A Simple Snapshot Algorithm for Multicore Systems

Damien Imbs, Michel Raynal

Publications Internes de l'IRISA
ISSN : 2102-6327
PI 1955 – juillet 2010

A Simple Snapshot Algorithm for Multicore Systems

Damien Imbs[*] , Michel Raynal[**]
damien.imbs@irisa.fr, raynal@irisa.fr

Abstract: An atomic snapshot object is an object that can be concurrently accessed by n asynchronous processes prone to crash. It is made of m components (base atomic registers) and is defined by two operations: an update operation that allows a process to atomically assign a new value to a component and a snapshot operation that atomically reads and returns the values of all the components. To cope with the net effect of concurrency, asynchrony and failures, the algorithm implementing the update operation has to help concurrent snapshot operations in order they can always terminate.

This paper presents a new and particularly simple construction of a snapshot object. This construction relies on a new principle, that we call "write first, help later" strategy. This strategy directs an update operation first to write its value and only then computes an helping snapshot value that can be used by a snapshot operation in order to terminate. Interestingly, not only the algorithms implementing the snapshot and update operations are simple and have easy proofs, but they are also efficient in terms of the number of accesses to the underlying atomic registers shared by the processes. An operation costs $O(m)$ in the best case and $O(n \times m)$ in the worst case.

Key-words: Asynchronous shared memory system, Atomicity, Atomic snapshot, Concurrency, Linearizability, Atomic registers, Process crash, Read/Write atomic register, Wait-freedom.

Un algorithme de snapshot simple pour les systèmes multi-cœurs

Résumé : *Ce rapport présente un algorithme de snapshot simple. Contrairement aux algorithmes proposés précédemment, il utilise le principe suivant: pour effectuer une mise à jour d'un élément du snapshot, les processus écrivent d'abord en mémoire partagée la valeur qu'ils veulent affecter à cet élément puis ils calculent une valeur d'aide.*

Mots clés : *Atomicité, Snapshot atomique, Concurrence, Linéarisabilité, Registres atomiques, Défaillances par crash, Mémoire partagée, Sans-attente.*

---

[*] Projet ASAP: équipe commune avec l'INRIA, le CNRS, l'université Rennes 1 et l'INSA de Rennes
[**] Projet ASAP: équipe commune avec l'INRIA, le CNRS, l'université Rennes 1 et l'INSA de Rennes

1 Introduction

Shared memory snapshot objects As noticed in [1], obtaining a consistent picture of the system global state from partial observations that are made asynchronously while the system evolves, and despite possible process crashes, is a fundamental problem of distributed and concurrent computing. Solving this problem, that has been abstracted under the name *snapshot problem*, requires to be able to cope simultaneously with the "adversaries" that are concurrency, asynchrony and failures.

More precisely, considering a shared memory system made up of base atomic read/write registers, that can be concurrently accessed by asynchronous processes prone to crash, a snapshot object is an object that (1) consists of m components (each component being a base atomic register that can contain any value), and (2) provides the processes with two operations, denoted update() and snapshot(). The update() operation allows the invoking process to atomically store a new value in an individual component. Differently, the snapshot() operation returns the values of all the components as if they had been read simultaneously.

From an execution point of view, a snapshot object has to satisfy the safety property called *linearizability*: the update and snapshot operations have to appear as if they had been executed one after the other, each being instantaneously executed at some point of the time line comprised between its start event and its end event [13]. From a liveness point of view, each update or snapshot operation has to terminate if the invoking process does not crash. This liveness property is called *wait-freedom* [10]. It means that an operation issued by a correct process has to terminate whatever the concurrency pattern and the behavior of the other processes (the fact that some processes crash or are very slow cannot prevent an operation from terminating, as long as the issuing process does not crash). Wait-freedom is starvation-freedom despite asynchrony and process failures. In order to implement the wait-freedom property, a process that issues an update() operation can be required to help terminate the processes that have concurrently issued a snapshot() operation (preventing them from looping forever). This helping mechanism is required to ensure that all snapshot() operations (issued by processes that do not crash) do always terminate [1].

The snapshot abstraction The snapshot object has proved to be a very useful abstraction for solving many other problems in asynchronous shared memory systems prone to process crashes, such as approximate agreement, randomized consensus, concurrent data structures, etc. A snapshot object hides the "implementation details" that are difficult to cope with in presence of the net effect of concurrency, asynchrony and failures. It is important to notice that, from a computational point of view, a snapshot object is not more powerful than the base atomic read/write objects it is built from. It only provides a higher abstraction level.

Shared memory snapshot vs message-passing snapshot The values returned by a snapshot() operation is a value of the part of the shared memory that is encapsulated in the corresponding snapshot object. It follows from the linearizability property satisfied by a snapshot object that there is a time instant at which the values returned by a snapshot() operation were simultaneously present in the shared memory, this time instant belonging to the time interval associated with that snapshot() operation.

The previous observation is in contrast with the notion of *distributed snapshot* used to capture consistent global states in asynchronous message-passing systems [7] where two distributed snapshots obtained by two processes can be consistent but incomparable in the sense that they cannot be linearized. The set of all the distributed snapshots that can be obtained from a message-passing distributed execution has only a lattice structure (basically, they can be partially ordered but not totally ordered). In that sense, the abstraction level provided by a shared memory snapshot object is a higher abstraction level than the one offered by message-passing distributed snapshots. They hide more asynchrony.

Types of snapshot objects As far as base atomic registers are concerned, two types of snapshot objects have been investigated: single-writer and multi-writer snapshot objects. A *single-writer snapshot* object has one component per process, and the component associated with a process can be written only by that process. The number of components (m) is then the same as the number of processes (n). The base registers from which a single-writer snapshot object is built are then single-writer/multi-reader atomic registers. Wait-free algorithms implementing single-writer snapshot objects for n processes are described in [1]. Their costs is $O(n^2)$ (when counting the number of shared memory accesses). An algorithm whose cost is $O(n \log\ n)$ is described in [5]. An implementation suited to systems with a possibly infinite number of processes (but where finitely many processes can take steps in each finite time interval) is described in [2].

A *multi-writer snapshot* object is a snapshot object of which each component can be written by any process. So, the base read/write registers on which its implementation relies are multi-writer/multi-reader atomic registers. Wait-free algorithms implementing multi-writer snapshot objects made up of m base components are described in [3, 15, 16]. The algorithm described in [15] has a linear cost $O(n)$. A short survey of algorithms that implement single-writer and multi-writer snapshot objects is presented in [8].

The notion of *partial* snapshot has been introduced in [4]. In that case, every snapshot operation specifies the specific set of components it wants to read (if this set always includes all the components, we have the classical snapshot operation). An efficient partial snapshot algorithm is described in [14].

Content of the paper This paper presents a novel *multi-writer snapshot algorithm* for asynchronous systems (hence, processes communicate through multi-writer/multi-reader atomic registers). This wait-free algorithm has two noteworthy properties. The first lies in its simplicity, and simplicity is a first class property (to be useful and manageable, algorithms have to be clearly understood).

The second important feature of the proposed algorithm lies in its design principle that allows the update operation to be more asynchronous and to require smaller size atomic registers than previous snapshot object implementations. We call this design principle "write first, help later". Previous snapshot implementations use one base atomic register $REG[r]$ per component r. These registers have to be large. They are made up of several fields, including a field for the last value written into the component r, a field storing a snapshot value, and a few other fields containing control data. A *snapshot value* is an array with one value per component. More precisely, every $\mathsf{update}(r, v)$ operation atomically writes into $REG[r]$ both the new value v and a snapshot value it has computed. That snapshot value can then be used by concurrent snapshot operations in order to terminate (this is called an *helping mechanism*). From an efficiency point of view, the atomic write into $REG[r]$ issued by an update operation can be expensive.

Differently, thanks to the "write first, help later" strategy, the proposed $\mathsf{update}(r, v)$ algorithm separates the write of the value v into $REG[r]$ and the write of an helping snapshot value. The fact that an $\mathsf{update}(r, v)$ operation first writes v and only later (and asynchronously) writes an helping snapshot value, (1) allows a snapshot operation to obtain a value for the component r that is at least as recent as v, and (2) allows the use of a single independent helping atomic register per process.

Motivation This work was motivated by multicore architectures. The challenging advent of these architectures and the fact that (albeit it can be rare) a core may crash make the investigation of wait-free algorithms more and more challenging. This, that has been called the "multicore revolution" [12], calls for a new approach for multicore synchronization [11, 18, 19]. Moreover, a snapshot object is a fundamental object as it can be used to encapsulate the data that define the critical part of the system state. Wait-free snapshot algorithms are really needed.

Roadmap The paper is made up of 5 sections. Section 2 presents the computation model and defines the snapshot object type. Section 3 presents the new snapshot algorithm. Section 4 proves its correctness and analyses its cost. Finally Section 5 concludes the paper.

2 Computation model and the Snapshot problem

2.1 Underlying shared memory model

The system is made up of n processes $p_1, \ldots, p_n$. The identity of p_i is its index i. These processes communicate through multi-writer/multi-reader atomic registers. Atomic means that each read or write operation on a register appears as if it has been executed sequentially at some point of the time line comprised between its start and end event. The registers are assumed to be reliable (this assumption is without loss of generality -from a computability point of view- as it is possible to build atomic reliable registers on top of crash prone atomic registers [6, 9, 11, 17]).

There is no assumption on the speed of processes: they are asynchronous. Moreover, up to $(n - 1)$ processes may crash. Before it crashes (if it ever crashes), a process executes correctly its algorithm. A crash is a premature halt: after it has crashed, a process executes no more step. Given a run, a process that does not crash is *correct* in that run, otherwise it is *faulty* in that run.

2.2 Snapshot object

Object operations As already said in the Introduction, a multi-writer snapshot object is made up of m components (each being a multi-writer/multi-reader atomic register) that provides the processes with two operations denoted $\mathsf{update}()$ and $\mathsf{snapshot}()$.

- $\mathsf{update}(r, v)$ allows the invoking process to write a value v in component r ($1 \leq r \leq m$) of the snapshot object. That operation returns the control value ok.
- $\mathsf{snapshot}()$ allows the invoking process to obtain the value of each component. Hence, it returns an array of m values, one for each component.

Snapshot object definition The behavior of a snapshot object is defined by the following properties.

- Termination. Every invocation of $\mathsf{update}()$ or $\mathsf{snapshot}()$ issued by a correct process terminates.
- Consistency. The operations issued by the processes (except possibly the last operation issued by a process if it is faulty[1]) appear as if they have been executed one after the other, each one being executed at some point of the time line between its start event and its end event.

The termination property is wait-freedom [10] (i.e., starvation-freedom despite asynchrony, concurrency and process crashes). The consistency property is linearizability [13] (here, it means that a $\mathsf{snapshot}()$ operation always returns component values that were simultaneously present in the shared memory at some time instant, and are consistent with the update operations that precede it).

[1] If such an operation does not appear in the sequence, it is as if it has not been invoked.

An additional property This paper considers the following additional property called *freshness* (that we have introduced in [14]). A snapshot object implementation satisfies the *freshness* property if, when a snapshot operation snapshot() is helped by an update(r, v) operation, the value returned for the component r is at least as recent as v.

The aim of this property is to provide every snapshot operation with an array of values "as fresh as possible". As noticed in the Introduction, (to our knowledge) no snapshot object implementation proposed so far (but [14]) satisfies this property. As we will see, this property is an immediate consequence of the "write first, help later" strategy.

3 A simple snapshot algorithm

3.1 Underlying shared objects

The algorithms implementing the update and snapshot operations use two arrays of atomic registers. Let us recall that m is the number of components of the snapshot object, and n is the number of processes.

- The first, denoted $REG[1..m]$, is made up of multi-writer/multi-reader atomic registers. Register $REG[r]$ is associated with component r. It has three fields $\langle value, pid, sn\rangle$ whose meaning is the following. $REG[r].value$ contains the current value of the component r, while $REG[r].(pid, sn)$ is the "identity" of v. $REG[r].pid$ is the index of the process that issued the corresponding update(r, v) operation, while $REG[r].sn$ is the sequence number of this update when considering all updates issued by p_{pid}.
- The second array, denoted $HELPSNAP[1..n]$ is made up of one one-writer/multi-reader atomic register per process. $HELPSNAP[i]$ is written only by p_i and contains a snapshot of $REG[1..m]$ computed by p_i during its last update() invocation. This snapshot value is destined to help processes that issued snapshot() invocations concurrent with p_i's update. More precisely, if during its invocation of snapshot() a process p_j discovers that it can be helped by p_i, it returns the value currently kept in $HELPSNAP[i]$ as output of its own snapshot invocation.

From a notation point of view, the previous shared objects are denoted with uppercase letters. On the contrary, variables local to a process p_i are denoted with lowercase letters (process index i is sometimes used as a subscript for p_i's local variables). The local variables are introduced in the algorithm description.

3.2 The snapshot() operation

Algorithm 1 defines the code executed by a process p_i when it invokes snapshot(). The read of the array $HELPSNAP[1..n]$ (at line 02 or line 04) is called a *scan*. A scan is asynchronous and not atomic (the array is read in any order and at any speed, only the reading of each entry is atomic). As in [1], process p_i first uses a "sequential double scan" to try compute a snapshot value by itself. If it cannot terminate by itself, it looks for a process p_w which can help it. As we will see, this occurs when p_i observes that there is a process p_w that issued two update invocations while it (p_i) is still executing its snapshot invocation.

```
operation snapshot():    % (code for p_i) %
(01)  can_help_i ← ∅;
(02)  for each r ∈ {1, ··· , m} do aa[r] ← REG[r] end for;
(03)  while (true) do
(04)      for each r ∈ {1, ··· , m} do bb[r] ← REG[r] end for;
(05)      if (∀r ∈ {1, ··· , m} : aa[r] = bb[r]) then return(bb[1].value, ··· , bb[m].value) end if;
(06)      for each r ∈ {1, ··· , m} such that bb[r] ≠ aa[r] do
(07)          can_help_i ← can_help_i ∪ {⟨w, sn⟩} where ⟨−, w, sn⟩ = bb[r]
(08)      end for;
(09)      if (∃⟨w, sn1⟩, ⟨w, sn2⟩ ∈ can_help_i such that sn1 ≠ sn2) then
(10)          return(HELPSNAP[w])
(11)      end if;
(12)      aa ← bb
(13)  end while.
```

Algorithm 1: snapshot() operation (code for p_i)

Try to terminate without help: successful double scan A process p_i first scans $HELPSNAP$ twice (line 02 and line 04). The important point here is that, when considering any two scans issued by a process, the second one always starts after the first one has terminated (scans issued by a process are sequential). The values obtained from the first scan are saved in the local array aa, while the values obtained from the second scan are saved in the local array bb.

If the local predicate $\forall r : aa[r] = bb[r]$ is true, p_i has obtained the same values in both scans. This is called a *successful double scan*. This means that $REG[1..m]$ was containing these values at any time during the period starting at the end of the first scan and finishing at the beginning of the second scan. Consequently, p_i returns the array of values $bb[1..m].value$ as the result of its snapshot invocation (line 05).

Otherwise, record the processes that can help If the predicate $\forall r : aa[r] = bb[r]$ is false, p_i looks for all entries r that have been modified during its previous double scan. Those are the entries r such that $aa[r] \neq bb[r]$. Let r be such an entry. As witnessed by $bb[r] = \langle -, w, sn\rangle$, the component r has been modified by p_w. Process p_i consequently adds the pair $\langle w, sn\rangle$ to the set can_help_i (line 07). Hence, can_help_i (that is initialized to $\emptyset$, line 01) contains pairs $\langle x, y\rangle$ indicating that process p_x has issued its yth update while p_i is executing its snapshot operation.

Then, p_i tests the helping predicate (line 09, see below). If this predicate is false, p_i moves the array bb into the array aa (line 12) and enters again the **while** loop. As we can see, the lines 12 and 04 constitute a new double scan.

Finally, try to benefit from the helping mechanism The predicate $\exists\langle w, sn1\rangle, \langle w, sn2\rangle \in can\_help_i$ such that $sn1 \neq sn2$ is the helping predicate. It means that there is a process p_w that issued two updates that are concurrent with p_i's snapshot invocation. As we are about to see (line 02 of the code of operation update(r, v) in Algorithm 2 and Figure 1), this means that p_w has issued a snapshot embedded in an update concurrent with p_i's snapshot invocation. The corresponding snapshot value, that has been saved in $HELPSNAP[i]$, can then be returned by p_i as output of its snapshot invocation (line 10).

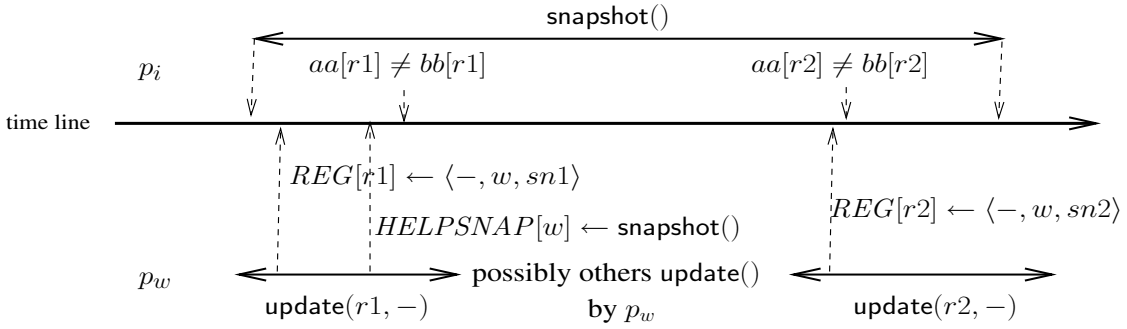

Figure 1: A snapshot with two concurrent updates by the same process

3.3 The update() operation

Algorithm 2 defines the code executed by a process when it invokes update(r, v). First, p_i increases the local sequence number generator sn_i (initialized to 0) and writes atomically the triple $\langle v, i, sn_i\rangle$ into $REG[r]$ (line 01). Then, (asynchronously) it computes a snapshot value and writes into $HELPSNAP[i]$ (line 02). This constitutes the "write first, help later" strategy. The way $HELPSNAP[i]$ can be used by other processes has been described previously. Finally, p_i returns from its update() invocation (line 03).

```
operation update(r, v):    % (code for p_i) %
(01) sn_i ← sn_i + 1; REG[r] ← ⟨v, i, sn_i⟩;
(02) HELPSNAP[i] ← snapshot();
(03) return(ok).
```

Algorithm 2: update(r, v) operation (code for p_i)

3.4 On the "write first, help later" strategy

As we can see, this strategy is very simple. It has three main advantages.

- It first allows atomic write operations (at lines 01 and 02 of algorithm 2) to write values into the base atomic registers $REG[r]$ and $HELPSNAP[i]$ that have a smaller size than the values written in traditional snapshot algorithms such as the ones described in [1]. These algorithms write atomically v, i, sn_i and a snapshot value into $REG[r]$. Implementing atomic writes of smaller values allows for more efficient solutions.

- Second, this simple strategy allows the atomic writes into the base atomic registers $REG[r]$ and $HELPSNAP[i]$ not to be synchronized (while they are strongly synchronized in additional snapshot algorithms as they are pieced into a single atomic write).
- Finally, as we will see later, the "write first, help later" strategy allows snapshot operations to return component values "as fresh as possible" (this notion will be precisely defined in the next section).

4 Proof of the algorithm

4.1 Preliminary definitions

Definition 1 *The array of values $[v_1, \ldots, v_m]$ returned by a* snapshot() *operation is* well-defined *if, for each r, $1 \leq r \leq m$, the value v_r has been read from $REG[r]$.*

Definition 2 *The values returned by a* snapshot() *operation are* mutually consistent *if there is a time at which they were simultaneously present in the snapshot object.*

Definition 3 *The values returned by a* snapshot() *operation are* fresh *if, for each r, $1 \leq r \leq m$, the value v_r returned for component r is not older than the last value written into $REG[r]$ before the snapshot invocation*[2].

Definition 4 *Let* snp *be a snapshot operation issued by a process p_i.*

- snp *is* 0-helped *if it terminates with a successful double scan (line 05 of algorithm 1).*
- snp *is* 1-helped *if it terminates by returning $HELPSNAP[w1]$ (line 10 of Algorithm 1), and the values in $HELPSNAP[w1]$ come from a successful double scan by p_{w1} (i.e., the values in $HELPSNAP[w1]$ have been computed at line 02 of Algorithm 2 by the snapshot operation embedded within an* update() *operation issued by p_{w1}).*
- snp *is* 2-helped *if it terminates by returning $HELPSNAP[w1]$ (line 10 of Algorithm 1), and the values in $HELPSNAP[w1]$ come from a 1-helped* snapshot() *operation invoked by a process p_{w2} at line 02 of Algorithm 2.*
- *For the next values of h, the h-*helped *notion is similarly defined by induction.*

4.2 The values returned are well-defined, mutually consistent and fresh

Lemma 1 *The values returned by a 0-helped* snapshot() *operation are well-defined, mutually consistent and fresh.*

Proof Let us consider a 0-helped snapshot() operation. As it terminates at line 05 of algorithm 1, the array of values returned are the values in the array $bb[1..m]$. It follows from line 04 that this array is well-defined (its value are from the array $REG[1..m]$).

Mutual consistency and freshness follow from the fact that the termination of the snapshot() operation is due to a successful double scan. More precisely, the values kept in $bb[1..m]$ were present simultaneously in $REG[1..m]$ during the period starting at the end of the first scan and ending at the beginning of the second scan (mutual consistency) and, for any entry r, the value returned from $REG[r]$ is not older that the value kept in $REG[r]$ when the snapshot operation started (freshness). $\square_{Lemma\ 1}$

Lemma 2 *The values returned by a 1-helped* snapshot() *operation are well-defined, mutually consistent and fresh.*

Proof Let snp be a 1-helped snapshot() operation. It follows from the definition of "1-helped snapshot() operation" that (a) the array returned by snp (namely the value read by p_i from $HELPSNAP[w1]$) has been computed by a snapshot operation snp$'$ embedded in an update operation issued by p_{w1}, and (b) snp$'$ is 0-helped (i.e., $HELPSNAP[w1]$ comes from a successful double scan). It then follows from Lemma 1 that the values in $HELPSNAP[w1]$ are well-defined and mutually consistent.

Let us now show that the values contained in $HELPSNAP[w1]$ are fresh, i.e., for each r, the value in $HELPSNAP[w1][r]$ is not older than the last value written into $REG[r]$ before snp started. Let up1 and up2 be the two updates issued by p_{w1} whose sequence numbers are $sn1$ and $sn2$ (let us observe that, due to the test of line 09, these updates do exist). Moreover, snp$'$ is embedded inside an update up$'$ from p_{w1} whose sequence number sn is such that $sn1 \leq sn \leq sn2$.

As the value of $HELPSNAP[w1]$ that is returned is computed during up$'$ and up$'$ has been invoked by p_{w1} after up1, the values obtained from REG and assigned to $HELPSNAP[w1]$, have been read after up1 started. We claim that up1 started after snp. It follows from that claim that the values in $HELPSNAP[w1]$ that are returned have been read from REG after snp started, which proves the lemma.

[2] Let us recall that, as each $REG[r]$ is an atomic register, its read and write operations can be totally ordered in a consistent way. The word "last" is used with respect to this total order.

Proof of the claim. Let $r1$ be the component that entailed the addition of the pair $\langle w1, sn1\rangle$ to can_help_i at line 07 (Algorithm 1) during the execution of snp. Moreover, let $aa1[r1]$ and $bb1[r1]$ be the values in the local arrays aa and bb that entailed this addition. Hence, we have $aa1[r1] \neq bb1[r1]$ (line 07, Algorithm 1) and $bb1[r1] = \langle -, w1, sn1\rangle$. As snp has read $REG[r1]$ twice (first to obtain $aa1[r1]$, and then to obtain $bb1[r1]$) and $aa1[r1] \neq bb1[r1]$, it follows that up1 started after the start of snp, which concludes the proof of the claim. End of proof of the claim. $\square_{Lemma\ 2}$

Lemma 3 *For any h, the values returned by a h-helped* snapshot() *operation are well-defined, mutually consistent and fresh.*

Proof The proof is by induction. The base case is $h = 0$ (Lemma 1). Assuming that the lemma is satisfied up to $h - 1$, the proof for h is similar to proof done for $h = 1$ that relies on the fact that we have a proof for the case $h - 1 = 0$ (Lemma 2). $\square_{Lemma\ 3}$

4.3 Wait-freedom

The next lemma shows that the algorithms that construct a snapshot object are wait-free, i.e., terminate despite the crash of any number of processes [10].

Lemma 4 *Any* update() *and* snapshot() *operation issued by a correct process terminates.*

Proof Let us first observe that if every snapshot() operation issued by a correct process terminates, then all its update() operations terminate. Hence, the proof only has to show that all snapshot() operations issued by a correct process terminate.

Let us consider a snapshot() operation issued by a correct process p_i. If, when p_i executes line 05, the predicate is true, the snapshot() operation terminates. So, we have to show that, if the predicate of line 05 is never satisfied, then the predicate of line 09 eventually becomes true. As the predicate of line 05 is never satisfied, each time p_i executes the loop body, there is a component r such that $aa[r] \neq bb[r]$. The process p_k that has modified $REG[k]$ between the two readings by p_i entails the addition of the pair $< k, snk >$ to can_help_i (where $< k, snk >$ is extracted from $bb[r]$). In the worst case, $n-1$ pairs (one associated with each process, but p_i because it cannot execute an update operation while it executes a snapshot operation) can be added to can_help_i while the predicate of line 09 remains false. But once can_help_i contains one pair per process (but p_i), the next pair that is added is necessarily due to a process p_w such that can_help_i already contains a pair $< w, sn1 >$. Consequently, after line 07 has been executed due to that process p_w, a second pair $< w, sn2 >$ is added to can_help_i. Then, the test of line 09 becomes satisfied, which proves the lemma. $\square_{Lemma\ 4}$

4.4 Linearizability

As already stated, the consistency property of a snapshot object is linearizability. This means that all the update() and snapshot() operations issued by the processes during a run (except possibly the last operation issued by faulty processes), have to appear as if they have been executed one after the other, each one being executed at some point of the time line between its start event and its end event.

Lemma 5 *Every run of a partial snapshot object whose* update() *and* snapshot() *are implemented with algorithms 1 and 2, is linearizable.*

Proof The proof consists in associating with each update or snapshot operation op a single point of the time line, denoted $\ell p(\mathsf{op})$ and called *linearization point* of op, such that:

- $\ell p(\mathsf{op})$ lies between the beginning (start event) of op and its end (end event),
- No two operations have the same linearization point,
- The sequence of the operations defined by their linearization points is a sequential execution of the snapshot object.

So the proof consists in (a) an appropriate definition of the linearization points and (b) showing that the associated sequence satisfies the specification of the snapshot object (as defined in Section 2.2).

- Point (a): definition of the linearization points.

  The linearization point of each operation (except possibly the last operation of faulty processes) is defined as follows.

  - The linearization point of an operation update$(r, -)$ is the time at which its embedded write of $REG[r]$ occurs (line 01, Algorithm 2)[3].
  - The linearization point of a snapshot operation psp = snapshot() depends on the line of Algorithm 1 at which its return() statement is executed.

[3]Let us recall that the underlying write and read operations accessing the shared registers are atomic and, consequently, their occurrence times, i.e., their own linearization points, are well-defined.

* Case 1: psp returns at line 05 due a successful double scan (i.e., psp is 0-helped). Its linearization point is any point of the time line between the first scan and the second scan of that successful double scan.
* Case 2: psp returns at line 10 (i.e., psp is h-helped with $h \geq 1$). In that case, the array of values returned by psp has been computed by some update operation at line 02 (Algorithm 2). Moreover, whatever the case, this array has been computed by a successful double scan executed by some process p_z. When considering this successful double scan, $\ell p(\mathsf{psp})$ is placed between the end of its first scan and the beginning of its second scan.

If two operations are about to be linearized at the same point, they are arbitrarily ordered (e.g., according to the identities of the processes that issued them).

It follows from the previous linearization point definitions that each operation is linearized between its beginning and its end, and no two operations are linearized at the same point.

- Point (b): the sequence of update and snapshot operations defined by their linearization points satisfies the specification of the snapshot object.

  This follows directly from Lemma 2 that showed that the values returned by every snapshot operation are well-defined, mutually consistent and fresh.

$\square_{Lemma\ 5}$

4.5 The construction is correct, wait-free and efficient

Theorem 1 *Algorithms 1 and 2 satisfy the termination and consistency properties (stated in Section 2.2) that define a snapshot object. Moreover, they satisfy the freshness property.*

Proof The proof that the algorithms satisfy termination and consistency properties follows from Lemmas 4 and 5. The freshness property follows from the definition of the linearization points given in Lemma 5. $\square_{Theorem\ 1}$

4.6 Cost of the algorithm

This section analyses the cost of the update and snapshot operations in terms of the number of base atomic registers that are accessed by a read or write operation.

- Snapshot operation (Algorithm 1).
  - Best case. In the best case a snapshot operation returns at line 05 after having read only twice the array $REG[1..m]$. The cost is then the $2m$.
  - Worst case. Let p_i be the process that invoked the snapshot operation. The worst case is when a process returns at line 10, and the local array can_help_i contains n pairs: a pair from every process but p_i, plus another pair from an other process. In that case, p_i has read $n+1$ times the array $REG[1..m]$ and, consequently, has accessed $(n+1)m$ times the shared memory.
- The cost of an update operation is the cost on a snapshot operation plus 1.

It follows that the cost of an operation is $O(n \times m)$.

5 Conclusion

This paper has presented a multi-writer/multi-reader snapshot algorithm. Differently from all other snapshot algorithms proposed so far (except [14]), the update() operation first writes the value to the shared memory, and later computes a helping snapshot value: this is the "write first, help later" strategy. This allows more asynchrony as the processes don't have to write the value along with the helping snapshot in an atomic way. This also allows processes that invoke a snapshot() operation to obtain values as recent as possible.

Another feature of this algorithm is its exceptional simplicity. This is a first class property for algorithms: it allows a better understanding of the algorithm and, consequently, of how processes can safely communicate and interact in a shared memory system.

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] Aguilera M.K., A Pleasant Stroll Through the Land of Infintely Many Creatures. *ACM SIGACT-NEWS*, 35(2):36-59, 2004.

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

[4] 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.

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

[6] Attiya H. and Welch J., *Distributed Computing: Fundamentals, Simulations and Advanced Topics,* (2d Edition), Wiley-Interscience, 414 pages, 2004.

[7] Chandy K.M. and Lamport L., Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63-75, 1985.

[8] Ellen F., How Hard Is It to Take a Snapshot? *Proc. 31th Conference on Current Trends in Theory and Practice of Computer Science (SOFTSEM'05)*, Springer-Verlag #3381, pp. 28-37, 2005.

[9] Guerraoui R. and Raynal M., From Unreliable Objects to Reliable Objects: the Case of Atomic Registers and Consensus. *9th Int'l Conference on Parallel Computing Technologies (PaCT'07)*, Springer Verlag LNCS #4671, pp. 47-61, 2007.

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

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

[12] Herlihy M.P. and Luchangco V., Distributed Computing and the Multicore Revolution. *ACM SIGACT News*, 39(1): 62-72, 2008.

[13] 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.

[14] Imbs D. and Raynal M. Help when needed, but no more: Efficient Read/Write Partial Snapshot. *Proc. 23th Int'l Symposium on Distributed Comput ing (DISC'09)*, Springer-Verlag LNCS #5805, pp. 142-156, 2009.

[15] Inoue I., Chen W., Masuzawa T. and Tokura N., Linear Time Snapshots Using Multi-writer Multi-reader Registers. *Proc. 8th Int'l Workshop on Distributed Algorithms (WDAG'94)*, Springer-Verlag #857, pp. 130-140, 1994.

[16] Jayanti P., An Optimal Multiwriter Snapshot Algorithm. *Proc. 37th ACM Symposium on Theory of Computing (STOCS'05)*, ACM Press, pp. 723-732, 2005.

[17] Lynch N.A., Distributed Algorithms. *Morgan Kaufmann Pub.*, San Francisco (CA), 872 pages, 1996.

[18] Raynal M., Synchronization is coming back, but is it the same? *IEEE 22nd Int'l Conference on Advanced Information Networking and Applications (AINA'08)*, pp. 1-10, Okinawa (Japan), 2008.

[19] Taubenfeld G., Synchronization Algorithms and Concurrent Programming. *Pearson Prentice-Hall*, ISBN 0-131-97259-6, 423 pages, 2006.