HAL
open science

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

Carole Delporte, Hugues Fauconnier, Sergio Rajsbaum, Michel Raynal

To cite this version:

Carole Delporte, Hugues Fauconnier, Sergio Rajsbaum, Michel Raynal. Implementing Snapshot Objects on Top of Crash-Prone Asynchronous Message-Passing Systems. 2016. hal-01313584v2

HAL Id: hal-01313584

https://inria.hal.science/hal-01313584v2

Preprint submitted on 31 May 2016

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.

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

Carole Delporte[†], Hugues Fauconnier[†], Sergio Rajsbaum[◦], Michel Raynal[⋆,‡]

[†] IRIF, Université Paris Diderot, Paris , France
[◦] Instituto de Matemáticas, UNAM, México D.F, 04510, México
[⋆] Institut Universitaire de France
[‡] IRISA, Université de Rennes, 35042 Rennes, France

Tech Report #2037, 16 pages, May 2016
IRISA, University of Rennes 1, France

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. One of the most famous example is the detection of stable properties of distributed computations, such as termination detection [10] 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 [8]. 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., [12, 17]).

It was shown in [8] 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 [9] 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 Chapter 6 of [25]). As far as fault-tolerance is concerned, the message-passing snapshot algorithms described in [8, 12, 17] 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 Single-Writer/Multi-Reader (SWMR) atomic read/write registers, the notion of a *snapshot object* was introduced in [1, 2]. *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 [19, 21]. The term *Linearizability* introduced in [14] 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 $\mathsf{write}()$ and $\mathsf{snapshot}()$. A process invokes $\mathsf{write}()$ to define the value of its atomic register. When it invokes $\mathsf{snapshot}()$, a process obtains the whole array of registers as if it read them simultaneously. Said differently, a snapshot object is atomic (linearizable): the operations $\mathsf{write}()$ and $\mathsf{snapshot}()$ 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* [13]. 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 (e.g. [7, 20, 26, 28]). This comes from the fact that a snapshot object allows processes to define and use consistent global states of a read/write-based com-

putation: 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 $\mathsf{snapshot}()$.

The previous snapshot object considers that each process has its "own" underlying atomic register. Hence, they are called SWMR snapshot objects. Snapshot objects were the underlying atomic registers are MWMR (Multi-Writer/Multi-Reader) have also been studied (e.g., [5, 15, 16]).

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, Bar-Noy, and Dolev [4], 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., [3, 7, 11, 20, 22, 23, 24, 25] 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 [27].

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, 6, 15]) 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, Section 6 concludes the paper.

2 System Model, and Snapshot Objects

2.1 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.

2.2 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 $\mathsf{write}(v)$, it stores the value v in its component $SNAP[i]$. When a process p_i invokes $\mathsf{snapshot}()$, it obtains the value of all the components $SNAP[1..n]$. A snapshot object is atomic (or linearizable), which means that the operations $\mathsf{write}()$ and $\mathsf{snapshot}()$ 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 $\mathsf{snapshot}()$, is such that $reg[k] = v$ if the closest preceding write operation issued by p_k is $\mathsf{write}(v)$. 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 $\mathsf{snapshot}()$ as it wants.

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

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 Section 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 $[\perp, \cdots, \perp]$.

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 $\mathsf{snapshot}()$ issued by p_i.

Algorithm implementing the operation $\mathsf{write}(v)$**: client side** This algorithm is described at lines 1-6, executed by the invoking process p_i (client), and lines 14-15, executed by all processes (in their server role).

When p_i invokes $\mathsf{write}(v)$, 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 $\mathsf{write}()$ at most once, and consequently, the values in $reg_i[k], reg(1)[k], \cdots, reg(m)[k]$ are all equal to $\perp$ if p_k has not yet invoked $\mathsf{write}()$, or belong to the set $\{\perp, v\}$ if p_k invoked $\mathsf{write}(v)$. After the update of $reg_i[1..n]$ is done, p_i returns from the operation.

Algorithm implementing the operation $\mathsf{write}(v)$**: 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 14). It then sends back to p_j the acknowledgment message WRITE_ACK(reg_i) (line 15). As seen above, if p_i knows writes not yet known by p_j, this message allows p_j to known them.

```
local variables initialization:
ssn_i ← 0; reg_i ← [⊥, ··· , ⊥] (⊥ is smaller than any value that can be written by a process).
%————————————————————————————————————————————————————————————————

operation write(v) is
(1)   reg_i[i] ← 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 ∈ {1, ··· , n} do reg_i[k] ← max(reg_i[k], reg(1)[k], ··· , reg(m)[k]) end for;
(6)   return()
end operation.

operation snapshot() is
(7)   repeat prev ← reg_i;
(8)          ssn_i ← ssn_i + 1; broadcast SNAPSHOT(reg_i, ssn_i);
(9)          wait (SNAPSHOT_ACK(reg, ssn_i) received from a majority of processes);
(10)         let reg(1), ..., reg(m) be the arrays received at the previous line;
(11)         for k ∈ {1, ··· , n} do reg_i[k] ← max(reg_i[k], reg(1)[k], ··· , reg(m)[k]) end for
(12)  until prev = reg_i end repeat;
(13)  return(reg_i)
end operation.
%————————————————————————————————————————————————————————————————

when a message WRITE(reg) is received from p_j do
(14)  for k ∈ {1, ··· n} do reg_i[k] ← max(reg_i[k], reg[k]) end for;
(15)  send WRITE_ACK(reg_i) to p_j.

when a message SNAPSHOT(reg, ssn) is received from p_j do
(16)  for k ∈ {1, ··· n} do reg_i[k] ← max(reg_i[k], reg[k]) end for;
(17)  send SNAPSHOT_ACK(reg_i, ssn) to p_j.
```

Figure 1: One-shot snapshot object in $\mathcal{CAMP}_{n,t}[t < n/2]$

Algorithm implementing the operation snapshot()**: client side** As previously, this algorithm is decomposed in two parts. The part described at lines 7-13 is executed by the invoking process p_i (client), while lines 16-17 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 9-11, 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 12), 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 16-17) 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.*

Proof Let us first consider the case where a correct process p_i invokes write(). It broadcasts a message WRITE() (line 2). As $t < n/2$, a majority of processes receive this message and send back an acknowledgment (line 15). Hence, p_i receives a message WRITE_ACK() from a majority of processes. It consequently cannot block forever at line 3, which proves the termination property for the write operation.

Let us now consider an invocation of snapshot() by a correct process p_i. Moreover, let τ be a time instant after which (a) no correct process invokes write(), (b) the messages generated by all the previous write operations and their acknowledgments have been received and processed, and (c) the faulty processes have crashed.

Hence, when after τ, p_i broadcasts a message SNAPSHOT$(-, sn)$, it eventually receives a message SNAPSHOT_ACK$(-, sn)$ from each correct process. It follows that $reg_i[1..n]$ eventually contains all the values known by all correct process. Let $\tau' \geq \tau$ be this time instant. After τ', reg_i can no longer be enriched with new values. It then follows that the predicate $prev = reg_i$ (line 12) becomes true (and remains true forever). When this occurs, p_i exits the repeat loop (if not already done). $\square_{Lemma\ 1}$

4.2 Definitions and notations

The following definitions are from [14]. 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 $invoc(\mathsf{op})$, which occurs when p_i invokes the operation, and a *response event*, denoted $resp(\mathsf{op})$, which occurs when p_i terminates the operation. The event $invoc(\mathsf{op})$ of an operation op occurs when it executes its first statement (line 1 or line 7), and its event $resp(\mathsf{op})$ (termination) occurs when it executes its return() statement (line 6 or line 13).

In addition to these events, sending and reception of messages create corresponding communication events [18]. 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 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

$\mathsf{snapshot}()$ operation returns an array reg such that $reg[i] = v$ if the invocation of $\mathsf{write}(v)$ 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 Lemma 2 and Lemma 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* $(reg1 \leq reg2) \;\vee\; (reg2 \leq reg1)$.

Proof Let $Q1$ (resp. $Q2$) be the majority quorum (set of processes) from which snap_1 (resp. snap_2) received messages SNAPSHOT_ACK() during its last execution of the "repeat"loop body (lines 8-11). As both Q_1 and Q_2 are majority quorums, there exists a process $p_i \in Q_1 \cap Q_2$. Let us consider the four following communication events.

- Events $\mathsf{rec1}$ and $\mathsf{send1}$: p_i receives reg_1 (event $\mathsf{rec1}$) which was sent by snap_1 at line 8 of its last loop iteration. From then on, we have $reg_i \geq reg1$ (because p_i receives WRITE($reg1, -$) from $\mathsf{snap1}$). Then p_i sends (event $\mathsf{send1}$) the corresponding SNAPSHOT_ACK($reg_i, -$) message (which –by assumption– is received by snap_1 as it makes p_i participate in the majority quorum $Q1$). Let us notice that $\mathsf{rec1} < \mathsf{send1}$.

- Events $\mathsf{rec2}$ and $\mathsf{send2}$ are the similar events as far as snap_2 is concerned. We have then $reg2 \leq reg_i$ when p_i sends to $\mathsf{snap2}$ the corresponding message SNAPSHOT_ACK($reg_i, -$) (which makes makes p_i participate in the majority quorum $Q2$). Moreover we also have $\mathsf{rec2} < \mathsf{send2}$.

As p_i belongs to $Q1 \cap Q2$ and is sequential, we have $\mathsf{rec1} < \mathsf{rec2}$ or $\mathsf{rec2} < \mathsf{rec1}$. Without loss of generality, let us assume $\mathsf{rec1} < \mathsf{rec2}$. It follows that p_i executed first $reg_i \leftarrow \mathsf{max}(reg_i, reg1)$ (event $\mathsf{rec1}$), later executed $reg_i \leftarrow \mathsf{max}(reg_i, reg2)$ (event $\mathsf{rec2}$), and finally sent $\text{SNAPSHOT_ACK}(reg_i, -)$ to $\mathsf{snap2}$. The message $\text{SNAPSHOT_ACK}(reg_i, -)$ sent by p_i to $\mathsf{snap2}$ is consequently such that $prev = reg_i \geq reg1$ (where $prev$ is the last value sent by $\mathsf{snap2}$ to p_i). Hence, as $prev = reg2$, we have $reg1 \leq reg2$, which concludes the proof of the lemma. $\square_{Lemma\ 5}$

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. $\big((\mathsf{ww1} \rightarrow \mathsf{ww2}) \wedge (reg[j] = v2)\big) \Rightarrow (reg[i] = v1)$.

Proof 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, such that $\mathsf{ww1} \rightarrow \mathsf{ww2}$. Let snap be a snapshot operation returning reg such that $reg[j] = v2$, and p_k the process that issued this snapshot.

Let S (resp. $W1$) be the majority quorum that allows snap (resp. $\mathsf{ww1}$) to terminate. As both S and $W1$ are majority quorums, there is a process $p_\ell \in S \cap W1$. Let τ be the time at which p_ℓ receives the message $\text{WRITE}(v1)$ from p_i. We have $\tau < resp(\mathsf{ww1})$ (this is because $\mathsf{ww1}$ terminates when p_i received a message $\text{ACK_WRITE}()$ (line 3) from all the processes in $W1$, which includes p_ℓ). There are two cases.

- Case 1: p_k sends its last message $\text{SNAPSHOT}()$ (line 3) to p_ℓ after time τ. In this case, reg_ℓ is such that $reg_\ell[i] = v1$, and consequently the message $\text{SNAPSHOT_ACK}()$ sent by p_ℓ to p_k carries $reg_\ell[i] = v1$. It follows that $reg_k[i]$ is set to $v1$ when p_k receives this message (and it processes it as $p_\ell \in S$).

- Case 2: p_k sends its last message $\text{SNAPSHOT}()$ (line 3) to p_ℓ before time τ. As $reg_k[j] = v2$ when p_k broadcasts $\text{SNAPSHOT}(reg_k, -)$, we necessarily have $invoc(\mathsf{ww2}) < \tau$. As $\tau < resp(\mathsf{ww1})$, it follows that $invoc(\mathsf{ww2}) < resp(\mathsf{ww1})$, which contradicts the initial assumption ($\mathsf{ww1} \rightarrow \mathsf{ww2}$), and concludes the proof of the lemma.

$\square_{Lemma\ 6}$

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.*

Proof $\widehat{H}$ being a history, let $\mathcal{S}$ be the set of its operations $\mathsf{snapshot}()$. It follows from Lemma 5, that there are at most $(n+1)$ different outputs for these snapshot operations. Let us extract ℓ sets from $\mathcal{S}$ ($1 \leq \ell \leq n+1$), denoted $\mathcal{S}_i$, partition of $\mathcal{S}$ and defined as follows: (1) for all $i \in [1..\ell]$: the snapshots of $\mathcal{S}_i$ have the same output denoted $reg(\mathcal{S}_i)$, and (2) if $i < j$: $reg(\mathcal{S}_i) < reg(\mathcal{S}_j)$. The proof is by induction.
Base case: $\mathcal{S}_1$. There are two sub-cases.

- If $reg(\mathcal{S}_1) = [\perp, \cdots, \perp]$, the snapshot operations of $\mathcal{S}_1$ are linearized in $\widehat{H'}$ in their invocation order. Moreover, in this case, let $\mu_1 = \mathsf{max}(\{invoc(\mathsf{snap}) \mid \mathsf{snap} \in \mathcal{S}_1\})$.

- Let us assume now that $reg(\mathcal{S}_1) \neq [\perp, \cdots, \perp]$. Hence there is at least one entry such that $reg(\mathcal{S}_1)[i] = v$. By Corollary 1, there is at least one process p_i that issued $\mathsf{write}(v)$ (concurrently or before any snapshot operation in $\mathcal{S}_1$). Let $\mathcal{W}_1$ be the set of the write operations such that $reg(\mathcal{S}_1)[x] \neq \perp$. These write operations are linearized in $\widehat{H'}$ at the time of their invocation events. Let τ_1 be the time at which occurs the last of them, i.e. $\tau_1 = \mathsf{max}(\{invoc(\mathsf{ww}) \mid \mathsf{ww} \in \mathcal{W}_1\})$.

Claim A: For each operation $\mathsf{snap} \in \mathcal{S}_1$ we have $\tau_1 < resp(\mathsf{snap})$.
Proof of Claim A. By contradiction. If the claim is false, there is $\mathsf{ww} = \mathsf{write}(v) \in \mathcal{W}_1$ such that $resp(\mathsf{snap}) < invoc(\mathsf{ww})$, from which we have $\mathsf{snap} \rightarrow \mathsf{ww}$. Let p_k the process that issued this write operation. By Lemma 3 we have then $reg(\mathcal{S}_1)[k] = \perp$, which contradicts the definition of $\mathcal{W}_1$. End of the proof of Claim A.

By Lemma 3 and Claim A, it follows that, for each write $\mathsf{ww} \in \mathcal{W}_1$, and for each $\mathsf{snap} \in \mathcal{S}_1$, we have $\mathsf{ww} < \mathsf{snap}$ or $\mathsf{ww} || \mathsf{snap}$, and $\tau_1 < resp(\mathsf{snap})$.[1]

In this case, let $\mu_1 = \mathsf{max}\big(\{invoc(\mathsf{ww}) \mid \mathsf{ww} \in \mathcal{W}_1\} \cup \{invoc(\mathsf{snap}) \mid \mathsf{snap} \in \mathcal{S}_1\}\big)$.

Second case: From $\mathcal{S}_1$ to $\mathcal{S}_2$.
Let snap be a snapshot operation in $\mathcal{S}_2$. There is a set of entries equal to $\perp$ in $reg(\mathcal{S}_1)$, which are no longer equal to $\perp$ in $reg(\mathcal{S}_2)$. By Corollary 1, there is a corresponding write for each of these entries. Let $\mathcal{W}_2$ be the set of these writes.
Claim B: For each write operation $\mathsf{ww} \in \mathcal{W}_2$ we have $\mu_1 < resp(\mathsf{ww})$.
Proof of Claim B. By contradiction. If the claim is false, let $\mathsf{ww} \in \mathcal{W}_2$ such that $resp(\mathsf{snap}) < invoc(\mathsf{ww})$ for some $\mathsf{snap} \in \mathcal{S}_1$. We have then $\mathsf{ww} < \mathsf{snap}$. Let p_k the process that issued ww. By Lemma 2, we must have $reg(\mathcal{S}_1)[k] \neq \perp$, which contradicts the definition of $\mathcal{W}_2$. End of the proof of claim B.
Thanks to Claim B, the write operations in $\mathcal{W}_2$ can be linearized in $\widehat{H'}$ after μ_1 ordered by their invocation times. Let τ_2 be the time at which occurs the last of them, i.e. $\tau_2 = \mathsf{max}(\mu_1, \mathsf{max}\{invoc(\mathsf{ww}) \mid \mathsf{ww} \in \mathcal{W}_2\})$.

Claim C: For every $\mathsf{snap} \in \mathcal{S}_2$, we have $\tau_2 < resp(\mathsf{snap})$.
Proof of Claim C. Following a proof similar to the one of Claim A, we have for each $\mathsf{snap} \in \mathcal{S}_2$: $\mathsf{max}(\{invoc(\mathsf{ww}) \mid \mathsf{ww} \in \mathcal{W}_2\}) < resp(\mathsf{snap})$. It then remains to show that, for each $\mathsf{snap} \in \mathcal{S}_2$, we have $\mu_1 < resp(\mathsf{snap})$. Considering, this is false for some snap, there are two cases (due to the definition of μ_1).

- $resp(\mathsf{snap})$ occurs before the invocation of some write. Let $\mathsf{ww} \in \mathcal{W}_1$ be a write such that $resp(\mathsf{snap}) < invoc(\mathsf{ww})$. Hence we have $\mathsf{snap} \rightarrow \mathsf{ww}$. Let p_k be the process that issued this write. By Lemma 3, we have $reg(\mathcal{S}_2)[k] = \perp$, which contradicts the definition of $\mathcal{W}_1$.

- $resp(\mathsf{snap})$ occurs before the invocation of some snapshot $\mathsf{snap}_1 \in \mathcal{S}_1$: $resp(\mathsf{snap}) < invoc(\mathsf{snap}_1)$. Hence, $\mathsf{snap} < \mathsf{snap}_1$. Due to Lemma 4 $reg(\mathcal{S}_2) \leq reg(\mathcal{S}_1)$, which contradicts the definition of $\mathcal{S}_2$. End of the proof of Claim C.

Thanks to Claim C, the scan operations in $\mathcal{S}_2$ can be linearized in $\widehat{H'}$ after τ_2 ordered by their invocation times. Let $\mu_2 = \mathsf{max}\big(\{invoc(\mathsf{ww}) \mid \mathsf{ww} \in \mathcal{W}_1 \cup \mathcal{W}_2\} \cup \{invoc(\mathsf{snap}) \mid \mathsf{snap} \in \mathcal{S}_1 \cup \mathcal{S}_2\}\big)$.

Next cases: from $\mathcal{S}_3$ until $\mathcal{S}_\ell$. Their proof is the same as the one for the case $\mathcal{S}_2$.
Let $\mu_\ell = \mathsf{max}\big(\{invoc(\mathsf{ww}) \mid \mathsf{ww} \in \mathcal{W}_1 \cup \cdots \cup \mathcal{W}_\ell\} \cup \{invoc(\mathsf{snap}) \mid \mathsf{snap} \in \mathcal{S}_1 \cup \cdots \cup \mathcal{S}_\ell\}\big)$.

Let us finally the operations that happen in the history $\widehat{H}$ and are not yet linearized. As all the snapshot operations and the write operations from which the snapshot operations obtain their values, have been linearized, the not yet linearized operations (if any) are the write operations whose values are never output by a snapshot operation. Let $\mathcal{W}$ the set of these write operations. There are two cases.

- $|\mathcal{S}_\ell|$ is infinite.
  Let $\mathsf{ww} \in \mathcal{W}$. As $|\mathcal{S}_l|$ is infinite, there exists a snapshot snap such that $\mathsf{ww} \rightarrow \mathsf{snap}$. By Lemma 2

[1] It is assumed that the time line is the line of real numbers, namely, between any two time instants, there is another time instant.

the value written by ww appears in $reg(\mathcal{S}_\ell)$. A contradiction, which proves that this case cannot occur.

- $|\mathcal{S}_l|$ is finite.
  Claim D: For each write operation ww $\in \mathcal{W}$: $\mu_\ell < resp(\mathsf{ww})$.
  Proof of Claim D. The proof is by contradiction. Let us assume that the claim is false. Let p_k be the process that issued ww. Then for all i $reg(\mathcal{S}_i)[k] = \perp$. There are two cases.
  - ww terminates before the invocation of some write operation of $\mathcal{W}_1 \cup \mathcal{W}_2 \cup \ldots \cup \mathcal{W}_\ell$. Let $\mathsf{ww}_1 \in \mathcal{W}_i$ such that $resp(\mathsf{ww}) < invoc(\mathsf{ww}_1)$. Hence, ww $\rightarrow \mathsf{ww}_1$. By Lemma 6, we have $reg(\mathcal{S}_i)[k] \neq \perp$, a contradiction.
  - ww terminates before the invocation of some snapshot operation in $\mathcal{S}_\ell$. Let snap $\in \mathcal{S}_\ell$ such that $resp(\mathsf{ww}) < invoc(\mathsf{snap})$. Hence, ww $\rightarrow$ snap. It then follows from Lemma 2 that $reg(\mathcal{S}_\ell)[k] \neq \perp$, a contradiction, which concludes the proof of Claim D.

The write operations of $\mathcal{W}$ are linearized in $\widehat{H'}$ after τ_ℓ in their invocation time order.

If follows from the previous construction rules of $\widehat{H'}$ that this sequential history respects the sequential specification of the one-shot snapshot object. $\square_{Lemma\ 7}$

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). $\square_{Theorem\ 1}$

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).

5.1 A non-blocking algorithm

It is easy to extend the basic algorithm depicted in Figure 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.

5.2 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 Figure 2. To ensure this strong termination property, two mechanisms are added to the basic algorithm.

```
local variables initialization:
    snw_i ← 0; sns_i ← 0; reg_i ← [⊥, ··· , ⊥]; for each i, j : repSnap[i, j] = ⊥.
---------------------------------------------------------------------------
operation write(v) is
(1)   snw_i ← snw_i + 1; write_pending ← (v, snw_i);
(2)   wait(write_pending = ⊥); return()
end operation.

operation snapshot() is
(3)   sns_i ← sns_i + 1; Rbroadcast SNAP(p_i, sns_i);
(4)   wait(repSnap[i, sns_i] ≠ ⊥); return(repSnap[i, sns_i])
end operation.
---------------------------------------------------------------------------
function base_write(wp) is
(5)   reg_i[i] ← wp;
(6)   broadcast WRITE(reg_i,wp);
(7)   wait (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 ∈ {1, ··· , n} do reg_i[k] ← max_≺sn {r[k]|r ∈ R ∪ reg_i} end for;
(10)  return()
end function.

function base_snapshot(s, t) is
(11)  while repSnap[s, t] = ⊥ do
(12)      prev ← reg_i; ssn_i ← ssn_i + 1; broadcast SNAPSHOT(s, t, reg_i, ssn_i);
(13)      wait until (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 ∈ {1, ··· , n} do reg_i[k] ← max_≺sn {r[k] | r ∈ R ∪ 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.
---------------------------------------------------------------------------
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.
```

Figure 2: Multi-shot snapshot object in $\mathcal{CAMP}_{n,t}[t < n/2]$ (code for p_i)

(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.

(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 Figure 2, 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: $\big((v,a) \prec_{sn} (w,b)\big) \Leftrightarrow (a < b)$.

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(),[2] 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 (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 [3].

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 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.

[2]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. [7, 20, 24]).

[3]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.

	Stacking [1] on [4]	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

Table 1: Cost comparison in favorable cases

6 Conclusion

Since a long time, snapshot algorithms suited to asynchronous message-passing reliable systems have been proposed (e.g. in [8, 12, 17]). These algorithms, which consider process local states and channels states, do not cope with failures, and provides snapshots which cannot always be compared [9, 25].

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 [4]. 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).

Acknowledgments

This work has been partially supported by the French ANR project DISPLEXITY, which is devoted to computability and complexity in distributed computing.

References

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

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

[3] Attiya H., Efficient and robust sharing of memory in message-passing systems. *Journal of Algorithms*, 34:109-127 (2000)

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

[5] Attiya H., Guerraoui R. and Ruppert E., Partial snapshot objects. *Proc. 20th ACM Symposium on Parallel Architectures and Algorithms (SPAA'08)*, ACM Press, pp. 336-343 (2008)

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

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

[8] Chandy K.M. and Lamport L., Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63-75 (1985)

[9] Cooper R. and Marzullo K., Consistent detection of global predicates. *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, ACM press, pp. 163–173 (1991)

[10] Dijkstra E.W.D., and Scholten C.S., Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1-4 (1980)

[11] Dutta P., Guerraoui R., Levy R., and Vukolic M., Fast access to distributed atomic memory. *SIAM Journal of Computing*, 39(8):3752-3783 (2010)

[12] Hélary J.-M., Mostéfaoui A., and Raynal M., Communication-induced determination of consistent snapshots. *IEEE Transactions on Parallel Distributed Systems*, 10(9):865-877 (1999)

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

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

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

[16] Inoue M., Chen W., Masuzawa T., and Tokura N., Linear time snapshot using multi-reader multi-writer registers. *Proc. 8th Int'l Workshop on Distributed Algorithms (WDAG'94)*, Springer LNCS 1857, pp. 130-140 (1994)

[17] Lai T. H. and Yang T. H., On distributed snapshots. *Information Processing Letters*, 25:153-15, (1987)

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

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

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

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

[22] Mostéfaoui A. and Raynal M., Time-efficient read/write register in crash-prone asynchronous message-passing systems. *Tech Report 2034*, IRISA, Université de Rennes (F), (2016) https://hal.inria.fr/hal-01256067, To appear in *Proc. 4th Int'l Conference on Networked Systems (NETYS'16)*, Springer LNCS (2016)

[23] Mostéfaoui A. and Raynal M., Two-bit messages are sufficient to implement atomic read/write registers in crash-prone systems. *Tech Report 2031*, IRISA, Université de Rennes (F), https://hal.inria.fr/hal-01271135, To appear in *Proc. 35th Int'l ACM Symposium on Principles of Distributed Computing (PODC'16)*, ACM Press (2016)

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

[25] Raynal M., *Distributed algorithms for message-passing systems*. Springer, 510 pages, ISBN 978-3-642-38122-5 (2013)

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

[27] Ruppert E., Implementing shared registers in asynchronous message-passing systems. *Springer Encyclopedia of Algorithms*, pp. 400-403 (2008)

[28] Taubenfeld G., Synchronization algorithms and concurrent programming. *Pearson Prentice-Hall*, 423 pages, ISBN 0-131-97259-6 (2006)