Implementing Snapshot Objects on Top of Crash-Prone Asynchronous Message-Passing Systems

Carole Delporte-Gallet[1], Hugues Fauconnier[1], Sergio Rajsbaum[2], and Michel Raynal[3(✉)]

[1] IRIF, Université Paris Diderot, Paris, France
[2] Instituto de Matemáticas, UNAM, 04510 México D.F, Mexico
[3] IUF and IRISA, Université de Rennes, Rennes, France
raynal@irisa.fr

Abstract. Distributed snapshots, as introduced by Chandy and Lamport in the context of asynchronous failure-free message-passing distributed systems, are consistent global states in which the observed distributed application might have passed through. It appears that two such distributed snapshots cannot necessarily be compared (in the sense of determining which one of them is the "first"). Differently, snapshots introduced in asynchronous crash-prone read/write distributed systems are totally ordered, which greatly simplify their use by upper layer applications.

In order to benefit from shared memory snapshot objects, it is possible to simulate a read/write shared memory on top of an asynchronous crash-prone message-passing system, and build then snapshot objects on top of it. This algorithm stacking is costly in both time and messages. To circumvent this drawback, this paper presents algorithms building snapshot objects *directly* on top of asynchronous crash-prone message-passing system. "Directly" means here "without building an intermediate layer such as a read/write shared memory". To the authors knowledge, the proposed algorithms are the first providing such constructions. Interestingly enough, these algorithms are efficient and relatively simple.

Keywords: Asynchronous message-passing system · Atomic read/write register · Linearizability · Process crash failure · Snapshot object

1 Introduction

Snapshots in Message-Passing Systems. Being able to compute global states of message-passing distributed applications is a central issue of distributed computing. This is because many problems can be stated as properties on global states.

M. Raynal—The French authors were partially supported by the French ANR project DESCARTES devoted to abstraction layers in distributed computing. The third author was supported in part by UNAM PAPIIT-DGAPA project IN107714.

J. Carretero et al. (Eds.): ICA3PP 2016, LNCS 10048, pp. 341–355, 2016.
DOI: 10.1007/978-3-319-49583-5_26

One of the most famous example is the detection of stable properties of distributed computations, such as termination detection or deadlock detection (once true, a stable property remains true forever).

One of the very first algorithms computing consistent global states of a distributed computation is due to Chandy and Lamport [5]. This simple and elegant algorithm introduced the term *snapshot* to denote a computed global state. It assumes FIFO channels, and uses additional control messages called *markers*. Later, snapshot algorithms, which require neither FIFO channels nor additional control messages, have been introduced (e.g., [9,14]).

It was shown in [5] that, while the snapshot returned by a snapshot algorithm is consistent, it is impossible to prove that the computation passed through it. It is only possible to claim a very weak property, namely that the computation could have passed through it. This has sometimes been called the relativistic nature of distributed computing. More generally, it was shown in [6] that the set of consistent global states that can be computed has a lattice structure. This means that if two processes launch concurrently two independent snapshot computations, each process obtain a consistent snapshot, but the snapshots they obtain, not only can be different, but can be incomparable in the sense that it is impossible to show that one of them occurred before the other one (the interested reader will find a pedagogical presentation of these issues in Chap. 6 of [18]). As far as fault-tolerance is concerned, the message-passing snapshot algorithms described in [5,9,14] assume failure-free systems (no process crash).

Snapshots in Shared Memory Read/Write Systems. Considering crash-prone asynchronous systems where the processes communicate by accessing atomic Single-Writer/Multi-Reader (SWMR) registers, the notion of a *snapshot object* was introduced in [1]. *Crash-prone* means here that any number of processes may unexpectedly stop progressing. *Atomic registers* means that each read or write operation appears as if it has been executed instantaneously at some point between its start and its end, and each read of a register returns the value written by the closest preceding write on this register. The term *Linearizability* introduced in [11] is synonym of atomicity. A correct sequence of read and write operations is called a *linearization* of these operations, and the time at which an operation appear to be instantaneously executed (linearized) is called its *linearization point*.

In this context a snapshot object is composed of n SWMR atomic registers, where n is the number of processes, which means that, while each process can read all registers, it can write only "its" register. The snapshot object offers to the processes a higher abstraction level, defined by two operations, denoted write() and snapshot(). A process invokes write() to define the value of its atomic register. When it invokes snapshot(), a process obtains the whole array of registers as if it read them simultaneously. Said differently, a snapshot object is atomic (linearizable): the operations write() and snapshot() appear as if they have been executed one after the other.

In a very interesting way, it is possible to build a snapshot object on top of SWMR atomic registers in a system of n asynchronous processes where up to

$t = n - 1$ of them may crash [1]. This progress condition, which tolerates any number of process crashes, is called the *wait-freedom* [10]. More precisely, any process that executes an operation and does not crash, terminates it whatever the behavior of the other processes.

Snapshot objects have a lot of applications in crash-prone asynchronous systems where processes communicate through a read/write shared memory (examples of algorithms based on snapshot objects can be found in several following textbooks such as [4,19,20]. This comes from the fact that a snapshot object allows processes to define and use consistent global states of a read/write-based computation: each process deposits the relevant part of its local state in the snapshot object, and can then obtain consistent global states by invoking the operation snapshot().

The previous snapshot object considers that each process has its "own" underlying atomic register. Hence, they are called SWMR snapshot objects. Snapshot objects, where the underlying atomic registers are Multi-Writer/Multi-Reader (MWMR) registers, have also been studied (e.g., [12,13]).

Construction of Read/Write Registers in Message-Passing Systems. Read/write registers are the most basic objects of computing science, and consequently, a fundamental problem of asynchronous message-passing distributed systems consists in building an SWMR or MWMR atomic register providing the processes with a higher abstraction level than message-passing. This allows to use read/write-based algorithms on top of message-passing systems. Moreover, as in distributed systems "failures are not on option but are blunded with software", such constructions must tolerate as many process failures as possible.

One of the most celebrated algorithm implementing an atomic read/write register on top of an asynchronous message-passing system is the algorithm due to Attiya et al. [3], called ABD in the literature. This construction copes with up to $t < n/2$ process crashes, which has been shown (in the same paper) to be an upper bound on the number of process crashes that can be tolerated. The algorithms, which implement the read and write operations, are particularly simple. They use of a simple broadcast facility, sequence numbers, and majority quorums. The fact that (a) any quorum contains at least one process that never crashes, and (b) any two majority quorums have a non-empty intersection, are key elements of this construction.

Many constructions of atomic read/write registers on top of message-passing systems have been proposed (e.g., [2,4,8,16–18] to cite a few). They differ in the type and the number of failures they tolerate, the number of messages they need to implement a read or a write operation, the size of control information carried by these implementation messages, and the time complexity of each operation.

Content of the Paper. This paper is on the construction of a (high level) t-tolerant SWMR snapshot object on top of an underlying (low level) asynchronous message-passing system where up to t processes may crash. As $t < n/2$ is an upper bound on the number of process crashes to build an read/write atomic

register on top of a crash-prone message-passing system, it follows that $t < n/2$ remains an upper bound when one wants to build a snapshot object.

A simple way to obtain such a construction consists first in using an algorithm (such as one of the previously mentioned ones) to build n SWMR atomic registers on top of the crash-prone asynchronous message-passing system, and then use any algorithm building an SWMR snapshot object (e.g., [1,12]) on top of the read/write shared memory build previously. This construction consists of a simple stacking of existing algorithms: the first layer going from message-passing to n SWMR atomic registers, the second layer going from n SWMR atomic registers to a snapshot object.

While it obeys basic structuring principles, this solution is not satisfactory for the following reason. The stacking-based construction is not genuine. More precisely, building intermediate SWMR atomic registers is a way to build a snapshot object, but is not a problem requirement. Maybe there are simpler and more efficient constructions, which build directly a snapshot object on top of a message-passing system, without requiring this intermediate level. Moreover, being not genuine, the stacking-based construction can be more costly and its engineering more difficult than an ad'hoc construction.

The paper presents a genuine construction of an SWMR snapshot object on top of a message-passing system in which, in any run, any minority of processes may crash. From a number of messages point of view, a write operation requires $O(n)$ messages, while a snapshot operation requires between $O(n)$ and $O(n^2)$ messages (this depends on the concurrency pattern involving the snapshot operation and the number of concurrent write operations). From a time complexity point of view, a write operation requires a round-trip delay, while a snapshot operation requires between one and $(n-1)$ round-trip delays (as before this depends on the concurrency pattern occurring during the snapshot).

Roadmap. The paper is made up of 6 sections. Section 2 presents the basic definitions: system model, one-shot and multi-shot snapshot objects. Section 3 presents a genuine algorithm constructing a one-shot snapshot object. Section 4 proves its correctness. Section 5 shows how to modify the previous algorithm to go from a one-shot to a multi-shot snapshot object. Finally, Sect. 6 concludes the paper. All missing proofs can be found in [7].

2 System Model, and Snapshot Objects

System Model

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

A process may halt prematurely (crash failure), but executes correctly its local algorithm until it possibly crashes. 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 *correct* or *non-faulty*. Let us notice that, as a faulty process behaves correctly until it crashes, no process knows if it is correct or faulty.

Communication. The processes cooperate by sending and receiving messages through bi-directional channels. The communication network is a complete network, which means that any process p_i can directly send a message to any process p_j (including itself). 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).

A process p_i invokes the operation "send TAG(m) to p_j" to send to p_j the message tagged TAG which carries the value m. It receives a message tagged TAG by invoking the operation "receive TAG()". The macro-operation "broadcast TAG(m)" is a shortcut for "**for each** $j \in \{1, \ldots, n\}$ send TAG(m) to p_j **end for**". (The sending order is arbitrary, which means that, if the sender crashes while executing this macro-operation, an arbitrary – possibly empty – subset of processes will receive the message.)

Let us notice that, due to process and message asynchrony, no process can know if an other process crashed or is only very slow.

Notation. In the following, the previous computation model, restricted by the feasibility predicate $t < n/2$, is denoted $\mathcal{CAMP}_{n,t_{n,t}}[t < n/2]$ ("Crash Asynchronous Message-Passing" model in which any minority of processes may crash).

It is important to notice that, in this model, all processes are a priori "equal". This allows each process to be at the same time a "client" (it invokes high level operations) and a "server" (it locally participates in the implementation of the object that is built).

Message types are denoted with small capital letters, while local variables are denoted with small italics letters, indexed by a process index.

Snapshot Object

Definition. The SWMR snapshot object has been informally presented in the Introduction. It is made up of n components (one per process), and provides the processes with two operations denoted write() and snapshot().

Let $SNAP$ be such an object. When a process p_i invokes write(v), it stores the value v in its component $SNAP[i]$. When a process p_i invokes snapshot(), it obtains the value of all the components $SNAP[1..n]$. A snapshot object is atomic (or linearizable), which means that the operations write() and snapshot() issued by the processes appear as if each of them had been executed instantaneously, at a single point of the time line between its start and its end. Moreover, no two operations appear at the same point of the time line, and the array $reg[1..n]$ returned by a process, when it terminates an invocation of snapshot(), is such that $reg[k] = v$ if the closest preceding write operation issued by p_k is write(v). If there is no such write by p_k, $reg[k] = \perp$ (a default value that, at the application level, no process can write).

One-Shot vs Multi-Shot. In the context of snapshot objects, we distinguish one and multi-shot objects. In both cases, a process can issue as many operations snapshot() as it wants.

- One-shot. No process invokes write(v) more than once.
- Multi-shot. There is no restriction on the number of times a process can invoke write().

In the following we consider first the implementation of a one-shot snapshot object. This construction is then generalized to the case of a multi-shot snapshot object in Sect. 5.

3 Implementing a One-Shot Snapshot Object

Algorithm 1 implements a one-shot snapshot object.

Local Representation of the Snapshot Object. Each process p_i manages a local array $reg_i[1..n]$, which contains its current view of the snapshot object. This array is initialized to $[\bot, \cdots, \bot]$.

Each process p_i manages also a sequence number ssn_i. Initialized to 0, this local variable is used to identify the successive requests generated by the invocations of the operation snapshot() issued by p_i.

Algorithm Implementing the Operation write(v)*: Client Side.* This algorithm is described at lines 1–6, executed by the invoking process p_i (client), and lines 15–16, executed by all processes (in their server role).

When p_i invokes write(v), it assigns the value v to its local register $reg_i[i]$ and broadcasts the message WRITE(reg_i) to inform the other processes of its write (lines 1–2). Then, p_i waits for acknowledgments (line 3). Each message WRITE_ACK(reg) carries the current value of $reg_j[1..n]$ of the sender p_j. After p_i received acknowledgments from a majority of processes, it updates its local view of the snapshot object, namely $reg_i[1..n]$, to have it as recent as possible (line 5). This is done, for each local register $reg_i[k]$, by taking the maximum on the value it received and its current value. As we consider here a one-shot snapshot object, a process invokes write() at most once, and consequently, the values in $reg_i[k], reg(1)[k], \cdots, reg(m)[k]$ are all equal to $\bot$ if p_k has not yet invoked write(), or belong to the set $\{\bot, v\}$ if p_k invoked write(v). After the update of $reg_i[1..n]$ is done, p_i returns from the operation.

Algorithm Implementing the Operation write(v)*: Server Side.* On the server side, when p_i receives a message WRITE(reg) from a process p_j, it updates its local array $reg_i[1..n]$ to have it as up to date as possible (line 15). It then sends back to p_j the acknowledgment message WRITE_ACK(reg_i) (line 16). As seen above, if p_i knows writes not yet known by p_j, this message allows p_j to known them.

Algorithm Implementing the Operation snapshot()*: Client Side.* As previously, this algorithm is decomposed in two parts. The part described at lines 8–14 is

local variables initialization:
$ssn_i \leftarrow 0$; $reg_i \leftarrow [\bot, \cdots, \bot]$ ($\bot$ is smaller than any value written by a process).
%
operation write(v) **is**
(1) $reg_i[i] \leftarrow v$;
(2) broadcast WRITE(reg_i);
(3) **wait** (WRITE_ACK(reg) received from a majority of processes);
(4) **let** $reg(1)$, .., $reg(m)$ be the arrays received at the previous line;
(5) **for** $k \in \{1, .., n\}$ **do** $reg_i[k] \leftarrow \mathsf{max}(reg_i[k], reg(1)[k], .., reg(m)[k])$ **end for**;
(6) return()
end operation.

operation snapshot() **is**
(7) **repeat**
(8) $prev \leftarrow reg_i$;
(9) $ssn_i \leftarrow ssn_i + 1$; broadcast SNAPSHOT($reg_i, ssn_i$);
(10) **wait** (SNAPSHOT_ACK(reg, ssn_i) received from a majority of processes);
(11) **let** $reg(1)$, .., $reg(m)$ be the arrays received at the previous line;
(12) **for** $k \in \{1, .., n\}$ **do** $reg_i[k] \leftarrow \mathsf{max}(reg_i[k], reg(1)[k], .., reg(m)[k])$ **end for**
(13) **until** $prev = reg_i$ **end repeat**;
(14) return(reg_i)
end operation.
%
when a message WRITE(reg) **is received from** p_j **do**
(15) **for** $k \in \{1, \cdots n\}$ **do** $reg_i[k] \leftarrow \mathsf{max}(reg_i[k], reg[k])$ **end for**;
(16) send WRITE_ACK(reg_i) to p_j.

when a message SNAPSHOT(reg, ssn) **is received from** p_j **do**
(17) **for** $k \in \{1, .., n\}$ **do** $reg_i[k] \leftarrow \mathsf{max}(reg_i[k], reg[k])$ **end for**;
(18) send SNAPSHOT_ACK(reg_i, ssn) to p_j.

Fig. 1. One-shot snapshot object in $\mathcal{CAMP}_{n,t}[t < n/2]$ (code for p_i)

executed by the invoking process p_i (client), while lines 17–18 are executed by all processes (in their server role).

The invoking process enters a repeat loop that it will exit when, from its point of view, its local array $reg_i[1..n]$ can no longer be enriched with new values. To this end it uses a local array variable $prev[1..n]$ (whose scope is restricted to the operation snapshot()). After it assigned reg_i to $prev$, p_i broadcasts an inquiry message SNAPSHOT(reg_i, ssn_i), in which the sequence number ssn_i is used to identify the different inquiries broadcast by p_i.

Then, p_i has exactly the same behavior as the one described at lines 3–5 of the write operation. Namely, p_i waits for acknowledgment messages from a majority of processes (those are messages SNAPSHOT_ACK(reg, ssn_i) carrying the appropriate sequence number). Hence, after it has executed lines 10–12, p_i possibly updated its local representation $reg_i[1..n]$ of the snapshot object.

Then, if reg_i has been updated (we have then $reg_i \neq prev$ at line 3), p_i re-enters the repeat loop. If reg_i has not been enriched with new values during the last iteration, p_i returns it as result of it snapshot invocation.

Algorithm Implementing the Operation snapshot(v)*: Server Side.* This part (reception of a message SNAPSHOT(reg, ssn) from a process p_j, lines 17–18) is the same as the reception of a message WRITE(reg, ssn). Namely, p_i updates $reg_i[1..n]$ and sends back to p_j an acknowledgment message SNAPSHOT_ACK(reg_i, ssn).

4 Proof of the One-Shot Snapshot Algorithm

4.1 Termination

Lemma 1. *If a correct process p_i invokes* write()*, it terminates. Any invocation of* snapshot() *by a correct process terminate.*

4.2 Definitions and Notations

The following definitions are from [11]. For simplicity, and as they are sufficient for the understanding, we consider here only the failure-free case.

Events. Let op be an operation write() or snapshot(). The execution of an operation op by a process p_i is modeled by two events: an *invocation event*, denoted *invoc*(op), which occurs when p_i invokes the operation, and a *response event*, denoted *resp*(op), which occurs when p_i terminates the operation. The event *invoc*(op) of an operation op occurs when it executes its first statement (line 1 or line 8), and its event *resp*(op) (termination) occurs when it executes its return() statement (line 6 or line 14).

In addition to these events, sending and reception of messages create corresponding communication events [15]. Without loss of generality, it is assumed that no two events occur at the same time.

Histories. A *history* models a run. It is a total order on the events produced by the processes. Given any two events e and f, $e < f$ if e occurs before f in the corresponding history. Let us notice that we always have $e < f$ or $f < e$. A history is denoted $\widehat{H} = \langle E, < \rangle$, where E is the set of events.

A history is *sequential* if (a) its first event is an invocation; (b) each invocation is followed by the matching response event; and (c) each response event – except the last one if the computation is finite – is followed by a an invocation event.

$\widehat{H}|i$ is called a *local history*; it is the sub-sequence of $\widehat{H}$ made up of the events generated by process p_i. Two histories are equivalent if no process can distinguish them, i.e., $\forall\, i, j :\ \widehat{H}|i = \widehat{H}|j$.

Linearizable Snapshot History. A snapshot-based history $\widehat{H} = \langle E, < \rangle$ is *correct* (or *linearizable*) if there is an equivalent sequential history $\widehat{H_{seq}} = \langle E, <_{seq} \rangle$ in which the sequence of write() or snapshot() operations issued by the processes is such that (a) each operation appears as if it has been executed at a single point of the time line between its invocation and response events, and (b) each snapshot() operation returns an array reg such that $reg[i] = v$ if the invocation of write(v) by p_i appears previously in the sequence, and $reg[i] = \perp$ if it does not.

When considering a sequential history it is possible to associate a time instant of the time line with each operation. As, in such a history, all operations are ordered, no two operations are associated with the same time instant.

Given two arrays $reg1$ and $reg2$ returned by two snapshot operations, $reg1 \leq reg2$ is a shortcut for $\forall\ x \in [1..n]$: $(reg1[x] \neq \perp) \Rightarrow (reg2[x] = reg1[x])$, and $reg1 < reg2$ is a shortcut for $(reg1 \leq reg2) \wedge (reg1 \neq reg2)$.

Concurrent Operations. Let op_1 and op_2 be two operations. We say "op_1 precedes op_2" (denoted $\mathsf{op}_1 \rightarrow \mathsf{op}_2$) if $resp(\mathsf{op}_1) < invoc(\mathsf{op}_2)$. If $\neg(\mathsf{op}_1 \rightarrow \mathsf{op}_2)$ and $\neg(\mathsf{op}_2 \rightarrow \mathsf{op}_1)$, we say "$\mathsf{op}_1$ and op_2 are *concurrent*", which is denoted $\mathsf{op}_1 || \mathsf{op}_2$. It follows that the relation "$\rightarrow_{op}$" defined on operations is an irreflexive partial order.

4.3 Basic Lemmas

The next three Lemmas follow directly from the algorithm.

Lemma 2. *Let* $\mathsf{ww} = \mathsf{write}(v)$ *a write operation issued by a process* p_i *and* snap *a snapshot operation returning the array* reg. $(\mathsf{ww} \rightarrow \mathsf{snap}) \Rightarrow (reg[i] = v)$.

Lemma 3. *Let* $\mathsf{ww} = \mathsf{write}(v)$ *a write operation issued by a process* p_i *and* snap *a snapshot operation returning the array* reg. $(\mathsf{snap} \rightarrow \mathsf{ww}) \Rightarrow (reg[i] = \perp)$.

The following corollary is an immediate consequence of Lemmas 2 and 3.

Corollary 1. *Let* snap *be a snapshot operation returning the array* reg*, such that* $reg[i] = v$*. There is an operation* $\mathsf{write}(v)$ *issued by process* p_i*, and it is such that* $\mathsf{write}(v) \rightarrow \mathsf{snap}$ *or* $\mathsf{write}(v) || \mathsf{snap}$.

Lemma 4. *Let* snap_1 *and* snap_2 *be two snapshot operations, returning* $reg1$ *and* $reg2$*, respectively.* $(\mathsf{snap}_1 \rightarrow \mathsf{snap}_2) \Rightarrow (reg1 \leq reg2)$.

4.4 A Linearization of the Write and Snapshot Operations

Lemma 5. *Let* snap_1 *and* snap_2 *be two snapshots operations, returning* $reg1$ *and* $reg2$*, respectively. We have* $(reg1 \leq reg2) \ \vee \ (reg2 \leq reg1)$.

Lemma 6. *Let* $\mathsf{ww1} = \mathsf{write}(v1)$ *a write operation issued by a process* p_i, $\mathsf{ww2} = \mathsf{write}(v2)$ *a write operation issued by a process* p_j*, and* snap *a snapshot operation returning the array* reg. $\big((\mathsf{ww1} \rightarrow \mathsf{ww2}) \ \wedge \ (reg[j] = v2)\big) \ \Rightarrow \ ((reg[i] = v1)$.

Lemma 7. *Given a history $\widehat{H}$ produced by Algorithm 1, there is an equivalent sequential history $\widehat{H'}$ which respects the sequential specification of the one-shot snapshot object.*

Theorem 1. *Algorithm 1 implements a one-shot snapshot object in the system model $\mathcal{CAMP}_{n,t}[t < n/2]$.*

Proof. The proof follows from Lemma 1 (Termination), and Lemma 7 (Linearizability). □

5 Implementing a Multi-shot Snapshot Object

This section extends the previous algorithm from a one-shot snapshot object (at most one write per process) to a multi-shot snapshot object (any number of writes per process).

A Non-blocking Algorithm. It is easy to extend the basic algorithm depicted in Fig. 1, which assumes that each process invokes at most once the write operation, to obtain a multi-shot algorithm in which, despite $t < n/2$ process crashes, at least once process can invoke any number of write operations without being blocked forever. This progress condition is called *non-blocking* (it can be seen as absence of deadlock in the presence of failures).

The extension is as follows. A sequence number is associated with each write or snapshot operation. They are then used to ensure that any snapshot returns an array containing values such that it is possible to build a sequence of all write and snapshot invocations where each snapshot returns the array defined by the most recent write that appear before it in the sequence. This implementation is *non blocking* because (a) it ensures that all write operations terminates, and (b) all snapshot operations which are not concurrent with a write operations terminate. A snapshot operation may not terminate if infinitely often write operations are concurrent with it.

An Always Terminating Algorithm

Underlying Principles. An extension ensuring that any invocation of a write or snapshot operation, issued by a correct process, does terminate, is described in Figs. 2 and 3. To ensure this strong termination property, two mechanisms are added to the basic algorithm.

(1) Every process helps perform all snapshot operations: when a process wants to perform a snapshot operation it broadcasts its query to every process, and, when receiving this query, each process issues a basic snapshot operation (essentially identical to the one-shot snapshot of the previous section). In this way, each process participates to every snapshot operation and in particular every process is aware of all snapshots that are not currently terminated.

local variables initialization:
$snw_i \leftarrow 0$; $sns_i \leftarrow 0$; $reg_i \leftarrow [\bot, \cdots, \bot]$; **for each** $i, j : repSnap[i, j] = \bot$.

---

operation write(v) **is**
(1) $snw_i \leftarrow snw_i + 1$; $write\_pending \leftarrow (v, snw_i)$;
(2) wait($write\_pending = \bot$); return()
end operation.

operation snapshot() **is**
(3) $sns_i \leftarrow sns_i + 1$; Rbroadcast SNAP($p_i, sns_i$);
(4) wait($repSnap[i, sns_i] \neq \bot$); return($repSnap[i, sns_i]$)
end operation.

---

function base_write(wp) **is**
(5) $reg_i[i] \leftarrow wp$;
(6) broadcast WRITE(reg_i,wp);
(7) **wait until** (WRITE_ACK(reg, wp) received from a majority of processes);
(8) **let** R be the set of reg arrays received at the previous line;
(9) **for** $k \in \{1, \cdots, n\}$ **do** $reg_i[k] \leftarrow \mathsf{max}_{\prec_{sn}}\{r[k] | r \in R \cup reg_i\}$ **end for**;
(10) return()
end function.

function base_snapshot(s, t) **is**
(11) **while** $repSnap[s, t] = \bot$ **do**
(12) $prev \leftarrow reg_i$; $ssn_i \leftarrow ssn_i + 1$; broadcast SNAPSHOT($s, t, reg_i, ssn_i$);
(13) wait (SNAPSHOT_ACK(s, t, reg, ssn_i) received from a majority of processes);
(14) **let** R be the set of reg arrays received at the previous line;
(15) **for** $k \in \{1, \cdots, n\}$ **do** $reg_i[k] \leftarrow \mathsf{max}_{\prec_{sn}}\{r[k] \mid r \in R \cup reg_i\}$ **end for**;
(16) **if** $prev = reg_i$ **then** Rbroadcast END($source, sn, repSnap[source, sn]$) **end if**
(17) **end while**;
(18) return()
end function.

Fig. 2. Multi-shot snapshot object in $\mathcal{CAMP}_{n,t}[t < n/2]$ (Part 1 of the code for p_i)

(2) To ensure that the snapshot operations are not prevented from terminating by write operations, each process, when there are some snapshot operations currently not terminated, is required to wait for the termination of the oldest snapshot operation among them. In this way, eventually no write operation can be concurrent with a snapshot operation, thereby ensuring their termination.

The corresponding extended algorithm is detailed in Figs. 2 and 3, where (as before) reg_i is the current view of the memory at process p_i. This view is updated when p_i receives a WRITE() or SNAPSHOT() message. The operator $\prec_{sn}$ is on pairs (value, seq. number). It orders them according to their increasing sequence numbers: $\big((v, a) \prec_{sn} (w, b)\big) \Leftrightarrow (a < b)$.

```
Background task: repeat forever
(19)    if (write_pending ≠ ⊥)
           then base_write(write_pending); write_pending ← ⊥ end if;
(20)    if (there are messages SNAP() received and not yet processed);
(21)    then let SNAP(source, sn) be the oldest of these messages;
(22)         base_snapshot(source, sn);
(23)         wait(readSnap[source, sn] ≠ ⊥)
(24)    end if
end repeat.

when a message WRITE(reg, w) is received from p_j do
(25)    for k ∈ {1, ··· n} do reg_i[k] ← max_{≺sn}(reg_i[k], reg[k]) end for;
(26)    send WRITE_ACK(reg_i, w) to p_j.

when a message SNAPSHOT(s, t, reg, ssn) is received from p_j do
(27)    for k ∈ {1, ··· n} do reg_i[k] ← max_{≺sn}(reg_i[k], reg[k]) end for;
(28)    send SNAPSHOT_ACK(s, t, reg_i, ssn) to p_j.

when a message END(s, t, val) is received from p_j do
(29)    repSnap[s, t] ← val.
```

Fig. 3. Multi-shot snapshot object in $\mathcal{CAMP}_{n,t}[t < n/2]$ (Part 2 of the code for p_i)

Algorithms Implementing the write() *and* snapshot() *Operations.* To perform a write operation, p_i does not immediately start to realize a write operation as in the one-shot algorithm. It records the value to be written into a variable *write_pending* with an appropriate sequence number (line 1). The write operation terminates (line 2) when the write is made in the background task of the algorithm (lines 19–23).

To perform a snapshot operation, a process p_i broadcasts in a reliable way, with the help of the underlying operation Rbroadcast(),[1] the request (message SNAP()) and its associated a sequence number to all processes (including itself) (Line 3). This request is processed in the background task at lines 20 and 22. Function base_snapshot() implements a "basic" snapshot that is essentially the same as for one-shot snapshot (waiting until the process obtains two identical vectors of values for the requested snapshot). Here this basic snapshot is stopped when at least one process has terminated a basic snapshot for the requesting upper layer snapshot. More precisely, the variable *repSnap* is an array such that $repSnap[j, m]$ contains the result of the m-th snapshot initiated by process p_j (and $\perp$ before). This variable is written at line 29 when process p_i is notified

[1] The main property of such a broadcast operation is that any message delivered by a (correct or faulty) process is delivered by all correct processes, and at least the messages broadcast by the correct processes are delivered. Hence all correct processes deliver the same set of messages S, and any faulty process delivers a subset of S. Algorithms implementing reliable broadcast in the presence of process crashes are described in many textbooks (e.g. [4,17]).

(by a message END()) that at least one of basic snapshots for the requested upper layer snapshot terminated. Then $repSnap[j, m]$ contains a snapshot value of the m-th snapshot initiated by process p_j[2].

In its background task (lines 19–23), process p_i performs a write (function base_write) if there a pending write (line 19). It easy to check that the function base_write always terminates. Then, if there are some requests for upper layer snapshots (corresponding to the reception of message SNAP()), process p_i chooses the oldest request and runs a basic snapshot for this request (line 22).

Let us first notice that each process executes *sequentially* the base operations denoted base_write() and base_snapshot(). Let us also notice that a upper layer snapshot terminates as soon as it is not concurrent with processes performing write operations. This follows from the following observation. Let us assume that an upper layer snapshot does not terminate. Then, all corresponding basic snapshots it generates are necessarily stuck in the execution of the underlying basic base_snapshot(). But, if this occurs, no non-crashed process is currently running a base write operation base_write, from which follows that the upper layer snapshot operation terminates.

6 Conclusion

Since a long time, snapshot algorithms suited to asynchronous message-passing reliable systems have been proposed (e.g. in [5,9,14]). These algorithms, which consider process local states and channels states, do not cope with failures, and provides snapshots which cannot always be compared [6,18].

Differently this paper has introduced the notion of a read/write snapshot object built on top of asynchronous message-passing systems in which any minority of processes may crash. A main property of these read/write snapshot lies in their Containment property (they can be totally ordered according to their occurrence order). The paper has considered two types of such snapshot objects: one-shot (in which a process may issue as many snapshot operations as it wants, but is restricted to issue only one write operation), and multi-shot (in which there is no restriction on the number of write operations issued by each process). The paper has also presented two algorithms, one for each type of snapshot object. The two main properties of these algorithms are their fault-tolerance and the total order on the snapshot values they return.

Table 1 compares the cost of the one-shot snapshot algorithm proposed in the paper with the stacking of the read/write snapshot algorithm described in [1], executed on the emulation of SWMR atomic registers in an asynchronous message-passing system described in [3]. This comparison considers the best cases, namely it assumes that each operation is invoked in a concurrency-free context (which is the most frequent case in practice).

[2] Let us notice that it is possible that several processes wrote snapshot values in $repSnap[j, m]$ to help p_j terminate its snapshot invocation. Any of these values is a correct snapshot value.

Table 1. Cost comparison in favorable cases

	Stacking [1] on [3]	Our algorithm
Messages per write	$2n$	$2n$
Messages per snapshot	$8n$	$2n$
Write duration	one round-trip	one round-trip
Snapshot duration	4 round-trips	one round-trip

References

1. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic snapshots of shared memory. J. ACM **40**(4), 873–890 (1993)
2. Attiya, H.: Efficient and robust sharing of memory in message-passing systems. J. Algorithms **34**, 109–127 (2000)
3. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in message passing systems. J. ACM **42**(1), 121–132 (1995)
4. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations and Advanced Topics, 2nd edn, 414 p. Wiley-Interscience (2004)
5. Chandy, K.M., Lamport, L.: Distributed snapshots: determining global states of distributed systems. ACM Trans. Comput. Syst. **3**(1), 63–75 (1985)
6. Cooper, R., Marzullo, K.: Consistent detection of global predicates. In: Proceedings of Workshop on Parallel and Distributed Debugging. ACM press (1991)
7. Delporte, C., Fauconnier, H., Rajsbaum, S., Raynal, M.: Implementing snapshot objects on top of crash-prone asynchronous message-passing systems, 15 p. Technical report 2037, IRISA, Université de Rennes (F) (2016)
8. Dutta, P., Guerraoui, R., Levy, R., Vukolic, M.: Fast access to distributed atomic memory. SIAM J. Comput. **39**(8), 3752–3783 (2010)
9. Hélary, J.-M., Mostéfaoui, A., Raynal, M.: Communication-induced determination of consistent snapshots. IEEE TPDS **10**(9), 865–877 (1999)
10. Herlihy, M.P.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. (TOPLAS) **13**(1), 124–149 (1991)
11. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM TOPLAS **12**(3), 463–492 (1990)
12. Imbs, D., Raynal, M.: Help when needed, but no more: efficient read/write partial snapshot. J. Parallel Distrib. Comput. **72**(1), 1–12 (2012)
13. Inoue, M., Masuzawa, T., Chen, W., Tokura, N.: Linear-time snapshot using multi-writer multi-reader registers. In: Tel, G., Vitányi, P. (eds.) WDAG 1994. LNCS, vol. 857, pp. 130–140. Springer, Heidelberg (1994). doi:10.1007/BFb0020429
14. Lai, T.H., Yang, T.H.: On distributed snapshots. Inf. Process. Lett. **25**, 153–158 (1987)
15. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
16. Mostéfaoui, A., Raynal, M.: Two-bit messages are sufficient to implement atomic read/write registers in crash-prone systems. In: Proceedings of 35th International ACM Symposium on Principles of Distributed Computing (PODC 2016), pp. 381–390. ACM Press (2016)

17. Raynal, M.: Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems, 251 p. Morgan & Claypool Publishers (2010). ISBN 978-1-60845-293-4
18. Raynal, M.: Distributed Algorithms for Message-Passing Systems, 510 p. Springer (2013). ISBN 978-3-642-38122-5
19. Raynal, M.: Concurrent Programming: Algorithms, Principles and Foundations, 515 p. Springer (2013). ISBN 978-3-642-32026-2
20. Taubenfeld, G.: Synchronization Algorithms and Concurrent Programming, 423 p. Pearson Prentice-Hall (2006). ISBN 0-131-97259-6