical lexicographical total order relation on timestamps, denoted $<_{ts}$. Let $ts1 = \langle h1, i1 \rangle$ and $ts2 = \langle h2, i2 \rangle$. We have $ts1 <_{ts} ts2 \stackrel{def}{=} (h1 < h2) \vee ((h1 = h2) \wedge (i1 < i2))$.

Algorithm 2: snapshot operation This algorithm consists of one instance of the communication pattern introduced in Section 4 (line 1), followed by the return of the local value of $reg_i[1..m]$ (line 2). The message $\text{SYNC}(i)$, which is scd-broadcast is a pure synchronization message, whose aim is to entail the refreshment of the value of $reg_i[1..m]$ (lines 5-11) which occurs before the setting of $done_i$ to `true` (line 12).

operation snapshot() **is**
(1) $done_i \leftarrow$ `false`; scd_broadcast $\text{SYNC}(i)$; wait$(done_i)$;
(2) return$(reg_i[1..m])$.

operation write(r, v) **is**
(3) $done_i \leftarrow$ `false`; scd_broadcast $\text{SYNC}(i)$; wait$(done_i)$;
(4) $done_i \leftarrow$ `false`; scd_broadcast$\text{WRITE}(r, v, \langle tsa_i[r].date + 1, i \rangle)$; wait$(done_i)$.

when the message set $\{$ $\text{WRITE}(r_{j_1}, v_{j_1}, \langle date_{j_1}, j_1 \rangle)$, $\ldots$, $\text{WRITE}(r_{j_x}, v_{j_x}, \langle date_{j_x}, j_x \rangle)$, $\text{SYNC}(j_{x+1})$, $\ldots$, $\text{SYNC}(j_y)$ $\}$ **is scd-delivered do**
(5) **for each** r such that $\text{WRITE}(r, -, -) \in$ scd-delivered message set **do**
(6) **let** $\langle date, writer \rangle$ be the greatest timestamp in the messages $\text{WRITE}(r, -, -)$;
(7) **if** $(tsa_i[r] <_{ts} \langle date, writer \rangle)$
(8) **then let** v the value in $\text{WRITE}(r, -, \langle date, writer \rangle)$;
(9) $reg_i[r] \leftarrow v$; $tsa_i[r] \leftarrow \langle date, writer \rangle$
(10) **end if**
(11) **end for**;
(12) **if** $\exists \ell \, : \, j_\ell = i$ **then** $done_i \leftarrow$ `true` **end if**.

Algorithm 2: Construction of an MWMR snapshot object $\mathcal{CAMP}_{n,t}$[SCD-broadcast] (code for p_i)

Algorithm 2: write operation (Lines 3-4) When a process p_i wants to assign a value v to $REG[r]$, it invokes $REG.\mathsf{write}(r, v)$. This operation is made up of two instances of the communication pattern. The first one is a re-synchronization (line 3), as in the snapshot operation, whose side effect is here to provide p_i with an up-to-date value of $tsa_i[r].date$. In the second instance of the communication pattern, p_i associates the timestamp $\langle tsa_i[r].date + 1, i\rangle$ with v, and scd-broadcasts the data/control message $\text{WRITE}(r, v, \langle tsa_i[r].date + 1, i\rangle)$. In addition to informing the other processes on its write of $REG[r]$, this message $\text{WRITE}()$ acts as a re-synchronization message, exactly as a message $\text{SYNC}(i)$. When this synchronization terminates (i.e., when the Boolean $done_i$ is set to `true`), p_i returns from the write operation.

Algorithm 2: scd-delivery of a set of messages When p_i scd-delivers a message set, namely,

$\{\ \text{WRITE}(r_{j_1}, v_{j_1}, \langle date_{j_1}, j_1\rangle),\ \ldots,\ \text{WRITE}(r_{j_x}, v_{j_x}, \langle date_{j_x}, j_x\rangle), \text{SYNC}(j_{x+1}),\ \ldots,\ \text{SYNC}(j_y)\ \}$

it first looks if there are messages $\text{WRITE}()$. If it is the case, for each register $REG[r]$ for which there are messages $\text{WRITE}(r, -, -)$ (line 5), p_i computes the maximal timestamp carried by these messages (line 6), and updates accordingly its local representation of $REG[r]$ (lines 7-10). Finally, if p_i is the sender of one of these messages ($\text{WRITE}()$ or $\text{SYNC}()$), $done_i$ is set to `true`, which terminates p_i's re-synchronization (line 12).

Time and Message costs An invocation of $\mathsf{snapshot}()$ involves one invocation of $\mathsf{scd\_broadcast}()$, while an invocation of $\mathsf{write}()$ involves two such invocations. As $\mathsf{scd\_broadcast}()$ costs $O(n^2)$ protocol messages and 2Δ time units, $\mathsf{snapshot}()$ cost the same, and $\mathsf{write}()$ costs the double.

5.3 Proof of Algorithm 2

As they are implicitly used in the proofs that follow, let us recall the properties of the SCD-broadcast abstraction. The non-faulty processes scd-deliver the same messages (exactly one each), and each of them was scd-broadcast. As a faulty process behaves correctly until it crashes, it scd-delivers a subset of the messages scd-delivered by the non-faulty processes.

Without loss of generality, we assume that there is an initial write operation issued by a non-faulty process. Moreover, if a process crashes in a snapshot operation, its snapshot is not considered; if a process crashes in a write operation, its write is considered only if the message WRITE() it sent at line 4 is scd-delivered to at least one non-faulty process (and by the Termination-2 property, at least to all non-faulty processes). Let us notice that a message SYNC() scd-broadcast by a process p_i does not modify the local variables of the other processes.

Lemma 8 *If a non-faulty process invokes an operation, it returns from its invocation.*

Proof Let p_i be a non-faulty process that invokes a read or write operation. By the Termination-1 property of SCD-broadcast, it eventually receives a message set containing the message SYNC() or WRITE() it sends at line 2, 3 or 4. As all the statements associated with the scd-delivery of a message set (lines 5-12) terminate, it follows that the synchronization Boolean $done_i$ is eventually set to `true`. Consequently, p_i returns from the invocation of its operation. $\square_{Lemma\ 8}$

Extension of the relation $<_{ts}$ The relation $<_{ts}$ is extended to a partial order on arrays of timestamps, denoted $\leq_{tsa}$, defined as follows: $tsa1[1..m] \leq_{tsa} tsa2[1..m] \stackrel{def}{=} \forall r : (tsa1[r] = tsa2[r] \vee tsa1[r] <_{ts} tsa2[r])$. Moreover, $tsa1[1..m] <_{tsa} tsa2[1..m] \stackrel{def}{=} (tsa1[1..m] \leq_{tsa} tsa2[1..m]) \wedge (tsa1[1..m] \neq tsa2[1..m])$.

Definition Let TSA_i be the set of the array values taken by $ts_i[1..m]$ at line 12 (end of the processing of a message set by process p_i). Let $TSA = \cup_{1 \leq i \leq n} TSA_i$.

Lemma 9 *The order $\leq_{tsa}$ is total on TSA.*

Proof Let us first observe that, for any i, all values in TSA_i are totally ordered (this comes from $ts_i[1..m]$ whose entries can only increase, lines 7 and 10). Hence, let $tsa1[1..m]$ be an array value of TSA_i, and $tsa2[1..m]$ an array value of TSA_j, where $i \neq j$.

Let us assume, by contradiction, that $\neg(tsa1 \leq_{tsa} tsa2)$ and $\neg(tsa2 \leq_{tsa} tsa1)$. As $\neg(tsa1 \leq_{tsa} tsa2)$, there is a registers r such that $tsa2[r] < tsa1[r]$. According to lines 7 and 9, there is a message $\text{WRITE}(r, -, tsa1[r])$ received by p_i when $tsa_i = tsa1$ and not received by p_j when $tsa_j = tsa2$ (because $tsa2[r] < tsa1[r]$). Similarly, there is a message $\text{WRITE}(r', -, tsa2[r'])$ received by p_j when $tsa_j = tsa2$ and not received by p_i when $tsa_i = tsa1$. This situation contradicts the MS-Ordering property, from which we conclude that either $tsa1 \leq_{tsa} tsa2$ or $tsa2 \leq_{tsa} tsa1$. $\Box_{Lemma\ 9}$

Definitions Let us associate a timestamp $ts(\text{write}(r, v))$ with each write operation as follows. Let p_i be the invoking process; $ts(\text{write}(r, v))$ is the timestamp of v as defined by p_i at line 4, i.e., $\langle tsa_i[r].date + 1, i\rangle$.

Let op1 and op2 be any two operations. The relation $\prec$ on the whole set of operations is defined as follows: $\text{op1} \prec \text{op2}$ if op1 terminated before op2 started. It is easy to see that $\prec$ is a real-time-compliant partial order on all the operations.

Lemma 10 *No two distinct write operations on the same register* $\text{write1}(r, v)$ *and* $\text{write2}(r, w)$ *have the same timestamp, and* $(\text{write1}(r, v) \prec \text{write2}(r, w)) \Rightarrow (ts(\text{write1}) <_{ts} ts(\text{write2}))$.

Proof Let $\langle date1, i\rangle$ and $\langle date2, j\rangle$ be the timestamp of $\text{write1}(r, v)$ and $\text{write2}(r, w)$, respectively. If $i \neq j$, $\text{write1}(r, v)$ and $\text{write2}(r, w)$ have been produced by different processes, and their timestamp differ at least in their process identity.

So, let us consider that the operations have been issued by the same process p_i, with $\text{write1}(r, v)$ first. As $\text{write1}(r, v)$ precedes $\text{write2}(r, w)$, p_i first invoked scd_broadcast $\text{WRITE}(r, v, \langle date1, i\rangle)$ (line 4) and later $\text{WRITE}(r, w, \langle date2, i\rangle)$. It follows that these SCD-broadcast invocations are separated by a local reset of the Boolean $done_i$ at line 4. Moreover, before the reset of $done_i$ due to the scd-delivery of the message $\{\ldots, \text{WRITE}(r, v, \langle date1, i\rangle), \ldots\}$, we have $tsa_i[r].date_i \geq date1$ (lines 6-10). Hence, we have $tsa_i[r].date \geq date1$ before the reset of $done_i$ (line 12). Then, due to the "+1" at line 4, $\text{WRITE}(r, w, \langle date2, i\rangle)$ is such that $date2 > date1$, which concludes the proof of the first part of the lemma.

Let us now consider that $\text{write1}(r, v) \prec \text{write2}(r, w)$. If $\text{write1}(r, v)$ and $\text{write2}(r, w)$ have been produced by the same process we have $date1 < date2$ from the previous reasoning. So let us assume that they have been produced by different processes p_i and p_j. Before terminating $\text{write1}(r, v)$ (when the Boolean $done_i$ is set `true` at line 12), p_i received a message set $ms1_i$ containing the message $\text{WRITE}(r, v, \langle date1, i\rangle)$. When p_j executes $\text{write2}(r, w)$, it first invokes scd_broadcast $\text{SYNC}(j)$ at line 3. Because $\text{write1}(r, v)$ terminated before $\text{write2}(r, w)$ started, this message $\text{SYNC}(j)$ cannot belong to $ms1_i$.

Due to Integrity and Termination-2 of SCD-broadcast, p_j eventually scd-delivers exactly one message set $ms1_j$ containing $\text{WRITE}(r, v, \langle date1, i\rangle)$. Moreover, it also scd-delivers exactly one message set $ms2_j$ containing its own message $\text{SYNC}(j)$. On the the other side, p_i scd-delivers exactly one message set $ms2_i$ containing the message $\text{SYNC}(j)$. It follows from the MS-Ordering property that, if $ms2_j \neq ms1_j$, p_j cannot scd-deliver $ms2_j$ before $ms1_j$. Then, whatever the case ($ms1_j = ms2_j$ or $ms1_j$ is scd-delivered at p_j before $ms2_j$), it follows from the fact that the messages $\text{WRITE}()$ are processed (lines 5-11) before the messages $\text{SYNC}(j)$ (line 12), that we have $tsa_j[r] \geq \langle date1, i\rangle$ when $done_j$ is set to `true`. It then follows from line 4 that $date2 > date1$, which concludes the proof of the lemma. $\Box_{Lemma\ 10}$

Associating timestamp arrays with operations Let us associate a timestamp array $tsa(\mathsf{op})[1..m]$ with each operation $\mathsf{op}()$ as follows.

- Case $\mathsf{op}() = \mathsf{snapshot}()$. Let p_i be the invoking process; $tsa(\mathsf{op})$ is the value of $tsa_i[1..m]$ when p_i returns from the snapshot operation (line 2).
- Case $\mathsf{op}() = \mathsf{write}(r, v)$. Let $\mathsf{min}_{tsa}(\{A\})$, where A is a set of array values, denote the smallest array value of A according to $<_{tsa}$. Let $tsa(\mathsf{op}) \stackrel{def}{=} \mathsf{min}_{tsa}(\{tsa[1..m] \in TSA$ such that $ts(\mathsf{op}) \leq_{ts} tsa[r]\})$. Hence, $tsa(\mathsf{op})$ is the first $tsa[1..m]$ of TSA, that reports the operation $\mathsf{op}() = \mathsf{write}(r, v)$.

Lemma 11 *Let* op *and* op' *be two distinct operations such that* $\mathsf{op} \prec \mathsf{op}'$. *We have* $tsa(\mathsf{op}) \leq_{tsa} tsa(\mathsf{op}')$. *Moreover, if* op' *is a write operation, we have* $tsa(\mathsf{op}) <_{tsa} tsa(\mathsf{op}')$.

Proof Let p_i and p_j be the processes that performed op and op', respectively. Let SYNC_j be the $\text{SYNC}(j)$ message sent by p_j (at line 2 or 3) during the execution of op'. Let $term\_tsa_i$ be the value of $tsa_i[1..m]$ when op terminates (line 2 or 4), and $sync\_tsa_j$ the value of $tsa_j[1..m]$ when $done_j$ becomes true for the first time after p_j sent SYNC_j (line 1 or 3). Let us notice that $term\_tsa_i$ and $sync\_tsa_j$ are elements of the set TSA.

According to lines 7 and 10, for all r, $tsa_i[r]$ is the largest timestamp carried by a message $\text{WRITE}(r, v, -)$ received by p_i in a message set before op terminates. Let m be a message such that there is a set sm scd-delivered by p_i before it terminated op. As p_j sent SYNC_j after p_i terminated, p_i did not receive any set containing SYNC_j before it terminated op. By the properties Termination-2 and MS-Ordering, p_j received message m in the same set as SYNC_j or in a message set sm' received before the set containing SYNC_j. Therefore, we have $term\_tsa_i \leq_{tsa} sync\_tsa_j$.

If op is a snapshot operation, then $tsa(\mathsf{op}) = term\_tsa_i$. Otherwise, $\mathsf{op}() = \mathsf{write}(r, v)$. As p_i has to wait until it processes a set of messages including its $\text{WRITE}()$ message (and executes line 12), we have $ts(\mathsf{op}) <_{ts} term\_tsa_i[r]$. Finally, due to the fact that $term\_tsa_i \in TSA$ and Lemma 9, we have $tsa(\mathsf{op}) \leq_{tsa} term\_tsa_i$.

If op' is a snapshot operation, then $sync\_tsa_j = tsa(\mathsf{op}')$ (line 2). Otherwise, $\mathsf{op}() = \mathsf{write}(r, v)$ and thanks to the $+1$ in line 4, $sync\_tsa_j[r]$ is strictly smaller than $tsa(\mathsf{op}')[r]$ which, due to Lemma 9, implies $sync\_tsa_j <_{tsa} tsa(\mathsf{op}')$.

It follows that, in all cases, we have $tsa(\mathsf{op}) \leq_{tsa} term\_tsa_i \leq_{tsa} sync\_tsa_j \leq_{tsa} tsa(\mathsf{op}')$ and if op' is a write operation, we have $tsa(\mathsf{op}) \leq_{tsa} term\_tsa_i \leq_{tsa} sync\_tsa_j <_{tsa} tsa(\mathsf{op}')$, which concludes the proof of the lemma. $\Box_{Lemma\ 11}$

The previous lemmas allow the operations to be linearized (i.e., totally ordered in an order compliant with both the sequential specification of a register, and their real-time occurrence order) according to a total order extension of the reflexive and transitive closure of the $\rightarrow_{lin}$ relation defined thereafter.

Definition 1 *Let* $\mathsf{op}, \mathsf{op}'$ *be two operations. We define the* $\rightarrow_{lin}$ *relation by* $\mathsf{op} \rightarrow_{lin} \mathsf{op}'$ *if one of the following properties holds:*

- $\mathsf{op} \prec \mathsf{op}'$,
- $tsa(\mathsf{op}) <_{tsa} tsa(\mathsf{op}')$,
- $tsa(\mathsf{op}) = tsa(\mathsf{op}')$, *op is a write operation and* op' *is a snapshot operation,*
- $tsa(\mathsf{op}) = tsa(\mathsf{op}')$, op *and* op' *are two write operations on the same register and* $ts(\mathsf{op}) <_{ts} ts(\mathsf{op}')$,

Lemma 12 *The snapshot object built by Algorithm 2 is linearizable.*

Proof We recall the definition of the $\rightarrow_{lin}$ relation: $\mathsf{op} \rightarrow_{lin} \mathsf{op}'$ if one of the following properties holds:

- $\mathsf{op} \prec \mathsf{op}'$,
- $tsa(\mathsf{op}) <_{tsa} tsa(\mathsf{op}')$,
- $tsa(\mathsf{op}) = tsa(\mathsf{op}')$, op is a write operation and op' is a snapshot operation,
- $tsa(\mathsf{op}) = tsa(\mathsf{op}')$, op and op' are two write operations on the same register and $ts(\mathsf{op}) <_{ts} ts(\mathsf{op}')$,

We define the $\rightarrow^{\star}_{lin}$ relation as the reflexive and transitive closure of the $\rightarrow_{lin}$ relation.

Let us prove that the $\rightarrow^{\star}_{lin}$ relation is a partial order on all operations. Transitivity and reflexivity are given by construction. Let us prove antisymmetry. Suppose there are $\mathsf{op}_0, \mathsf{op}_2, ..., \mathsf{op}_m$ such that $\mathsf{op}_0 = \mathsf{op}_m$ and $\mathsf{op}_i \rightarrow_{lin} \mathsf{op}_{i+1}$ for all $i < m$. By Lemma 11, for all $i < m$, we have $tsa(\mathsf{op}_i) \leq_{tsa} tsa(\mathsf{op}_{i+1})$, and $tsa(\mathsf{op}_m) = tsa(\mathsf{op}_0)$, so the timestamp array of all operations are the same. Moreover, if op_i is a snapshot operation, then $\mathsf{op}_i \prec \mathsf{op}_{(i+1)\%m}$ is the only possible case ($\%$ stands for "modulo") , and by Lemma 11 again, $\mathsf{op}_{(i+1)\%m}$ is a snapshot operation. Therefore, only two cases are possible.

- Let us suppose that all the op_i are snapshot operations and for all i, $\mathsf{op}_i \prec \mathsf{op}_{(i+1)\%m}$. As $\prec$ is a partial order relation, it is antisymmetric, so all the op_i are the same operation.
- Otherwise, all the op_i are write operations. By Lemma 11, for all $\mathsf{op}_i \not\prec \mathsf{op}_{(i+1)\%m}$. The operations op_i and $\mathsf{op}_{i+1\%m}$ are ordered by the fourth point, so they are write operations on the same register and $ts(\mathsf{op}_i) <_{ts} ts(\mathsf{op}_{i+1\%m})$. By antisymmetry of the $<_{ts}$ relation, all the op_i have the same timestamp, so by Lemma 10, they are the same operation, which proves antisymmetry.

Let $\leq_{lin}$ be a total order extension of $\rightarrow^{\star}_{lin}$. Relation $\leq_{lin}$ is real-time compliant because $\rightarrow^{\star}_{lin}$ contains $\prec$.

Let us consider a snapshot operation op and a register r such that $tsa(\mathsf{op})[r] = \langle date1, i\rangle$. According to line 4, it is associated to the value v that is returned by $\mathsf{read1}()$ for r, and comes from a $\text{WRITE}(r, v, \langle date1, i\rangle)$ message sent by a write operation $\mathsf{op}_r = \mathsf{write}(r, v)$. By definition of $tsa(\mathsf{op}_r)$, we have $tsa(\mathsf{op}_r) \leq_{tsa} tsa(\mathsf{op})$ (Lemma 11), and therefore $\mathsf{op}_r \leq_{lin} \mathsf{op}$. Moreover, for any different write operation op'_r on r, by Lemma 10, $ts(\mathsf{op}'_r) \neq ts(\mathsf{op}_r)$. If $ts(\mathsf{op}'_r) <_{ts} ts(\mathsf{op}_r)$, then $\mathsf{op}'_r \leq_{lin} \mathsf{op}_r$. Otherwise, $tsa(\mathsf{op}) <_{tsa} tsa(\mathsf{op}'_r)$, and (due to the first item of the definition of $\rightarrow_{lin}$) we have $\mathsf{op} \leq_{lin} \mathsf{op}'_r$. In both cases, the value written by op_r is the last value written on r before op, according to $\leq_{lin}$. $\square_{Lemma\ 12}$

Theorem 2 *Algorithm 2 builds an* MWMR *atomic snapshot object in the model* $\mathcal{CAMP}_{n,t}$[SCD-broadcast]. *The operation* $\mathsf{snapshot}$ *costs one* SCD-broaddast, *the* $\mathsf{write}()$ *operation costs two.*

Proof The proof follows from Lemmas 8-12. The cost of the operation $\mathsf{snapshot}()$ follows from line 1, and the one of $\mathsf{write}()$ follows from lines 3-4. $\square_{Theorem\ 2}$

Comparison with other algorithms Interestingly, Algorithm 2 is more efficient (from both time and message point of views) than the stacking of a read/write snapshot algorithm running on top of a message-passing emulation of a read/write atomic memory (such a stacking would costs $O(n^2 \log n)$ messages and $O(n\Delta)$ time units, see Section 5.1).

Sequentially consistent snapshot object When considering Algorithm 2, let us suppress line 1 and line 3 (i.e., the messages SYNC are suppressed). The resulting algorithm implements a sequentially consistent snapshot object. This results from the suppression of the real-time compliance due to the messages SYNC. The operation $\mathsf{snapshot}()$ is purely local, hence its cost is 0. The cost of the operation $\mathsf{write}()$ is one SCD-broadcast, i.e., 2Δ time units and n^2 protocol messages. The proof of this algorithm is left to the reader.

6 SCD-broadcast in Action (its Power): Counter Object

Definition Let a *counter* be an object which can be manipulated by three parameterless operations: increase(), decrease(), and read(). Let C be a counter. From a sequential specification point of view C.increase() adds 1 to C, C.decrease() subtracts 1 from C, C.read() returns the value of C. As indicated in the Introduction, due to its commutative operations, this object is a good representative of a class of CRDT objects (*conflict-free replicated data type* as defined in [32]).

operation increase() **is**
(1) $done_i \leftarrow$ `false`; scd_broadcast $\text{PLUS}(i)$; wait$(done_i)$;
(2) return().

operation decrease() **is** the same as increase() where $\text{PLUS}(i)$ is replaced by $\text{MINUS}(i)$.

operation read() **is**
(3) $done_i \leftarrow$ `false`; scd_broadcast $\text{SYNC}(i)$; wait$(done_i)$;
(4) return$(counter_i)$.

when the message set $\{\ \text{PLUS}(j_1),\ \ldots, \text{MINUS}(j_x),\ \ldots,\ \text{SYNC}(j_y),\ \ldots\ \}$ **is scd-delivered do**
(5) **let** p $=$ number of messages PLUS() in the message set;
(6) **let** m $=$ number of messages MINUS() in the message set;
(7) $counter_i \leftarrow counter_i + p - m$;
(8) **if** $\exists \ell\ :\ j_\ell = i$ **then** $done_i \leftarrow$ `true` **end if**.

Algorithm 3: Construction of an atomic counter in $\mathcal{CAMP}_{n,t}$[SCD-broadcast] (code for p_i)

An algorithm satisfying linearizability Algorithm 3 implements an atomic counter C. Each process manages a local copy of it denoted $counter_i$. The text of the algorithm is self-explanatory.

The operation read() is similar to the operation snapshot() of the snapshot object. Differently from the write() operation on a snapshot object (which requires a synchronization message SYNC() and a data/synchronization message WRITE()), the update operations increase() and decrease() require only one data/synchronization message PLUS() or MINUS(). This is the gain obtained from the fact that, from a process p_i point of view, the operations increase() and decrease() which appear between two consecutive of its read() invocations are commutative.

Lemma 13 *If a non-faulty process invokes an operation, it returns from its invocation.*

Proof Let p_i be a non-faulty process that invokes an increase(), decrease() or read() operation. By the Termination-1 property of SCD-broadcast, it eventually receives a message set containing the message PLUS(), MINUS() or SYNC() it sends at line 1 or 3. As all the statements associated with the scd-delivery of a message set (lines 5-8) terminate, it follows that the synchronization Boolean $done_i$ is eventually set to `true`. Consequently, p_i returns from the invocation of its operation. $\square_{Lemma\ 13}$

Definition 2 *Let* op_i *be an operation performed by* p_i. *We define* $past(\mathsf{op}_i)$ *as a set of messages by:*

- *If* op_i *is an* increase() *or* decrease() *operation, and* m_i *is the message sent during its execution at line* 1*, then* $past(\mathsf{op}_i) = \{m : m \mapsto m_i\}$.
- *If* op_i *is a* read() *operation, then* $past(\mathsf{op}_i)$ *is the union of all sets of messages* $scd\_delivered$ *by* p_i *before it executed line* 4.

We define the $\rightarrow_{lin}$ *relation by* $\mathsf{op} \rightarrow_{lin} \mathsf{op}'$ *if one of the following conditions hold:*

- $past(\mathsf{op}) \subsetneq past(\mathsf{op}')$;

- $past(\mathsf{op}) = past(\mathsf{op}')$, *$\mathsf{op}$ is an* increase() *or a* decrease() *operation and* op' *is a* read() *operation.*

Lemma 14 *The counter object built by Algorithm* 3 *is linearizable.*

Proof Let us prove that $\rightarrow_{lin}$ is a strict partial order relation. Let us suppose $\mathsf{op} \rightarrow_{lin} \mathsf{op}' \rightarrow_{lin} \mathsf{op}''$. If op' is a read() operation, we have $past(\mathsf{op}) \subseteq past(\mathsf{op}') \subsetneq past(\mathsf{op}'')$. If op' is an increase() or a decrease() operation, we have $past(\mathsf{op}) \subsetneq past(\mathsf{op}') \subseteq past(\mathsf{op}'')$. In both cases, we have $past(\mathsf{op}) \subsetneq past(\mathsf{op}'')$, which proves transitivity as well as antisymmetry and irreflexivity since it is impossible to have $past(\mathsf{op}) \subsetneq past(\mathsf{op})$.

Let us prove that $\rightarrow_{lin}$ is real-time compliant. Let op_i and op_j be two operations performed by processes p_i and p_j respectively, and let m_i and m_j be the message sent during the execution of op_i and op_j respectively, on line 1 or 3. Suppose that $\mathsf{op}_i \prec \mathsf{op}_j$ (op_i terminated before op_j started). When p_i returns from op_i, by the waiting condition of line 1 or 3, it has received m_i, but p_j has not yet sent m_j. Therefore, $m_i \mapsto_i m_j$, and consequently $m_j \notin past(\mathsf{op}_i)$. By the waiting condition during the execution of op_j (line 1 or 3), we have $m_j \in past(\mathsf{op}_j)$. By the Containment property of SCD-broadcast, we therefore have $past(\mathsf{op}_i) \subsetneq past(\mathsf{op}_j)$, so $\mathsf{op}_i \rightarrow_{lin} \mathsf{op}_j$. Let $\leq_{lin}$ be a total order extension of $\rightarrow_{lin}^{\star}$. It is real-time compliant because $\rightarrow_{lin}^{\star}$ contains $\prec$.

Let us now consider the value returned by a read() operation op. Let p be the number of PLUS() messages in $past(\mathsf{op})$ and let m be the number of MINUS() messages in $past(\mathsf{op})$. According to line 1, op returns the value of $counter_i$ that is modified only at line 7 and contains the value $p - m$, by commutativity of additions and substractions. Moreover, due to the definition of $\rightarrow_{lin}$, all pairs composed of a read() and an increase() or decrease() operations are ordered by $\rightarrow_{lin}$, and consequently, op has the same increase() and decrease() predecessors according to both $\rightarrow_{lin}$ and to $\leq_{lin}$. Therefore, the value returned by op is the number of times increase() has been called, minus the number of times increase() has been called, before op according to $\leq_{lin}$, which concludes the lemma. $\Box_{Lemma\ 14}$

Theorem 3 *Algorithm* 3 *implements an atomic counter.*

Proof The proof follows from Lemmas 13 and 14. $\Box_{Theorem\ 3}$

An algorithm satisfying sequential consistency The previous algorithm can be easily modified to obtain a sequentially consistent counter. To this end, a technique similar to the one introduced in [8] can be used to allow the operations increase() and decrease() to have a fast implementation. "Fast" means here that these operations are purely local: they do not require the invoking process to wait in the algorithm implementing them. Differently, the operation read() issued by a process p_i cannot be fast, namely, all the previous increase() and decrease() operations issued by p_i must be applied to its local copy of the counter for its invocation of read() terminates (this is the rule known under the name "read your writes").

Algorithm 4 is the resulting algorithm. In addition to $counter_i$, each process manages a local synchronization counter lsc_i initialized to 0, which counts the number of increase() and decrease() executed by p_i and not locally applied to $counter_i$. Only when lsc_i is equal to 0, p_i is allowed to read $counter_i$.

The cost of an operation increase() and decrease() is 0 time units plus the n^2 protocol messages of the underlying SCD-broadcast. The time cost of the operation read() by a process p_i depends on the value of lsc_i. It is 0 when p_i has no "pending" counter operations.

Remark As in [8], using the same technique, it is possible to design a sequentially consistent counter in which the operation read() is fast, while the operations increase() and decrease() are not.

operation increase() **is**
$lsc_i \leftarrow lsc_i + 1$;
scd_broadcast PLUS(i);
return().

operation decrease() **is** the same as increase() where PLUS(i) is replaced by MINUS(i).

operation read() **is**
wait$(lsc_i = 0)$;
return$(counter_i)$.

when the message set { PLUS(j_1), ...,MINUS(j_x), ... } **is scd-delivered do**
let p = number of messages PLUS() in the message set;
let m = number of messages MINUS() in the message set;
$counter_i \leftarrow counter_i + p - m$;
let c = number of messages PLUS(i) and MINUS(i) in the message set;
$lsc_i \leftarrow lsc_i - c$.

Algorithm 4: Construction of a seq. consistent counter in $\mathcal{CAMP}_{n,t}$[SCD-broadcast] (code for p_i)

7 SCD-broadcast in Action (its Power): Lattice Agreement Task

Definition Let S be a partially ordered set, and $\leq$ its partial order relation. Given $S' \subseteq S$, an upper bound of S' is an element x of S such that $\forall\, y \in S' :\; y \leq x$. The *least upper bound* of S' is an upper bound z of S' such that, for all upper bounds y of S', $z \leq y$. S is called a *semilattice* if all its finite subsets have a least upper bound. Let lub(S') denotes the least upper bound of S'.

Let us assume that each process p_i has an input value in_i that is an element of a semilattice S. The *lattice agreement* task was introduced in [6] and generalized in [16]. It provides each process with an operation denoted propose(), such that a process p_i invokes propose(in_i) (we say that p_i proposes in_i); this operation returns an element $z \in S$ (we say that it decides z). The task is defined by the following properties, where it is assumed that each non-faulty process invokes propose().

- Validity. If process p_i decides out_i, we have $in_i \leq out_i \leq$ lub$(\{in_1, \ldots, in_n\})$.
- Containment. If p_i decides out_i and p_j decides out_j, we have $out_i \leq out_j$ or $out_j \leq out_j$.
- Termination. If a non-faulty proposes a value, it decides a value.

Algorithm Algorithm 5 implements the lattice agreement task. It is a very simple algorithm, which uses one instance of the communication pattern introduced in Section 4. The text of the algorithm is self-explanatory.

operation propose(in_i) **is**
$out_i \leftarrow in_i$;
$done_i \leftarrow$ `false`; scd_broadcast MSG(i, in_i); wait$(done_i)$;
return(out_i).

when the message set { MSG(j_1, v_{j_1}), ..., MSG(j_x, v_{j_x})} **is scd-delivered do**
for each MSG$(j, v) \in$ the scd-delivered message set **do** $out_i \leftarrow out_i \cup v$ **end for**;
if $\exists \ell \,:\, j_\ell = i$ **then** $done_i \leftarrow$ `true` **end if**.

Algorithm 5: Solving Lattice Agreement in $\mathcal{CAMP}_{n,t}$[SCD-broadcast] (code for p_i)

Theorem 4 *Algorithm 5 solves the lattice agreement task.*

Proof The Termination property follows from the assumption that all non-faulty processes propose a value, lines 2 and 5. The Validity property follows directly from lines 1 and 4.

As far as the Containment property is concerned we have the following. Let us assume, by contradiction, that there are two processes p_i ans p_j such that we have neither $out_i \leq out_j$ nor $out_j \leq out_j$.This means that there is a value $v \in out_i \setminus out_j$, and a value $v' \in out_j \setminus out_i$. Let ms_i and ms'_i be the message sets (scd-delivered by p_i) which contained v and v' respectively. As $v \in out_i$ and $v' \notin out_i$, we have $ms_i \neq ms'_i$, and ms_i was scd-delivered before ms'_i.

Defining similarly ms_j (containing v') and ms'_j (containing v), we have $ms'_j \neq ms_j$, and ms'_j was scd-delivered before ms_j. It follows (see Section 2) that we have $m \mapsto_i m'$ and $m' \mapsto_j m$, from which it follows that $\mapsto = \cup_{1 \leq x \leq n} \mapsto_x$ is not a partial order. A contradiction with SCD-broadcast definition. $\Box_{Theorem\ 4}$

Remark 1 SCD-broadcast can be built on top of read/write registers (see below Theorem 5). It follows that the combination of Algorithm 5 and Algorithm 6 provides us with a pure read/write algorithm solving the lattice agreement task. As far as we known, this is the first algorithm solving lattice agreement, based only on read/write registers.

Remark 2 Similarly to the algorithms implementing snapshot objects and counters satisfying sequential consistency (instead of linearizability), Algorithm 5 uses no message SYNC().

Let us also notice the following. Objects are specified by "witness" correct executions, which are defined by sequential specifications. According to the time notion associated with these sequences we have two consistency conditions: linearizability (the same "physical" time for all the objects) or sequential consistency (a logical time is associated with each object, independently from the other objects). Differently, as distributed tasks are defined by relations from input vectors to output vectors (i.e., without referring to specific execution patterns or a time notion), the notion of a consistency condition (such as linearizability or sequential consistency) is meaningless for tasks.

8 The Computability Power of SCD-broadcast (its Limits)

This section presents an algorithm building the SCD-broadcast abstraction on top of SWMR snapshot objects. (Such snapshot objects can be easily obtained from MWMR snapshot objects.) Hence, it follows from (a) this algorithm, (b) Algorithm 1, and (c) the impossibility proof to build an atomic register on top of asynchronous message-passing systems where $t \geq n/2$ process may crash [5], that SCD-broadcast cannot be implemented in $\mathcal{CAMP}_{n,t}[t \geq n/2]$, and snapshot objects and SCD-broadcast are computationally equivalent.

8.1 From snapshot to SCD-broadcast

Shared objects The shared memory is composed of two SWMR snapshot objects. Let ϵ denote the empty sequence.

- $SENT[1..n]$: is a snapshot object, initialized to $[\emptyset, \ldots, \emptyset]$, such that $SENT[i]$ contains the messages scd-broadcast by p_i.
- $SETS\_SEQ[1..n]$: is a snapshot object, initialized to $[\epsilon, \ldots, \epsilon]$, such that $SETS\_SEQ[i]$ contains the sequence of the sets of messages scd-delivered by p_i.

The notation $\oplus$ is used for the concatenation of a message set at the end of a sequence of message sets.

Local objects Each process p_i manages the following local objects.

- $sent_i$ is a local copy of the snapshot object $SENT$.
- $sets\_seq_i$ is a local copy of the snapshot object $SETS\_SEQ$.
- $to\_deliver_i$ is an auxiliary variable whose aim is to contain the next message set that p_i has to scd-deliver.

The function members(set_seq) returns the set of all the messages contained in set_seq.

Description of Algorithm 6 When a process p_i invokes scd_broadcast(m), it adds m to $sent_i[i]$ and $SENT[i]$ to inform all the processes on the scd-broadcast of m. It then invokes the internal procedure progress() from which it exits once it has a set containing m (line 1).

A background task T ensures that all messages will be scd-delivered (line 2). This task invokes repeatedly the internal procedure progress(). As, locally, both the application process and the underlying task T can invoke progress(), which accesses the local variables of p_i, those variables are protected by a local fair mutual exclusion algorithm providing the operations enter_mutex() and exit_mutex() (lines 3 and 11).

operation scd_broadcast(m) **is**
(1) $sent_i[i] \leftarrow sent_i[i] \cup \{m\}$; $SENT$.write($sent_i[i]$); progress().

(2) **background task** T **is repeat forever** progress() **end repeat**.

procedure progress() **is**
(3) enter_mutex();
(4) catchup();
(5) $sent_i \leftarrow SENT$.snapshot();
(6) $to\_deliver_i \leftarrow (\cup_{1\leq j\leq n}\ sent_i[j]) \setminus$ members($sets\_seq_i[i]$);
(7) **if** ($to\_deliver_i \neq \emptyset$)
(8) **then** $sets\_seq_i[i] \leftarrow sets\_seq_i[i] \oplus to\_deliver_i$; $SETS\_SEQ$.write($sets\_seq_i[i]$);
(9) scd_deliver($to\_deliver_i$)
(10) **end if**;
(11) exit_mutex().

procedure catchup() **is**
(12) $sets\_seq_i \leftarrow SETS\_SEQ$.snapshot();
(13) **while** ($\exists j, set$: set is the first set in $sets\_seq_i[j]$: $set \not\subseteq$ members($sets\_seq_i[i]$)) **do**
(14) $to\_deliver_i \leftarrow set \setminus$ members($sets\_seq_i[i]$);
(15) $sets\_seq_i[i] \leftarrow sets\_seq_i[i] \oplus to\_deliver_i$; $SETS\_SEQ$.write($sets\_seq_i[i]$);
(16) scd_deliver($to\_deliver_i$)
(17) **end while**.

Algorithm 6: An implementation of SCD-broadcast on top of snapshot objects (code for p_i)

The procedure progress() first invokes the internal procedure catchup(), whose aim is to allow p_i to scd-deliver sets of messages which have been scd-broadcast and not yet locally scd-delivered.

To this end, catchup() works as follows (lines 12-17). Process p_i first obtains a snapshot of $SETS\_SEQ$, and saves it in $sets\_seq_i$ (line 12). This allows p_i to know which message sets have been scd-delivered by all the processes; p_i then enters a "while" loop to scd-deliver as many message sets as possible according to what was scd-delivered by the other processes. For each process p_j that has scd-delivered a message set set containing messages not yet scd-delivered by p_i (predicate of line 13), p_i builds a set $to\_deliver_i$ containing the messages in set that it has not yet scd-delivered (line 14), and locally scd-delivers it (line 16). This local scd-delivery needs to update accordingly both $sets\_seq_i[i]$ (local update) and $SETS\_SEQ[i]$ (global update).

When it returns from catchup(), p_i strives to scd-deliver messages not yet scd-delivered by the other processes. To this end, it first obtains a snapshot of $SENT$, which it stores in $sent_i$ (line 5). If there are messages that can be scd-delivered (computation of $to\_deliver_i$ at line 6, and predicate at line 7), p_i scd-delivers them and updates $sets\_seq_i[i]$ and $SETS\_SEQ[i]$ (lines 7-9) accordingly.

8.2 Proof of Algorithm 6

Lemma 15 *If a process scd-delivers a set containing a message m, some process invoked* scd_broadcast(m).

Proof The proof follows directly from the text of the algorithm, which copies messages from $SENT$ to $SETS\_SEQ$, without creating new messages. $\Box_{Lemma\ 15}$

Lemma 16 *No process scd-delivers the same message twice.*

Proof Let us first observe that, due to lines 7 and 15, all messages that are scd-delivered at a process p_i have been added to $sets\_seq_i[i]$. The proof then follows directly from (a) this observation, (b) the fact that (due to the local mutual exclusion at each process) $sets\_seq_i[i]$ is updated consistently, and (c) lines 6 and 14, which state that a message already scd-delivered (i.e., a message belonging to $sets\_seq_i[i]$) cannot be added to $to\_deliver_i$. $\Box_{Lemma\ 16}$

Lemma 17 *Any invocation of* scd_broadcast() *by a non-faulty process p_i terminates.*

Proof The proof consists in showing that the internal procedure progress() terminates. As the mutex algorithm is assumed to be fair, process p_i cannot block forever at line 3. Hence, p_i invokes the internal procedure catchup(). It then issues first a snapshot invocation on $SETS\_SEQ$ and stores the value it obtains the value of $sets\_seq_i$. There is consequently a finite number of message sets in $sets\_seq_i$. Hence, the "while" of lines 13-17 can be executed only a finite number of times, and it follows that any invocation of catchup() by a non-faulty process terminates. The same reasoning (replacing $SETS\_SEQ$ by $SENT$) shows that process p_i cannot block forever when it executes the lines 5-10 of the procedure progress(). $\Box_{Lemma\ 17}$

Lemma 18 *If a non-faulty process scd-broadcasts a message m, it scd-delivers a message set containing m.*

Proof Let p_i be a non-faulty process that scd-broadcasts a message m. As it is non-faulty, p_i adds m to $SENT[i]$ and then invokes progress() (line 1). As $m \in SENT$, it is eventually added to $to\_deliver_i$ if not yet scd-delivered (line 6), and scd-delivered at line 9, which concludes the proof of the lemma. $\Box_{Lemma\ 18}$

Lemma 19 *If a non-faulty process scd-delivers a message m, every non-faulty process scd-delivers a message set containing m.*

Proof Let us assume that a process scd-delivers a message set containing a message m. It follows that the process that invoked scd_broadcast(m) added m to $SENT$ (otherwise no process could scd-deliver m). Let p_i be a correct process. It invokes progress() infinitely often (line 2). Hence, there is a first execution of progress() such that $sent_i$ contains m (line 5). If then follows from line 6 that m will be added to $to\_deliver_i$ (if not yet scd-delivered). If follows that p_i will scd-deliver a set of messages containing m at line 9. $\Box_{Lemma\ 19}$

Lemma 20 *Let p_i be a process that scd-delivers a set ms_i containing a message m and later scd-delivers a set ms'_i containing a message m'. No process p_j scd-delivers first a set ms'_j containing m' and later a set ms_j containing m.*

Proof Let us consider two messages m and m'. Due to total order property on the operations on the snapshot object $SENT$, it is possible to order the write operations of m and m' into $SENT$. Without loss of generality, let us assume that m is added to $SENT$ before m'. We show that no process scd-delivers m' before m.[4]

Let us consider a process p_i that scd-delivers the message m'. There are two cases.

- p_i scd-delivers the message m' at line 9. Hence, p_i obtained m' from the snapshot object $SENT$ (lines 5-6). As m was written in $SENT$ before m', we conclude that $SENT$ contains m. It then follows from line 6 that, if p_i has not scd-delivered m before (i.e., m is not in $sets\_seq_i[i]$), then p_i scd-delivers it in the same set as m'.
- p_i scd-delivers the message m' at line 16. Due to the predicate used at line 13 to build a set of message to scd-deliver, this means that there is a process p_j that has previously scd-delivered a set of messages containing m'.
  Moreover, let us observe that the first time the message m' is copied from $SENT$ to some $SETS\_SEQ[x]$ occurs at line 8. As m was written in $SENT$ before m', the corresponding process p_x cannot see m' and not m. It follows from the previous item that p_x has scd-delivered m in the same message set (as the one including m'), or in a previous message set. It then follows from the predicate of line 13 that p_i cannot scd-delivers m' before m.

  To summarize, the scd-deliveries of message sets in the procedure catchup() cannot violate the MS-Ordering property, which is established at lines 6-10.

$\square_{Lemma\ 20}$

Theorem 5 *Algorithm* 6 *implements the* SCD-Broadcast *abstraction in the system model* $\mathcal{CARW}_{n,t}[t < n]$.

Proof The proof follows from Lemma 15 (Validity), Lemma 16 (Integrity), Lemmas 17 and 18 (Termination-1), Lemma 19 (Termination-2), and Lemma 20 (MS-Ordering). $\square_{Theorem\ 5}$

9 Conclusion

What was the paper on? This paper has introduced a new communication abstraction, suited to asynchronous message-passing systems where computing entities (processes) may crash. Denoted SCD-broadcast, it allows processes to broadcast messages and deliver *sets of messages* (instead of delivering each message one after the other). More precisely, if a process p_i delivers a set of messages containing a message m, and later delivers a set of messages containing a message m', no process p_j can deliver a set of messages containing m' before a set of messages containing m. Moreover, there is no local constraint imposed on the processing order of the messages belonging to a same message set. SCD-broadcast has the following noteworthy features:

- It can be implemented in asynchronous message passing systems where any minority of processes may crash. Its costs are upper bounded by twice the network latency (from a time point of view) and $(O(n^2)$ (from a message point of view).
- Its computability power is the same as the one of atomic read/write register (anything that can be implemented in asynchronous read/write systems can be implemented with SCD-broadcast).

[4]Let us notice that it is possible that a process scd-delivers them in two different message sets, while another process scd-delivers them in the same set (which does not contradicts the lemma).

- It promotes a communication pattern which is simple to use, when one has to implement concurrent objects defined by a sequential specification or distributed tasks.
- When interested in the implementation of a concurrent object O, a simple weakening of the SCD-broadcast-based atomic implementation of O provides us with an SCD-broadcast-based implementation satisfying sequential consistency (moreover, the sequentially consistent implementation is more efficient than the atomic one).

On programming languages for distributed computing Differently from sequential computing for which there are plenty of high level languages (each with its idiosyncrasies), there is no specific language for distributed computing. Instead, addressing distributed settings is done by the enrichment of sequential computing languages with high level communication abstractions. When considering asynchronous systems with process crash failures, *total order broadcast* is one of them. SCD-broadcast is a candidate to be one of them, when one has to implement read/write solvable objects and distributed tasks.

Acknowledgments

This work has been partially supported by the Franco-German DFG-ANR Project 14-CE35-0010-02 DISCMAT (devoted to connections between mathematics and distributed computing) and the French ANR project 16-CE40-0023-03 DESCARTES (devoted to layered and modular structures in distributed computing).

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] Ahamad M., Neiger G., Burns J.E., Hutto P.W., and Kohli P. Causal memory: definitions, implementation and programming. *Distributed Computing*, 9:37-49 (1995)

[3] Anderson J., Multi-writer composite registers. *Distributed Computing*, 7(4):175-195 (1994)

[4] Aspnes J. and Herlihy M., Wait-free data structures in the asynchronous PRAM model. *Proc. 2nd ACM Symposium on Parallel algorithms and architectures (SPAA'00)*, ACM Press, pp. 340-349 (1990)

[5] Attiya H., Bar-Noy A. and Dolev D., Sharing memory robustly in message passing systems. *Journal of the ACM*, 42(1):121-132 (1995)

[6] Attiya H., Herlihy M., and Rachman O., Atomic snapshots using lattice agreement. *Distributed Computing*, 8:121-132 (1995)

[7] Attiya H. and Rachman O., Atomic snapshots in $O(n \log n)$ operations. *SIAM Journal of Computing*, 27(2):319-340 (1998)

[8] Attiya H. and Welch J.L., Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91-12 (1994)

[9] Attiya H. and Welch J.L., *Distributed computing: fundamentals, simulations and advanced topics*, (2d Edition), Wiley-Interscience, 414 pages (2004)

[10] Biran O., Moran S., and Zaks S., A combinatorial characterization of the distributed tasks which are solvable in the presence of one faulty processor. *Proc. 7th ACM Symposium on Principles of Distributed Computing (PODC'88)*, ACM Press, pp. 263-275 (1988)

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

[12] Chandra T. and Toueg S., Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225-267 (1996)

[13] Chaudhuri S., More choices allow more faults: set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132-158 (1993)

[14] Delporte-Gallet C., Fauconnier H., Rajsbaum S., and Raynal M., Implementing snapshot objects on top of crash-prone asynchronous message-passing systems. *Proc. 16th Int'l Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'16)*, Springer LNCS 10048, pp. 341–355 (2016)

[15] Ellen F., How hard is it to take a snapshot? *Proc. 31th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'05)*, Springer LNCS 3381, pp. 27-35 (2005)

[16] Faleiro J.M., Rajamani S., Rajan K., Ramalingam G., and Vaswani K., Generalized lattice agreement. *Proc. 31th ACM Symposium on Principles of Distributed Computing (PODC'12)*, ACM Press, pp. 125-134 (2012)

[17] Fischer M.J., Lynch N.A. and Paterson M.S., Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374-382 (1985)

[18] Fischer M.J. and Merritt M., Appraising two decades of distributed computing theory research. *Distributed Computing*, 16(2-3):239-247 (2003)

[19] Herlihy M.P. and Shavit N., The Art of Multiprocessor Programming. *Morgan Kaufmann Pub.*, San Francisco (CA), 508 pages (2008)

[20] Herlihy M. P. and Wing J. M., Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492 (1990)

[21] Imbs D., Mostéfaoui A., Perrin M., and Raynal M., Which broadcast abstraction captures k-set agreement? (Extended version). *Tech Report*, ArXiv:1705.04835.pdf, 19 pages (2017)

[22] Imbs D. and Raynal M., Help when needed, but no more: efficient read/write partial snapshot. *Journal of Parallel and Distributed Computing*, 72(1):1-12 (2012)

[23] Inoue I., Chen W., Masuzawa T. and Tokura N., Linear time snapshots using multi-writer multi-reader registers. *Proc. 8th Int'l Workshop on Distributed Algorithms (WDAG'94)*, Springer LNCS 857, pp. 130-140 (1994)

[24] Jayanti P., An optimal multiwriter snapshot algorithm. *Proc. 37th ACM Symposium on Theory of Computing (STOC'05)*, ACM Press, pp. 723-732 (2005)

[25] Lamport L., How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C28(9):690–691 (1979)

[26] Lamport L., On interprocess communication, Part I: basic formalism. *Distributed Computing*, 1(2):77-85 (1986)

[27] Moran S. and Wolfstahl Y., Extended impossibility results for asynchronous complete networks. *Information Processing Letters*, 26(3):145-151 (1987)

[28] Raynal M., *Communication and agreement abstractions for fault-tolerant asynchronous distributed systems.* Morgan & Claypool Publishers, 251 pages, ISBN 978-1-60845-293-4 (2010)

[29] Raynal M., *Concurrent programming: algorithms, principles and foundations*. Springer, 515 pages, ISBN 978-3-642-32026-2 (2013)

[30] Raynal M., Set agreement. *Encyclopedia of Algorithms*, Springer, pp. 1956-1959 (2016)

[31] Raynal M., Schiper A., and Toueg S., The causal ordering abstraction and a simple way to implement it. *Information Processing Letters*, 39:343-351 (1991)

[32] 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)

[33] Shavit N. and Touitou D., Software transactional memory. *Distributed Computing*, 10(2):99-116 (1997)