Another Look at the Implementation of Read/write Registers in Crash-prone Asynchronous Message-Passing Systems (Extended Version)

Damien Imbs, Achour Mostefaoui, Matthieu Perrin, Michel Raynal

To cite this version:

HAL Id: hal-01476201
https://hal.archives-ouvertes.fr/hal-01476201
Submitted on 24 Feb 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Another Look at the Implementation of Read/write Registers in Crash-prone Asynchronous Message-Passing Systems (Extended Version)

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

°LIF, Université Aix-Marseille, 13288 Marseille, France
†LINA, Université de Nantes, 44322 Nantes, France
△Computer science department, Technion, Haifa, 3200003, Israel
⋆Institut Universitaire de France
‡IRISA, Université de Rennes, 35042 Rennes, France

Abstract

“Yet another paper on” the implementation of read/write registers in crash-prone asynchronous message-passing systems! Yes..., but, differently from its predecessors, this paper looks for a communication abstraction which captures the essence of such an implementation in the same sense that total order broadcast can be associated with consensus, or message causal delivery can be associated with causal read/write registers. To this end, the paper introduces a new communication abstraction, named SCD-broadcast (SCD standing for “Set Constrained Delivery”), which, instead of a single message, delivers to processes sets of messages (whose size can be arbitrary), such that the sequences of message sets delivered to any two processes satisfies some constraints. The paper then shows that: (a) SCD-broadcast allows for a very simple implementation of a snapshot object (and consequently also of atomic read/write registers) in crash-prone asynchronous message-passing systems; (b) SCD-broadcast can be built from snapshot objects (hence SCD-broadcast and snapshot objects—or read/write registers—are “computationally equivalent”); (c) SCD-broadcast can be built in message-passing systems where any minority of processes may crash (which is the weakest assumption on the number of possible process crashes needed to implement a read/write register).

Keywords: Asynchronous system, Atomicity, Communication abstraction, Linearizability, Message-passing system, Process crash, Read/write atomic register, Snapshot object.
1 Introduction

The “one-shot” terracotta tablets introduced and used at Sumer about 3030 BC [23], and the “multi-shot” palimpsests used in the middle-age, can be considered as ancestors of the read/write register abstraction. Such an object provides its users with a write operation which defines a new value of the register, and a read operation which returns its value. When considering sequential computing, read/write registers are universal in the sense that they are assumed to allow solving any problem that can be solved [37].

On the variety of read/write registers and their distributed implementation In a shared read/write memory system, the registers are given for free. The situation is different in a message-passing system, where the computing entities (processes) communicate by sending and receiving messages transmitted through a communication network. Hence, in such a distributed context, a register is not given for free, but constitutes a communication abstraction which must be built by a distributed algorithm with the help of the local memories of the processes and the communication network.

Several types of registers have been proposed. They differ according to (a) their size (from binary registers which contain a single bit, to bounded and unbounded registers); (b) their behavior in the presence of concurrency (safe, regular, atomic [25]); (c) the number of processes which are allowed to read them (Single-Reader -SR- vs Multi-Reader -MR- register); and (d) the number of processes which are allowed to write them (Single-Writer -SR- vs Multi-Writer -MR- register), which gives four possible combinations from SWSR to MWMR. There are algorithms building MWMR atomic (bounded and unbounded) registers from SWSR binary safe registers [25] (see [8, 26, 34] for surveys of such algorithms).

As far as a read/write register is concerned, atomicity means that (a) each read or write operation appears as if it had been executed instantaneously at a single point of the time line, (b) this point appears between its start event and its end event, (c) no two operations appear at the same point of the time line, and (d) a read returns the value written by the closest preceding write operation (or the initial value of the register if there is no preceding write) [25, 27]. Linearizability is atomicity extended to any object defined from a sequential specification on total operations [18]. In the following, we consider the terms atomicity and linearizability as synonyms. Hence, a sequence of read and write operations satisfying atomicity is said to be linearizable, and is called a linearization. The point of the time line at which an operation appears to have been executed is called its linearization point.

Many distributed algorithms have been proposed, which build a read/write register on top of a message-passing system, be it failure-free or failure-prone. In the failure-prone case, the addressed failure models are the process crash failure model, and the Byzantine process failure model (see textbooks, e.g., [8, 26, 32, 33]). When considering process crash failures (the one considered in this paper\(^1\)), the most famous of these algorithms was proposed by H. Attiya, A. Bar-Noy, and D. Dolev in [5]. This algorithm, usually called ABD according to the names of its authors, considers an \(n\)-process asynchronous system in which up to \(t < n/2\) processes may crash. As \(t < n/2\) is an upper bound of the number of process crashes which can be tolerated (see [5]), this algorithm is \(t\)-resilient optimal. Its instances implementing SWMR or MWMR atomic read/write registers rely on (a) quorums [38], and (b) a classical broadcast/reply communication pattern. This communication pattern is used twice in a read operation, and once (twice) in a write operation for an SWMR (MWMR) atomic read/write register.

Other algorithms –each with its own properties– implementing atomic read/write registers on top of crash-prone asynchronous message-passing systems can be found in the literature ([4, 12, 17, 29] to cite a few; see also the analytic presentation given in [36]).

From registers to snapshot objects The snapshot object was introduced in [1, 3]. A snapshot object is an array \(\text{REG}[1..m]\) of atomic read/write registers which provides the processes with two operations, denoted \(\text{write}()\) and \(\text{snapshot}()\). If the base registers are SWMR the snapshot is called SWMR snapshot (and we have then \(m = n\)). In this case, the invocation of \(\text{write}(v)\) by a process \(p_i\) assigns \(v\) to \(\text{REG}[i]\), and the invocation of \(\text{snapshot}()\) by a process \(p_i\) returns the value of the full array as if the operation had been executed instantaneously. If the base registers are MWMR, the snapshot is called MWMR snapshot. The invocation of \(\text{write}(r, v)\), where \(1 \leq r \leq m\), by a process \(p_i\) assigns \(v\) to \(\text{REG}[r]\), and \(\text{snapshot}()\) is defined as before. Said another way, the operations \(\text{write}()\) and \(\text{snapshot}()\) are atomic, i.e., in any execution of an SWMR (or MWMR) snapshot object, its operations \(\text{write}()\) and \(\text{snapshot}()\) are linearizable.

Implementations of both SWMR and MWMR snapshot objects on top of read/write atomic registers have

---

\(^1\)For Byzantine failures, see for example [28].
been proposed (e.g., [1, 3, 19, 20]). The “hardness” to build snapshot objects in read/write systems and associated lower bounds are presented in the survey [13]. The best algorithm known to implement an SWMR snapshot requires $O(n \log n)$ read/write on the base SWMR registers for both the write() and snapshot() operations [6]. As far as MWMR snapshot objects are concerned, there are implementations where each operation has an $O(n)$ cost.

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 (such as ABD), 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 communication pattern as ABD, which builds an atomic SWMR snapshot object “directly” (i.e., without stacking algorithms) was recently presented in [11] (the aim of this algorithm is to perform better that the stacking approach in concurrency-free executions).

**Another look at the implementation of read/write registers and snapshot objects** In sequential computing, there are “natural” pairings linking data structures and control structures. The most simple examples are the pair “array and for loop”, and the pair “tree and recursion”.

When we look at the implementation of a causal read/write register [2] on top of a (crash-free or crash-prone) message-passing system, the causal message delivery broadcast abstraction [9, 35] is the appropriate communication abstraction. Namely, given this abstraction for free, the algorithms implementing the read and write operations build on top of it, become very simple, need only a few lines, and are easy to understand and to prove correct. Of course, this is due to the fact that the causal broadcast abstraction captures and abstracts the causality relation needed to implement a causal read/write register. Similarly, total order broadcast is the communication abstraction associated with the consensus object [10]. This is summarized in Table 1.

<table>
<thead>
<tr>
<th>Concurrent object</th>
<th>Communication abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Causal read/write registers</td>
<td>Causal message delivery [9, 35]</td>
</tr>
<tr>
<td>Consensus</td>
<td>Total order broadcast [10]</td>
</tr>
<tr>
<td>Snapshot object (and R/W register)</td>
<td>SCD-broadcast (This paper)</td>
</tr>
</tbody>
</table>

Table 1: Associating objects and communication abstractions in a wait-free model

As already said, all the algorithms we know which implement atomic read/write registers, and (by stacking transitivity or directly) SWMR or MWMR snapshots objects, on top of crash-prone asynchronous message-passing systems, are based on a broadcast/reply pattern plus the use of intersecting quorums. Hence, the following question naturally arises: Is this approach the “only” way to implement a snapshot object (or an atomic register), or is there a specific communication abstraction which captures the essence and simplifies the implementation of snapshot objects (and atomic read/write registers)?

**Content of the paper** Informatics in general (and distributed computing in particular) is a science of abstractions, and this paper is distributed programming abstraction-oriented. It strives to address 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 [16]. More precisely, it answers the previous question in a positive way. To this end, it presents a simple broadcast abstraction which matches –and therefore captures the essence of– snapshot objects (and atomic read/write registers). We call it Set-Constrained Delivery Broadcast (in short SCD-broadcast). Given this communication abstraction, it is possible to quorum-free build snapshot objects, and vice versa. Hence, similarly to consensus and total order broadcast, SCD-broadcast and snapshot objects have the same computational power (Table 1).

The SCD-broadcast communication abstraction allows a process to broadcast messages, and to deliver sets of messages (instead of single messages) 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.

\footnote{Snapshot objects built in read/write models enriched with operations such as Compare&Swap, or LL/SC, have also been considered, e.g., [21, 19]. Here we are interested in pure read/write models.}

\footnote{In the rest of the paper, the identifiers starting with “ms” denote message sets.}
The implementation of an instance of SCD-broadcast costs $O(n^2)$ messages. It follows that the cost of a snapshot operation (or a read/write register operation) on top of a message-passing asynchronous system, where any minority of processes may crash, is also $O(n^2)$ for both SWMR and MWMR snapshot objects (i.e., better than the stacking approach for SWMR snapshot objects). Additionally, be the snapshot objects that are built SWMR or MWMR, their implementation differ only in the fact that their underling read/write registers are SWMR or MWMR. This provides us with a noteworthy genericity-related design simplicity.

Of course, there is rarely something for free. The algorithms implementing the snapshot and write operations are simple because the SCD-broadcast abstraction hides enough “implementation details” and provides consequently a high level abstraction (much higher than the simple broadcast used in ABD-like algorithms). Its main interest lies in its capture of the high level message communication abstraction that, despite asynchrony and process failures, allows simple message-passing implementations of shared memory objects such as snapshot objects and atomic read/write registers.

Roadmap The paper is composed of 7 sections. Section 2 presents the two base computation models concerned in this paper, (read/write and message-passing). Section 3 presents the SCD-broadcast communication abstraction. Then, Section 4 presents a simple algorithm which implements a snapshot object on top of an asynchronous system enriched with SCD-broadcast, in which any number of processes may crash. Section 6 addresses the other direction, namely, it presents an algorithm building the SCD-broadcast abstraction on top of an asynchronous system enriched with snapshot objects and where any number of processes may crash. Section 7 concludes the paper. A noteworthy feature of the algorithms that are presented lies in their simplicity, which is a first class property.

Appendix A describes an implementation of SCD-broadcast suited to asynchronous message-passing systems where any minority of processes may crash. Hence, being implementable in the weakest message-passing system model in which a read/write register can be built, SCD-broadcast is not “yet another oracle” which makes things simpler to understand but cannot be implemented. Appendix B presents simplified SCD-based algorithms which build atomic and sequentially consistent read/write registers.

2 Basic Computation Models

This section presents two basic computation models. In both cases, the process model is the same.

2.1 Processes

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 its possible crash. The model parameter $t$ denotes the maximal number of processes that may crash in a run. A process that crashes in a run is said to be faulty. Otherwise, it is non-faulty. Hence a faulty process behaves as a non-faulty process until it crashes.

2.2 Basic crash-prone asynchronous shared memory model

Atomic read/write register The notion of an atomic read/write register has been formalized in [25, 27]. An MWMR atomic register (say REG) is a concurrent object which provides each process with an operation denoted REG.write(), and an operation denoted REG.read(). When a process invokes REG.write(v) it defines v as being the new value of REG. An MWMR atomic register is defined by the following set of properties.

- Liveness. An invocation of an operation by a non-faulty process terminates.

- Consistency (safety). All the operations invoked by the processes, except possibly –for each faulty process– the last operation it invoked, appear as if they have been executed sequentially and this sequence of operations is such that:

  - each read returns the value written by the closest write that precedes it (or the initial value of REG if there is no preceding write).

4 From the point of view of the maximal number of process crashes that can be tolerated, assuming failures are independent.
Similarly, when it invokes scd classical terminology, when a process invokes scd parameter. The second one returns a non-empty set of messages to the process that invoked it. Using a clas-
messages in CAMP A Notation and necessary and sufficient condition or is only very slow.

This set of properties states that, from an external observer point of view, the read/write register appears as if it is accessed sequentially by the processes, and this sequence (a) respects the real-time access order, and (b) belongs to the sequential specification of a register.

**Notation** The previous computation model is denoted \( C.A.R.W_{n,t}[\emptyset] \) (Crash Asynchronous Read-Write). This basic read/write model is also called wait-free read/write model. The symbol \( \emptyset \) means there is no specific constraint on \( t \), which is equivalent to \( t < n \), as it is always assumed that not all processes crash.

**Snapshot object** This object was defined in the introduction. As we have seen, snapshot objects can be built in \( C.A.R.W_{n,t}[\emptyset] \). As we have seen there are two types of snapshot objects. SWMR snapshot objects (whose base registers are SWMR), and MWMR snapshot objects (whose base registers are MWMR). In the following we consider MWMR snapshot objects, but the algorithms can be trivially adapted to work with SWMR snapshot objects.

\( C.A.R.W_{n,t}[\emptyset] \) enriched with snapshot objects is denoted \( C.A.R.W_{n,t}[\text{snapshot}] \). As a snapshot object can be built in \( C.A.R.W_{n,t}[\emptyset] \) this model has the same computational power as \( C.A.R.W_{n,t}[\emptyset] \). It only offers a higher abstraction level.

### 2.3 Basic crash-prone asynchronous message-passing model

**Communication** Each pair of processes communicate by sending and receiving messages through two unidirectional 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.

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 synchrony, no process can know if another process crashed or is only very slow.

**Notation and necessary and sufficient condition** This computation model is denoted \( C.A.M.P_{n,t}[\emptyset] \) (Crash Asynchronous Message-Passing).

The constraint \( t < n/2 \) is a necessary and sufficient condition to implement an atomic read/write register in \( C.A.M.P_{n,t}[\emptyset] \) [5]. Hence, the model \( C.A.M.P_{n,t}[\emptyset] \) whose runs are constrained by \( t < n/2 \) is denoted \( C.A.M.P_{n,t}[t < n/2] \).

### 3 A Broadcast Abstraction: Set-Constrained Message Delivery

**Definition** The set-constrained broadcast abstraction (SCD-broadcast) provides the processes with two operations, denoted scd\_broadcast() and 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 scd\_broadcast\( (m) \), we say that it “scd-broadcasts a message \( m \)”. Similarly, when it invokes scd\_deliver\( () \) and obtains a set of messages \( ms \), we say that it “scd-delivers a set of messages \( ms \)”. By a slight abuse of language, we also say that a process “scd-delivers a message \( m \)” when it delivers a message \( m \in ms \).

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.** If a process \( p_i \) scd-delivers first a message \( m \) belonging to a set \( ms_i \) and later a message \( m' \) belonging to a set \( ms_i' \neq ms_i \), then no process scd-delivers first the message \( m' \) in some scd-delivered set \( ms_j' \) and later the message \( m \) in some scd-delivered set \( ms_j \neq ms_j' \).
• 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 non-faulty 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). 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.

An example Let \( m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_8, \ldots \) 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 \( p_1 \): \( \{m_1, m_2\}, \{m_3, m_4, m_5\}, \ldots \)
- at \( p_2 \): \( \{m_1, m_3\}, \{m_2\}, \ldots \)

A containment property Let \( ms^\ell_i \) be the \( \ell \)-th message set scd-delivered by \( p_i \). Hence, at some time, \( p_i \) scd-delivered the sequence of message sets \( ms^1_i, \ldots, ms^\phi_i \). Let \( MS^\ell_i = ms^1_i \cup \cdots \cup ms^\phi_i \). The following property follows directly from the MS-Ordering and Termination-2 properties:

- Containment. \( \forall i, j, x, y: (MS^\ell_i \subseteq MS^\ell_j) \lor (MS^\phi_j \subseteq MS^\ell_i) \).

Remark 1: Weakening SCD-broadcast If the messages in a message set are delivered one at a time, and the MS-Ordering property is suppressed, SCD-broadcast boils down to Reliable Broadcast.

Remark 2: On the partial order 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 \( \rightarrow_i \) be the local message delivery order at a process \( p_i \) defined as follows: \( m \rightarrow_i m' \) if \( p_i \) scd-delivers the set containing \( m \) before the set containing \( m' \). As no message is scd-delivered twice, it is easy to see that \( \rightarrow_i \) is a partial order (locally known 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 \( \cup_{1 \leq i \leq n} \rightarrow_i \). This is where SCD-broadcast can be seen as a weakening of total order broadcast.

## 4 From SCD-broadcast to an MWMR Snapshot Object

Let \( \mathcal{CAMP}_{n,t}[\text{SCD-broadcast}] \) denote \( \mathcal{CAMP}_{n,t}[0] \) enriched with the SCD-broadcast abstraction. Hence, this abstraction is given for free. This section presents and proves correct a simple algorithm building an MWMR snapshot object on top of \( \mathcal{CAMP}_{n,t}[\text{SCD-broadcast}] \). The same algorithm with very few simple modifications can be used to build SWMR or MWMR atomic registers in \( \mathcal{CAMP}_{n,t}[\text{SCD-broadcast}] \) (see Appendix B).
4.1 Building an MWMR snapshot object on top of $CAMP_{\text{SCD-broadcast}}$

Let $REG[1..m]$ denote the MWMR snapshot object that is built.

**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 $(0, -)$. The fields associated with $tsa_i[r]$ are denoted $(tsa_i[r].date, tsa_i[r].proc)$.

**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 $ts1 <_ts ts2 \iff (h1 < h2) \lor ((h1 = h2) \land (i1 < i2))$.

**Algorithm 1: snapshot operation** (Lines 1-4) When $p_i$ invokes $REG.snapshot()$, it first sets $done_i$ to false, and invokes $scd.broadcast \ SYNCH(i)$. $SYNCH$ is a synchronization message, whose aim is to entail the refreshment of the value of $reg_i[1..m]$ (lines 11-17) which occurs before the setting of $done_i$ to true (line 18). When this happens, $p_i$ returns the value of its local variable $reg_i[1..m]$ and terminates its snapshot invocation.

```plaintext
operation snapshot() is
(1) done_i ← false;
(2) scd.broadcast SYNCH(i);
(3) wait(done_i);
(4) return(reg_i[1..m]).

operation write(r, v) is
(5) done_i ← false;
(6) scd.broadcast SYNCH(i);
(7) wait(done_i);
(8) done_i ← false;
(9) scd.broadcast WRITE(r, v, (tsa_i[r].date + 1, i));
(10) wait(done_i).

when the message set \{ WRITE(r_{j_1}, v_{j_1}, \langle date_{j_1}, j_1 \rangle), \ldots, WRITE(r_{j_x}, v_{j_x}, \langle date_{j_x}, j_x \rangle),
SYNCH(j_{x+1}), \ldots, SYNCH(j_y) \} is scd-delivered do
(11) for each $r$ such that WRITE($r, -, -$) ∈ scd-delivered message set do
(12) (date, writer) be the greatest timestamp in the messages WRITE($r, -, -$);
(13) if $tsa_i[r] <_{ts} \langle date, writer \rangle$
(14) then let $v$ be written in WRITE($r, -, \langle date, writer \rangle$);
(15) $reg_i[r] ← v$; tsa_i[r] ← \langle date, writer \rangle
(16) end if
(17) end for;
(18) if $\exists j : j_i = i$ then done_i ← true end if.
```

Algorithm 1: Construction of an MWMR snapshot object $CAMP_{\text{SCD-broadcast}}$ (code for $p_i$)

**Algorithm 1: write operation** (Lines 5-10) When a process $p_i$ wants to assign a value $v$ to $REG[r]$, it invokes $REG.write(r, v)$. This operation is made up of two parts. First, $p_i$ executes a re-synchronization (lines 5-7, exactly 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 part, $p_i$ associates the timestamp $\langle tsa_i[r].date + 1, i \rangle$ with $v$, and invokes $scd.broadcast \ WRITE(r, v, \langle tsa_i[r].date + 1, i \rangle)$ (line 9). In addition to informing the other processes on its write of $REG[r]$, this message $WRITE()$ acts as a re-synchronization message, exactly as a message $SYNCH(i)$. When this synchronization terminates (i.e., when the Boolean $done_i$ is set to true), $p_i$ returns from the write operation (line 10).

**Algorithm 1: scd-delivery of a set of messages** When $p_i$ scd-delivers a message set, namely,
\{ WRITE($r_{j_1}, v_{j_1}, \langle date_{j_1}, j_1 \rangle$), \ldots, WRITE($r_{j_x}, v_{j_x}, \langle date_{j_x}, j_x \rangle$), SYNCH($j_{x+1}$), \ldots, SYNCH($j_y$) \}
it first looks if there are messages WRITE(). If it is the case, for each register $REG[r]$ for which there are
messages $\text{WRITE}(r,-,-)$ (line 11), $p_i$ computes the maximal timestamp carried by these messages (line 12), and updates accordingly its local representation of $\text{REG}[r]$ (lines 13-15). Finally, if $p_i$ is the sender of one of these messages ($\text{WRITE}()$ or $\text{SYNC}()$), $\text{done}_i$ is set to $\text{true}$, which terminates $p_i$’s re-synchronization (line 18).

**Message cost** An invocation of snapshot() involves one invocation of scd_broadcast(), and an invocation of write() involves two such invocations. It is shown in Appendix A that, in a message-passing system, scd_broadcast() costs $O(n^2)$ protocol messages. It follows that, in such systems, the message cost of both operations of a snapshot object is $O(n^2)$. (This remains true for SWMR snapshot objects, see Appendix B.)

### 4.2 Proof of Algorithm 1

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-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 9 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 $\text{SYNC}()$ scd-broadcast by a process $p_i$ does not modify the local variables of the other processes.

### 5 Proof of Lemmas for Theorem 1

**Lemma 1** 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 $\text{SYNC}()$ or WRITE() it sends at line 2, 6 or 9. As all the statements associated with the scd-delivery of a message set (lines 11-18) terminate, it follows that the synchronization Boolean $\text{done}_i$ is eventually set to $\text{true}$. Consequently, $p_i$ returns from the invocation of its operation.

**Extension of the relation $<_{ts}$** The relation $<_{ts}$ is extended to a partial order on arrays of timestamps, denoted $\leq_{tsa}$, defined as follows: $tsa_1[1..m] \leq_{tsa} tsa_2[1..m]$ \text{def}= \forall r : (tsa_1[r] = tsa_2[r] \lor tsa_1[r] <_{ts} tsa_2[r]).$

Moreover, $tsa_1[1..m] <_{tsa} tsa_2[1..m]$ \text{def}= (tsa_1[1..m] \leq_{tsa} tsa_2[1..m]) \land (tsa_1[1..m] \neq tsa_2[1..m]).$

**Definition** Let $TSA_i$ be the set of the array values taken by $tsa_i[1..m]$ at line 18 (end of the processing of the message set by process $p_i$). Let $TSA = \cup_{1 \leq i \leq n} TSA_i$.

**Lemma 2** 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 $tsa_i[1..m]$ whose entries can only increase, lines 13 and 15). Hence, let $tsa_1[1..m]$ be an array value of $TSA_i$, and $tsa_2[1..m]$ an array value of $TSA_j$, where $i \neq j$.

Let us assume, by contradiction, that $\lnot (tsa_1 \leq_{tsa} tsa_2)$ and $\lnot (tsa_2 \leq_{tsa} tsa_1)$. As $\lnot (tsa_1 \leq_{tsa} tsa_2)$, there is a registers $r$ such that $tsa_2[r] < tsa_1[r]$. According to lines 13 and 15, there is a message WRITE($r,-,tsa_1[r]$) received by $p_i$ when $tsa_i = tsa_1$ and not received by $p_j$ when $tsa_j = tsa_2$ (because $tsa_2[r] < tsa_1[r]$). Similarly, there is a message WRITE($r',-,tsa_2[r']$) received by $p_j$ when $tsa_j = tsa_2$ and not received by $p_i$ when $tsa_i = tsa_1$. This situation contradicts the MS-Ordering property, from which we conclude that either $tsa_1 \leq_{tsa} tsa_2$ or $tsa_2 \leq_{tsa} tsa_1$.

**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 9, i.e., $(tsa_i[r].date + 1, i)$.

Let $op_1$ and $op_2$ be any two operations. The relation $\prec$ on the whole set of operations is defined as follows: $op_1 \prec op_2$ if $op_1$ terminated before $op_2$ started. It is easy to see that $\prec$ is a real-time-compliant partial order on all the operations.

**Lemma 3** No two distinct write operations on the same register write1($r,v$) and 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 \( \langle \text{date1}, i \rangle \) and \( \langle \text{date2}, j \rangle \) be the timestamp of write1\((r, v)\) and write2\((r, w)\), respectively. If \( i \neq j \), write1\((r, v)\) and 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 write1\((r, v)\) first. As write1\((r, v)\) precedes write2\((r, w)\), \( p_i \) first invoked scd_broadcast write\((r, v, \langle \text{date1}, i \rangle)\) (line 9) and later write\((r, w, \langle \text{date2}, i \rangle)\). It follows that these SCD-broadcast invocations are separated by a local reset of the Boolean \( \text{done}_j \) at line 16. Moreover, before the reset of \( \text{done}_j \) due to the scd-delivery of the message \( \{ \cdots \text{write}(r, v, \langle \text{date1}, i \rangle), \cdots \} \), we have \( \text{tsa}_j[r]. \text{date} \geq \text{date1} \) (lines 12-16). Hence, we have \( \text{tsa}_j[r]. \text{date} \geq \text{date1} \) before the reset of \( \text{done}_j \) (line 18). Then, due to the “+1” at line 9, write\((r, w, \langle \text{date2}, i \rangle)\) is such that \( \text{date2} > \text{date1} \), which concludes the proof of the first part of the lemma.

Let us now consider that write1\((r, v) \prec \text{write2}(r, w)\). If write1\((r, v)\) and write2\((r, w)\) have been produced by the same process we have \( \text{date1} < \text{date2} \) from the previous reasoning. So let us assume that they have been produced by different processes \( p_i \) and \( p_j \). Before terminating write1\((r, v)\) (when the Boolean \( \text{done}_i \) is set true at line 18), \( p_i \) received a message set \( \text{ms1} \) containing the message write\((r, v, \langle \text{date1}, i \rangle)\). When \( p_j \) executes write2\((r, w)\), it first invokes scd_broadcast \( \text{SYNC}(j) \) at line 6. Because write1\((r, v)\) terminated before write2\((r, w)\) started, this message \( \text{SYNC}(j) \) cannot belong to \( \text{ms1} \).

Due to Integrity and Termination-2 of SCD-broadcast, \( p_j \) eventually scd-delivers exactly one message set \( \text{ms1}_j \) containing write\((r, v, \langle \text{date1}, i \rangle)\). Moreover, it also scd-delivers exactly one message set \( \text{ms2}_j \) containing its own message \( \text{SYNC}(j) \). On the other hand, \( p_i \) scd-delivers exactly one message set \( \text{ms2}_i \) containing the message \( \text{SYNC}(j) \). It follows from the MS-Ordering property that, if \( \text{ms2}_j \neq \text{ms1}_j \), \( p_j \) cannot scd-deliver \( \text{ms2}_j \) before \( \text{ms1}_j \). Then, whatever the case (\( \text{ms1}_i = \text{ms2}_j \) or \( \text{ms1}_i \) is scd-delivered at \( p_j \) before \( \text{ms2}_j \)), it follows from the fact that the messages write\()\ are processed (lines 11-17) before the messages \( \text{SYNC}(j) \) (line 18), that we have \( \text{tsa}_j[r] \geq \langle \text{date1}, i \rangle \) when \( \text{done}_j \) is set to \( \text{true} \). It then follows from line 9 that \( \text{date2} > \text{date1} \), which concludes the proof of the lemma. \( \square \)

Lemma 3

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

- Case \( \text{op}() = \text{snapshot}() \). Let \( p_i \) be the invoking process; \( \text{tsa}(\text{op}) \) is the value of \( \text{tsa}_i[1..m] \) when \( p_i \) returns from the snapshot operation (line 4).
- Case \( \text{op}() = \text{write}(r, v) \). Let \( \min_{\text{tsa}}(\{ A \}) \), where \( A \) is a set of array values, denote the smallest array value of \( A \) according to \( \prec_{\text{tsa}} \). Let \( \text{tsa}(\text{op}) \overset{\text{def}}{=} \min_{\text{tsa}}(\{ \text{tsa}[1..m] \in \text{TSA} \text{ such that } ts(\text{op}) \leq ts(\text{tsa}[r]) \}) \).

Hence, \( \text{tsa}(\text{op}) \) is the first \( \text{tsa}[1..m] \) of \( \text{TSA} \), that reports the operation \( \text{op}() = \text{write}(r, v) \).

Lemma 4 Let \( \text{op} \) and \( \text{op}' \) be two distinct operations such that \( \text{op} \prec \text{op}' \). We have \( \text{tsa}(\text{op}) \leq_{\text{tsa}} \text{tsa}(\text{op}') \). Moreover, if \( \text{op}' \) is a write operation, we have \( \text{tsa}(\text{op}) <_{\text{tsa}} \text{tsa}(\text{op}') \).

Proof Let \( p_i \) and \( p_j \) be the processes that performed op and op', respectively. Let \( \text{SYNC}_j \) be the \( \text{SYNC}(j) \) message sent by \( p_j \) (at line 2 or 6) during the execution of op'. Let \( \text{term}_{\text{tsa}_i} \) be the value of \( \text{tsa}_i[1..m] \) when op terminates (line 4 or 10), and \( \text{sync}_{\text{tsa}_j} \) the value of \( \text{tsa}_j[1..m] \) when \( \text{done}_j \) becomes true for the first time after \( p_j \) sent \( \text{SYNC}_j \) (line 3 or 7). Let us notice that \( \text{term}_{\text{tsa}_i} \) and \( \text{sync}_{\text{tsa}_j} \) are elements of the set \( \text{TSA} \).

According to lines 13 and 15, for all \( r, \text{tsa}_i[r] \) is the largest timestamp carried by a message \( \text{write}(r, v, \text{ms}) \) received by \( p_i \) in a message set before op terminates. Let \( m \) be a message such that there is a set \( \text{ms} \) scd-delivered by \( p_i \) before it terminated op. As \( p_j \) sent \( \text{SYNC}_j \) after \( p_i \) terminated, \( p_i \) did not receive any set containing \( \text{SYNC}_j \) before it terminated op. By the properties Termination-2 and MS-Ordering, \( p_j \) received message \( m \) in the same set as \( \text{SYNC}_j \) or in a message set \( \text{ms}' \) received before the set containing \( \text{SYNC}_j \). Therefore, we have \( \text{term}_{\text{tsa}_i} \leq_{\text{tsa}} \text{sync}_{\text{tsa}_j} \).

If \( \text{op} \) is a snapshot operation, then \( \text{tsa}(\text{op}) = \text{term}_{\text{tsa}_i} \). Otherwise, \( \text{op}() = \text{write}(r, v) \). As \( p_i \) has to wait until it processes a set of messages including its \( \text{write}() \) message (and executes line 18), we have \( \text{ts}(\text{op}) <_{\text{tsa}} \text{term}_{\text{tsa}_i} \). Finally, due to the fact that \( \text{term}_{\text{tsa}_i} \in \text{TSA} \) and Lemma 2, we have \( \text{tsa}(\text{op}) \leq_{\text{tsa}} \text{term}_{\text{tsa}_i} \).

If \( \text{op}' \) is a snapshot operation, then \( \text{sync}_{\text{tsa}_j} = \text{tsa}(\text{op}') \) (line 4). Otherwise, \( \text{op}() = \text{write}(r, v) \) and thanks to the “+1” in line 9, \( \text{sync}_{\text{tsa}_j}[r] \) is strictly smaller than \( \text{tsa}(\text{op}')[r] \) which, due to Lemma 2, implies \( \text{sync}_{\text{tsa}_j} <_{\text{tsa}} \text{tsa}(\text{op}') \).
It follows that, in all cases, we have \( tsa(op) \leq tsa ~ term \_ tsa \_ i \leq tsa ~ sync \_ tsa \_ j \leq tsa ~ tsa(op') \) and if \( op' \) is a write operation, we have \( tsa(op) \leq tsa ~ term \_ tsa \_ i \leq tsa ~ sync \_ tsa \_ j < tsa ~ tsa(op') \), which concludes the proof of the lemma. \( \square \)

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 \( op, op' \) be two operations. We define the \( \rightarrow_{lin} \) relation by \( op \rightarrow_{lin} op' \) if one of the following properties holds:

- \( op < op' \),
- \( tsa(op) < tsa ~ tsa(op') \),
- \( tsa(op) = tsa(op') \), \( op \) is a write operation and \( op' \) is a snapshot operation,
- \( tsa(op) = tsa(op') \), \( op \) and \( op' \) are two write operations on the same register and \( ts(op) < ts(op') \).

**Lemma 5** The snapshot object built by Algorithm 1 is linearizable.

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

- \( op < op' \),
- \( tsa(op) < tsa ~ tsa(op') \),
- \( tsa(op) = tsa(op') \), \( op \) is a write operation and \( op' \) is a snapshot operation,
- \( tsa(op) = tsa(op') \), \( op \) and \( op' \) are two write operations on the same register and \( ts(op) < ts(op') \).

We define the \( \rightarrow_{lin} \) relation as the reflexive and transitive closure of the \( \rightarrow_{lin} \) relation.

Let us prove that the \( \rightarrow_{lin}^* \) relation is a partial order on all operations. Transitivity and reflexivity are given by construction. Let us prove antisymmetry. Suppose there are \( op_0, op_1, ..., op_m \) such that \( op_0 = op_m \) and \( op_i \rightarrow_{lin} op_{i+1} \) for all \( i < m \). By Lemma 4, for all \( i < m \), we have \( tsa(op_i) \leq tsa(head_{op_{i+1}}) \), and \( tsa(op_m) = tsa(op_0) \), so the timestamp array of all operations are the same. Moreover, if \( op_0 \) is a snapshot operation, then \( op_1 < op_{(i+1)m} \) is the only possible case (% stands for “modulo”), and by Lemma 4 again, \( op_{(i+1)m} \) is a snapshot operation. Therefore, only two cases are possible.

- Let us suppose that all the \( op_i \) are snapshot operations and for all \( i, op_i < op_{(i+1)m} \). As \( < \) 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 4, for all \( i \neq op_{(i+1)m} \). The operations \( op_i \) and \( op_{(i+1)m} \) are ordered by the fourth point, so they are write operations on the same register and \( ts(op_i) < ts(op_{(i+1)m}) \). By antisymmetry of the \( <_{ts} \) relation, all the \( op_i \) have the same timestamp, so by Lemma 3, they are the same operation, which proves antisymmetry.

Let \( \leq_{lin} \) be a total order extension of \( \rightarrow_{lin}^* \). Relation \( \leq_{lin} \) is real-time compliant because \( \rightarrow_{lin}^* \) contains \( < \).

Let us consider a snapshot operation \( op \) and a register \( r \) such that \( tsa(op)[r] = \langle date1, i \rangle \). According to line 10, it is associated to the value \( v \) that is returned by \( read() \) for \( r \), and comes from a \( WRITE(r, v, \langle date1, i \rangle) \) message sent by a write operation \( op_r = write(r, v) \). By definition of \( tsa(op_r) \), we have \( tsa(op_r) \leq tsa(tsa(op)) \) (Lemma 4), and therefore \( op_r \leq_{lin} op \). Moreover, for any different write operation \( op'_r \) on \( r \), by Lemma 3, \( ts(op'_r) \neq ts(op_r) \). If \( ts(op'_r) < ts(op_r) \), then \( op'_r \leq_{lin} op_r \). Otherwise, \( tsa(op) <_{ts} tsa(op'_r) \), and (due to the first item of the definition of \( \rightarrow_{lin}^* \)) we have \( op \leq_{lin} op'_r \). In both cases, the value written by \( op_r \) is the last value written on \( r \) before \( op \), according to \( \leq_{lin} \).

**Theorem 1** Algorithm 1 builds an MWMR snapshot object in the system model \( \mathcal{CAMP}_n \mid \mathcal{SCD}-broadcast \).

**Proof** The proof follows from Lemmas 1-5. \( \square \)
6 From SWMR Snapshot to SCD-broadcast

This section presents an algorithm which builds the SCD-broadcast abstraction in C\textsc{ARW}_{n,t}[snapshot]. This algorithm completes the computational equivalence of snapshot and SCD-broadcast. (SWMR snapshot objects can be easily implemented in C\textsc{AMP}_{n,t}[SCD-broadcast] by instantiating Algorithm 1 with \(m = n\), and only allowing \(p_i\) to invoke \textit{REG}.\textit{write}(r, -).)

6.1 Algorithm 2

**Shared objects** The shared memory is composed of two SWMR snapshot objects (as defined above). Let \(\epsilon\) denote the empty sequence.

- \(\text{SENT}[1..n]\): is a snapshot object, initialized to \([\emptyset, \cdots, \emptyset]\), such that \(\text{SENT}[i]\) contains the messages scd-broadcast by \(p_i\).
- \(\text{SETS}_\text{-SEQ}[1..n]\): is a snapshot object, initialized to \([\epsilon, \cdots, \epsilon]\), such that \(\text{SETS}_\text{-SEQ}[i]\) 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.

- \(\text{sent}_i\) is a local copy of the snapshot object \(\text{SENT}\).
- \(\text{sets}_\text{-seq}_i\) is a local copy of the snapshot object \(\text{SETS}_\text{-SEQ}\).
- \(\text{to}_\text{-deliver}_i\) is an auxiliary variable whose aim is to contain the next message set that \(p_i\) has to scd-deliver.

The function \textit{members}([\text{set}_\text{-seq}]) returns the set of all the messages contained in \text{set}_\text{-seq}.

**Description of Algorithm 2** When a process \(p_i\) invokes \textit{scd\_broadcast}(\(m\)), it adds \(m\) to \(\text{sent}_i[i]\) and \(\text{SENT}[i]\) to inform all the processes on the scd-broadcast of \(m\). It then invokes the internal procedure \textit{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 \textit{progress}(). As, locally, both the application process and the underlying task \(T\) can invoke \textit{progress}(), which accesses the local variables of \(p_i\), those variables are protected by a local fair mutual exclusion algorithm providing the operations \textit{enter\_mutex}() and \textit{exit\_mutex}() (lines 3 and 11).

```
operation \textit{scd\_broadcast}(m) is
(1) \text{sent}_i[i] \leftarrow \text{sent}_i[i] \cup \{m\}; \text{SENT}\text{.write}(\text{sent}_i[i]); \text{progress}()
(2) \text{background task } T \text{ is repeat forever } \text{progress()} \text{ end repeat}.

procedure \textit{progress}() is
(3) \text{enter\_mutex}()
(4) \text{catch\_up}()
(5) \text{sent}_i \leftarrow \text{SENT}\text{.snapshot}()
(6) \text{to\_deliver}_i \leftarrow \{\cup_{1 \leq j \leq n} \text{sent}_i[i] \} \setminus \text{members}(\text{sets}_\text{-seq}_i[i])
(7) \text{if } (\text{to\_deliver}_i \neq \emptyset) \text{ then } \text{sets}_\text{-seq}_i[i] \leftarrow \text{sets}_\text{-seq}_i[i] \oplus \text{to\_deliver}_i;
(8) \text{SETS}_\text{-SEQ}[i] \leftarrow \text{sets}_\text{-seq}_i[i];
(9) \text{scd\_deliver}(\text{to\_deliver}_i)
(10) \text{end if};
(11) \text{exit\_mutex}()

procedure \textit{catch\_up}() is
(12) \text{sets}_\text{-seq}_i \leftarrow \text{SETS}_\text{-SEQ}\text{.snapshot}()
(13) \text{while } (\exists j : \text{ set } is the first set in \text{sets}_\text{-seq}_j[j] : \text{ set } \not\subseteq \text{members}(\text{sets}_\text{-seq}_i[i])) \text{ do}
(14) \text{to\_deliver}_i \leftarrow \text{ set } \setminus \text{members}(\text{sets}_\text{-seq}_i[i])
(15) \text{sets}_\text{-seq}_i[i] \leftarrow \text{sets}_\text{-seq}_i[i] \oplus \text{to\_deliver}_i; \text{SETS}_\text{-SEQ}[i] \leftarrow \text{sets}_\text{-seq}_i[i];
(16) \text{scd\_deliver}(\text{to\_deliver}_i)
(17) \text{end while}.
```

Algorithm 2: An implementation of SCD-broadcast in C\textsc{ARW}_{n,t}[snapshot] (code for \(p_i\))
The procedure progress() first invokes the internal procedure catch_up(), 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, catch_up() 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 sc-delivers it (line 16). This local sc-delivery needs to update accordingly both $sets_seq_i[i]$ (local update) and $SETS_SEQ[i]$ (global update).

When it returns from catch_up(), $p_i$ 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.

### 6.2 Proof of Algorithm 2

**Lemma 6** 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. $\square$ Lemma 6

**Lemma 7** 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 $to_deliver_i$. $\square$ Lemma 7

**Lemma 8** 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 catch_up(). 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 catch_up() 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(). $\square$ Lemma 8

**Lemma 9** 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 sc-delivered at line 9, which concludes the proof of the lemma. $\square$ Lemma 9

**Lemma 10** 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 sc-deliver a set of messages containing $m$ at line 9. $\square$ Lemma 10
Lemma 11 Let $p_i$ be a process that scd-delivers a set $ms_j$ containing a message $m$ and later scd-delivers a set $ms'_j$ 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 $m$.

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 $sets_{seq}[i]$), 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 catch_up() cannot violate the MS-Ordering property, which is established at lines 6-10.

\[\square\] Lemma 11

Theorem 2 Algorithm 2 implements the SCD-Broadcast abstraction in the system model $CARW_{n,t}[t < n]$.

Proof The proof follows from Lemma 6 (Validity), Lemma 7 (Integrity), Lemmas 8 and 9 (Termination-1), Lemma 10 (Termination-2), and Lemma 11 (MS-Ordering).

\[\square\] Theorem 2

7 Conclusion

This paper has introduced a new communication abstraction (SCD-broadcast) providing processes with an abstraction level between reliable broadcast and total order broadcast (which captures the necessary and sufficient constraint on message deliveries which allows consensus objects to be implemented in asynchronous crash-prone message-passing systems).

More precisely, SCD-broadcast captures the abstraction level which is “necessary and sufficient” to implement read/write registers and snapshot objects on top of asynchronous message-passing systems prone to process failures. “Sufficient” means here that no other notion or object\(^5\) is needed to build a register or a snapshot object at the abstraction level provided by SCD-broadcast, while “necessary” means that the objects that are built (registers and snapshot objects) are the weakest from a shared memory computational point of view.

As announced in the Introduction, an algorithm implementing SCD-broadcast in an asynchronous message-passing system where any minority of processes may crash is described in Appendix A. This algorithm requires $O(n^2)$ protocol messages per invocation of scd_broadcast(). It follows that the SCD-broadcast-based MWMR snapshot algorithm presented in the paper requires $O(n^2)$ protocol messages per invocation of snapshot() or write() operation. This is the best read/write snapshot algorithm we know in the context of asynchronous message-passing systems.

---

\(^5\)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).

\(^6\)The notion of intersecting quorums is neither provided by the abstraction level offered by SCD-broadcast, nor required—in addition to SCD-broadcast—to implement registers or snapshot objects. Actually, it is hidden and majority quorums appear only in the implementation of SCD-broadcast.
Acknowledgments

This work has been partially supported by the Franco-German DFG-ANR Project 40300781 DISCMAT (devoted to connections between mathematics and distributed computing), and the French ANR project DESCARTES (devoted to layered and modular structures in distributed computing). The authors want to thank Faith Ellen for fruitful exchanges on shared memory snapshot.

References


[27] Misra J., Axioms for memory access in asynchronous hardware systems. ACM Transactions on Programming Languages and Systems, 8(1):142-153 (1986)


A An Implementation of SCD-broadcast in Message-Passing Systems

This section shows that the SCD-broadcast communication abstraction is not an oracle-like object which allows us to extend our understanding of computing, but cannot be implemented. It describes an implementation of SCD-broadcast in $\text{CAM}^{\text{P}^t_n, t < n/2}$, which 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] (see footnote 4).

To simplify the presentation, and without loss of generality, we consider that the communication channels are FIFO. The associated communication operations are denoted $\text{fifo} \_\text{broadcast}()$ and $\text{fifo} \_\text{deliver}()$.

A.1 Algorithm 3

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

- $\text{buffer}_i$: buffer where are stored the messages not yet scd-delivered in a message set.
- $\text{to} \_\text{deliver}_i$: next set of messages to be scd-delivered.
- $\text{sn}_i$: local sequence number (initialized to 0), which measures the local progress of $p_i$.
- $\text{clock}_i[1..n]$: array of sequence numbers. $\text{clock}_i[j]$ is the greatest sequence number $x$ such that the application message identified by $\langle x, j \rangle$ was in a message set scd-delivered by $p_i$.

Operation scd\_broadcast() When $p_i$ invokes scd\_broadcast$(m)$, where $m$ is an application message, it sends the message FORWARD$(m, i, \text{sn}_i, i, \text{sn}_i)$ to itself (this simplifies the writing of the algorithm), and waits until it has no more message from itself pending in $\text{buffer}_i$, which means it has scd-delivered a set containing $m$.

A protocol message FORWARD() (line 1) is made up of five fields: the associated application message $m$, and two pairs, each made up of a sequence number and a process identity. The first pair $(sd, \text{sn}_i)$ is the identity of the application message, while the second one $(f, \text{sn}_f)$ is the local progress $(\text{sn}_f)$ of the forwarder process $p_f$ when it forwards this protocol message.

Reception of FORWARD$(m, sd, \text{sn}_d, f, \text{sn}_f)$ When a process $p_i$ receives such a protocol message, it first invokes forward$(m, sd, \text{sn}_d, f, \text{sn}_f)$ to participate in the reliable broadcast of this message (line 3), and then invokes try\_deliver() to see if a message set can be scd-delivered (line 4).

Procedure forward() This procedure can be seen as an enrichment (with the fields $f$ and $\text{sn}_f$) of the reliable broadcast implemented by the messages FORWARD$(m, sd, \text{sn}_d, i, \text{sn}_i)$ to itself (this simplifies the writing of the algorithm), and waits until it has no more message from itself pending in $\text{buffer}_i$, which means it has scd-delivered a set containing $m$.

- The message $m$ is not in $\text{buffer}_i$. In this case, $p_i$ creates a quadruplet $\text{msg}$, and adds it to $\text{buffer}_i$ (lines 8-10). This quadruplet $(\text{msg}\_m, \text{msg}\_sd, \text{msg}\_f, \text{msg}\_cl)$ is such that
  - the field $\text{msg}\_m$ contains the application message $m$,
  - the field $\text{msg}\_sd$ contains the id of the sender of this application message,
  - the field $\text{msg}\_sn$ contains the local date associated with $m$ by its sender, 
  - the field $\text{msg}\_cl$ is an array of size $n$, such that $\text{msg}\_cl[x] = \text{sequence number (initially } +\infty\text{)}$ associated with $m$ by $p_x$ when it broadcast FORWARD$(m, i, \text{sn}_i, i, \text{sn}_i)$ to itself (line 6). This last field is crucial in the scd-delivery of a message set containing $m$.

After the quadruplet $\text{msg}$ has been built, $p_i$ first adds it to $\text{buffer}_i$ (line 10), and invokes (line 11) fifo\_broadcast FORWARD$(m, sd, \text{sn}_d, i, \text{sn}_i)$ to implement the reliable broadcast of $m$ identified by $(sd, \text{sn}_d)$. Finally, $p_i$ records its progress by increasing $\text{sn}_i$ (line 12).

- There is a quadruplet $\text{msg}$ in $\text{buffer}_i$ associated with $m$, i.e., $\text{msg} = \langle m, sd, i, \text{sn}_i \rangle \in \text{buffer}_i$ (predicate of line 6). In this case, $p_i$ assigns $\text{sn}_f$ to $\text{msg}\_cl[f]$ (line 7), thereby indicating that $m$ was known and forwarded by $p_f$ at its local time $\text{sn}_f$. 


delivering a set containing $m$ messages. The first of these messages ever sent was sent after a process invoked $\text{try\_deliver}()$.

Theorem 12 (Proof of Algorithm 3) For any $p_i$ that invokes $\text{try\_deliver}()$, when it executes $\text{try\_deliver}()$, $p_i$ first computes the set $\text{to\_deliver}_i$ of the quadruplets $\text{msg}$ 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 $\text{msg}.$cl.$f$ has been set to a finite value (line 7). If a majority of processes received first a message $\text{FORWARD}(m',-, -, -, -)$ and later another message $\text{FORWARD}(m, -, -, -, -)$, it would 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 $\text{msg}$ corresponding to $m$ if it has not enough information to deliver $m'$ (i.e. the corresponding $\text{msg}'$ is not in $\text{to\_deliver}_i$ or it does not have the proof that the situation cannot happen, i.e. no majority of processes saw the message corresponding to $m$ before the message corresponding to $m'$).

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

A.2 Proof of Algorithm 3
Lemma 12 If a process scd-delivers a set containing $m$, some process invoked $\text{scd\_broadcast}(m)$.

Proof If process $p_i$ scd-delivers a set containing a message $m$, it has previously added into $\text{buffer}_i$ a quadruplet $\text{msg}$ such that $\text{msg}.$m = $m$ (line 10), for which it has fifo-received at least $\frac{n}{2}$ $\text{FORWARD}(m, -, -, -, -)$ messages. The first of these messages ever sent was sent after a process invoked $\text{scd\_broadcast}(m)$. \(\Box\) Lemma 12

Lemma 13 No process scd-delivers the same message twice.

Proof After a message $m$ scd-broadcast by $p_{sd}$ with a sequence number $\text{sn}_{sd}$ is scd-delivered by $p_i$, $\text{clock}_i[\text{sd}] \geq \text{sn}_{sd}$ thanks to line 18 and there is no $\text{msg} \in \text{buffer}_i$ with $\text{msg}.$sd = $sd$ and $\text{msg}.$sn = $\text{sn}_{sd}$, as it was re-
moved on line 19. Thanks to line 5, no such \( \text{msg}' \) will be added again in \( \text{buffer}_i \). As \( \text{to\_deliver}_i \) is defined as a subset of \( \text{buffer}_i \) on line 15, \( m \) will never be scd-delivered by \( p_i \) again. \( \square \) 

**Lemma 14** If a message FORWARD\((m, sd, sn_{sd}, i, sn_i)\) is broadcast by a non-faulty process \( p_i \), then each non-faulty process \( p_j \) broadcasts a single message FORWARD\((m, sd, sn_{sd}, j, sn_j)\).

**Proof** First, we prove that \( p_j \) broadcasts a message FORWARD\((m, sd, sn_{sd}, i, sn_i)\). 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, \( \text{buffer}_i \) contains a value \( \text{msg} \) with \( \text{msg}.sd = sd \) and \( \text{msg}.sn_{sd} = sn_{sd} \). That \( \text{msg} \) 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 \( \text{msg}' \) added to \( \text{buffer}_j \) after \( p_j \) received a (first) message FORWARD\((\text{msg}'.m, sd, sn_{sd}, f, clock_j[|sd|])\) from \( p_j \). Because the messages FORWARD() are fifo-broadcast (hence they are delivered in their sending order), \( p_{sd} \) sent message FORWARD\((\text{msg}.m, sd, sn_{sd}, sd, sn_{sd})\) before FORWARD\((\text{msg}'.m, sd, clock_j[|sd|], sd, clock_j[|sd|])\), and all other processes only forward messages, \( p_j \) received a message FORWARD\((\text{msg}.m, sd, sn_{sd}, --, --)\) from \( p_j \) before the message FORWARD\((\text{msg}'.m, sd, clock_j[|sd|], --, --)\). 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 \( \text{msg} \in buffer_j \) with \( ts(\text{msg}) = (sd, sn_{sd}) \), 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 \( ts(\text{msg}) \) and \( p_j \) will never execute line 11 with the same stamp \( (sd, sn_{sd}) \) later. \( \square \) 

**Lemma 15** Let \( p_i \) be a process that scd-delivers a set \( m_i \) containing a message \( m \) and later scd-delivers a set \( m_j \) containing a message \( m' \). No process \( p_j \) scd-delivers first a set \( m_j \) containing \( m' \) and later a set \( m_i \) 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 \( m_i \) containing \( m \) and later scd-delivers a set \( m_j \) containing \( m' \) and \( p_j \) scd-delivers a set \( m_j \) containing \( m' \) and later scd-delivers a set \( m_i \) containing \( m \).

When \( m \) is delivered by \( p_i \), there is an element \( \text{msg} \in buffer_i \) such that \( \text{msg}.m = m \) 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 \( \text{msg}' \in buffer_i \) such that \( \text{msg}'.m = m' \), 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). Therefore, 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, \( \text{msg}' \notin to\_deliver_i \) after line 16. As the communication channels are FIFO, more than \( \frac{n}{2} \) 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 both sent a message FORWARD\((m', --, --, --)\) before sending FORWARD\((m, --, --, --)\) and sent a message FORWARD\((m', --, --, --)\) before sending FORWARD\((m, --, --, --)\). However, by Lemma 14, \( p_k \) can only send one message FORWARD\((m', --, --, --)\) and one message FORWARD\((m, --, --, --)\), which leads to a contradiction. \( \square \) 

**Lemma 16** If a message FORWARD\((m, sd, sn_{sd}, i, sn_i)\) is fifo-broadcast by a non-faulty process \( p_i \), this process scd-delivers a set containing \( m \).

**Proof** Let \( p_i \) be a non-faulty process. For any pair of messages \( \text{msg} \) and \( \text{msg}' \) ever inserted in \( \text{buffer}_i \), let \( ts = ts(\text{msg}) \) and \( ts' = ts'(\text{msg}') \). Let \( \rightarrow_i \) be the dependency relation defined as follows: \( ts \rightarrow_i ts' \) if \( \forall j : \text{msg}' . cl[j] < \text{msg} . cl[j] \) (i.e. the dependency does not exist if \( p_i \) knows that a majority of processes have seen the first update due to \( \text{msg}' \) before the second due to \( \text{msg} \)). Let \( \rightarrow_i^* \) denote the transitive closure of \( \rightarrow_i \).
by the same process, let us say scd according to broadcast an infinity of messages Let sn
\langle FORWARD their first message line 2 became true, so m channels are FIFO and thanks to lines 15 and 16, it means that a majority of processes have sent a message \langle containing messages associated with a timestamp It inserted a quadruplet non-faulty processes, which are in majority.

Let us suppose (by contradiction) that the timestamp (sd, snsd) associated with the message m (carried by the protocol message FORWARD(m, sd, snsd, i, sni) fifo-broadcast by pi), has an infinity of predecessors according to \rightarrow^*_i. As the number of processes is finite, an infinity of these predecessors have been generated by the same process, let us say pf. Let (f, snf(k))k\in\mathbb{N} be the infinite sequence of the timestamps associated with the invocations of the scd_broadcast() issued by pf. The situation is depicted by Figure 1.

As pi is non-faulty, pf eventually receives a message FORWARD(m, sd, snsd, i, sni), which means pf broadcast an infinity of messages FORWARD(m(k), f, snf(k), f, snf(k)) after FORWARD(m, sd, snsd, f, snf). Let (f, snf(k1)) and (f, snf(k2)) be the timestamps associated with the next two messages sent by pf, with snf(k1) < snf(k2). By hypothesis, we have (f, snf(k2)) \rightarrow^*_i (sd, snsd). Moreover, all processes received their first message FORWARD(m, sd, snsd, -,-) before their first message FORWARD(m(k), f, snf(k), -,-), so (sd, snsd) \rightarrow^*_i (f, snf(k1)). Let us express the path (f, snf(k2)) \rightarrow^*_i (f, snf(k1)):
\langle f, snf(k2) \rangle \rightarrow^*_i \langle f, snf(k1) \rangle:
\langle f, snf(k1) \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, snf(k1) \rangle.

In the time interval starting when pf sent the message FORWARD(m(k1), f, snf(k1), f, snf(k1)) and finishing when it sent the the message FORWARD(m(k2), f, snf(k2), f, snf(k2)), the waiting condition of line 2 became true, so pf scd-delivered a set containing the message m(k1), and according to Lemma 12, no set containing the message m(k2). Therefore, there is an index l such that process pf 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 pi has fifo-broadcast a message FORWARD(m, sd, snsd, i, sni) (line 10). It inserted a quadruplet msg with timestamp (sd, snsd) on line 9 and by what precedes, (sd, snsd) has a finite number of predecessors \langle sd_1, sn_1 \rangle, \ldots, \langle sd_k, sn_k \rangle according to \rightarrow^*_i. As pi is non-faulty, according to Lemma 14, 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 msg' such that \langle msg', sd, msg'.sn, snsd \rangle \rightarrow^*_i \langle sd, snsd \rangle. Let us consider the moment when pi receives the last message FORWARD(-, sd_k, sn_k, f, snf) sent by a correct process pf. For all msg' \in pred, either msg'.m has already been delivered or msg' is inserted to_deliver_i on line 15. Moreover, no msg' \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 msg' = msg, 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.

\textbf{Lemma 17} If a non-faulty process scd-broadcasts a message m, it scd-delivers a message set containing m.

\textbf{Proof} If a non-faulty process scd-broadcasts a message m, it sends a message FORWARD(m, i, snsd, i, snsd) on line 11, so it scd-delivers a message set containing m by lemma 16.

\textbf{Lemma 18} If a non-faulty process scd-delivers a message m, every non-faulty process scd-delivers a message set containing m.

\textbf{Figure 1: Message pattern introduced in Lemma 16}
Proof Suppose a non-faulty process $p_i$ scd-delivers a message $m$. At line 20, there is $msg \in to\_deliver_i$ such that $msg.m = m$. At line 15, $msg \in buffer_i$, and $msg$ was inserted in $buffer_i$ at line 10, just before $p_i$ sent message $\text{FORWARD}(m, sd, sn_{sd}, i, sn_i)$. By Lemma 14, every non-faulty process $p_j$ sends a message $\text{FORWARD}(m, sd, sn_{sd}, j, sn_j)$, so by Lemma 16, $p_j$ scd-delivers a message set containing $m$. \hfill \Box_{\text{Lemma 18}}

Theorem 3 Algorithm 3 implements the SCD-broadcast communication abstraction in $\text{CAMP}_{n,t}[t < n/2]$. Moreover, it requires $O(n^2)$ messages per invocation of $\text{scd\_broadcast()}$.

Proof The proof follows from Lemma 12 (Validity), Lemma 13 (Integrity), Lemma 15 (MS-Ordering), Lemma 17 (Termination-1), and Lemma 18 (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). \hfill \Box_{\text{Theorem 3}}

The next corollary follows from (i) Theorems 1 and 3, 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 3 is resiliency optimal.

B Building an MWMR atomic register on top of $\text{CAMP}_{n,t}[\text{SCD-broadcast}]$

This appendix shows the genericity dimension of Algorithm 1. It presents trivial simplifications of it, which build MWMR atomic registers and MWMR sequentially consistent registers.

B.1 The algorithm

Let $\text{REG}$ denote the MWMR atomic read/write register that is built. The algorithm that builds it is a trivial simplification of the snapshot Algorithm 1, namely its projection on a single MWMR atomic register.

$\text{REG}$ is now locally represented by a local variable $\text{reg}_i$ and the associated timestamp $ts_i$ initialized to $(0, -)$. The message sent at Line 9 is now $\text{WRITE}(v, (ts_i, date_i + 1, i))$, and the predicate of line 11 simplifies to “there are messages $\text{WRITE}()$”.

B.2 Proof of the algorithm

The proof is a simplified version of the proof of Theorem 1. For self-completeness, we give here its full proof even if some parts of it are “cut-and-paste” of parts of proofs given in Section 4.2. As in that section, let us associate a timestamp $ts(op)$ with each operation $op()$ as follows (this is the place where the proof is simplified with respect to a snapshot object).

- Case $op() = \text{write}(v)$. Let $p_i$ be the invoking process; $ts(op)$ is the timestamp of $v$ as defined by $p_i$ at line 9, i.e., $(ts_i, date_i + 1, i)$.
- Case $op() = \text{read}()$. Let $w$ be the value returned by the read; $ts(op)$ is then the timestamp associated with $w$ at line 15 by its writer.

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.

The reader can easily check that the statement and the proof of Lemma 1 (applied to the termination of read and write operations), and Lemma 3 (applied to the total order on the write operations, compliant with both the sequential specification of a register, and their real-time occurrence order) remain valid for the algorithm suited to an MWMR atomic read/write register. The next lemma addresses the read operations (which are simpler to manage than snapshot operations).

Lemma 19 The read/write register $\text{REG}$ is linearizable.

Proof Let us now insert each read operation in the previous (real time compliant) total order as follows.

Let $\text{read}1()$ be a read operation whose timestamp is $(date1, i)$. This operation is inserted just after the write operation $\text{write}1()$ that has the same timestamp (this write wrote the value read by $\text{read}1()$). Let us remark that, as $\text{read}1()$ obtained the value timestamped $(date1, i)$, it did not terminate before $\text{write}1()$ started.
It follows that the insertion of \texttt{read1()} into the total order cannot violate the real-time order between \texttt{write1()} and \texttt{read1}().

Let us consider the operation \texttt{write2()} that follows \texttt{write1()} in the write total order. If \texttt{read1()} \textless \texttt{write2()}, the placement of \texttt{read1()} in the total order is real-time-compliant. If \neg (\texttt{read1()} \textless \texttt{write2()}), due to the timestamp obtained by \texttt{read1()}, we cannot have \texttt{write2()} \textless \texttt{read1}(). It follows that in this case also, the placement of \texttt{read1()} in the total order is real-time-compliant.

Finally, let us consider two read operations \texttt{read1()} and \texttt{read2()} which have the same timestamp \langle date, i \rangle (hence, they read from the same write operation, say \texttt{write1}()). Both are inserted after \texttt{write1()} in the order of their invocations (if \texttt{read1()} and \texttt{read2()} started simultaneously, they are inserted according to the order on the identities of the processes that invoked them). Hence, the read and write operations are linearizable, which concludes the proof of the lemma.

\hfill $\square$ Lemma 19

\textbf{Theorem 4} The read/write register REG is an MWMR atomic read/write register.

\textbf{Proof} The proof follows from Lemma 1, Lemma 3, and Lemma 19.

\hfill $\square$ Theorem 4

\section*{B.3 The case of an SWMR atomic register}

When the register \texttt{REG} can be written by a single process (say \texttt{p_k}), the algorithm simplifies. The timestamps disappear at all processes, and as only the writer \texttt{p_k} can invoke \texttt{REG.write()}, it manages a simple date \texttt{date_k} (which is actually a sequence number). The modifications are:

- Line 9 becomes: \texttt{date_k} \leftarrow \texttt{date_k} + 1; \texttt{scd.broadcast WRITE(v, date_k)}.
- The lines 11-17 become:

\begin{verbatim}
if (there are messages WRITE())
  then let date be the maximal date in the messages WRITE() received;
  reg_i \leftarrow the value associated with date
end if.
\end{verbatim}

Let us remark that, due to the Boolean \texttt{done_k}, the writer \texttt{p_k} scd-delivers message sets containing at most one message \texttt{WRITE()}.}

\section*{B.4 On sequentially consistency}

\textbf{The case of an MWMR sequentially consistent register} As indicated in the Introduction, sequential consistency was introduced in [24]. It is atomicity minus the requirement stating that “if an operation \texttt{op1} terminates before an operation \texttt{op2} starts, then \texttt{op1} must appear before \texttt{op2} in the sequence of the read and write operations”. As noticed in [31], sequential consistency can be seen as a weakened form of atomicity, namely lazy linearizability. The composition of sequentially consistent registers is investigated in [30]. The algorithm for sequential consistency presented in [30] and Algorithm 3 are based on similar principles. The constraint \((t < n/2)\) is also a necessary and sufficient condition to implement a sequentially consistent read/write register in \texttt{CAMP}_{n,\ell}[\emptyset].

The reader can check that an algorithm building a a sequentially consistent MWMR read/write register can easily be obtained from Algorithm 1 as simplified in Section B.1. One only needs to suppress the synchronization messages \texttt{SYNC()} which ensure the compliance with respect to real-time. The concerned lines are lines 1-3 (read synchronization), and lines 5-7 (write synchronization). In a simple way, this shows the versatility dimension of Algorithm 1.

\textbf{From sequential consistency to atomicity} Given a sequentially consistent snapshot object, Algorithm 2 builds the SCD-broadcast communication abstraction. (As the reader can check, this follows from the fact that, when looking at its proof, this algorithm relies only on the fact that the operations on the snapshot object can be totally ordered.) Hence, using on top of it the SCD-broadcast-based Algorithm 1, we obtain an atomic snapshot object. It follows that, thanks to SCD-broadcast, the algorithms presented in the paper allow a sequentially consistent snapshot object to be transformed into an atomic snapshot object (and it is known that –differently from sequential consistent objects– atomic objects are composable for free [18]).