On Composition and Implementation of Sequential Consistency

Matthieu Perrin, Matoula Petrolia, Achour Mostefaoui, Claude Jard

On Composition and Implementation of Sequential Consistency

Matthieu Perrin, Matoula Petrolia, Achour Mostéfaoui, and Claude Jard

LINA – University of Nantes, 2 rue de la Houssinière, 44322 Nantes Cedex 3, France

matthieu.perrin@univ-nantes.fr
stamatina.petrolia@univ-nantes.fr
achour.mostefaoui@univ-nantes.fr
claude.jard@univ-nantes.fr

Abstract. It has been proved that to implement a linearizable shared memory in synchronous message-passing systems it is necessary to wait for a time proportional to the uncertainty in the latency of the network for both read and write operations, while waiting during read or during write operations is sufficient for sequential consistency.
This paper extends this result to crash-prone asynchronous systems. We propose a distributed algorithm that builds a sequentially consistent shared memory abstraction with snapshot on top of an asynchronous message-passing system where less than half of the processes may crash. We prove that it is only necessary to wait when a read/snapshot is immediately preceded by a write on the same process.
We also show that sequential consistency is composable in some cases commonly encountered: 1) objects that would be linearizable if they were implemented on top of a linearizable memory become sequentially consistent when implemented on top of a sequential memory while remaining composable and 2) in round-based algorithms, where each object is only accessed within one round.

Key Words. Asynchronous message-passing system, Crash-failures, Composability, Sequential consistency, Shared memory, Snapshot.

1 Introduction

A distributed system is abstracted as a set of entities (nodes, processes, agents, etc) that communicate with each other using a communication medium. The two most used communication media are communication channels (message-passing system) and shared memory (read/write operations). Programming with shared objects is generally more convenient as it offers a higher level of abstraction to the programmer, therefore facilitates the work of designing distributed applications. A natural question is the level of consistency ensured by shared objects. An intuitive property is that shared objects should behave as if all processes accessed the same physical copy of the object. *Sequential consistency* [1] ensures that all the operations that happen in a distributed history appear as if they were executed sequentially, in an order that respects the sequential order of each process (called the *process order*).

Unfortunately, sequential consistency is not composable: if a program uses two or more objects, despite each object being sequentially consistent individually, the set of all objects may not be sequentially consistent. An example is shown in Fig. 1, where two processes share two registers named X and Y; although the operations of each register may be totally ordered (the read precedes the write), it is impossible to order all the operations at once. *Linearizability* [2] overcomes this limitation by adding constraints on real time: each operation appears at a single point in time, between its start event and its end event. As a consequence, linearizability enjoys the locality property [2] that ensures its composability. Because of this composability, much more effort has been focused on linearizability than on sequential consistency so far. However, one of our contributions implies that in asynchronous systems where no global clock can be implemented to measure real time, a process cannot distinguish between linearizability and sequential consistency, thus the connection to real time seems to be a worthless — though costly — guarantee.

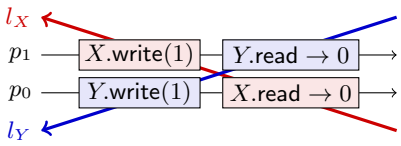


Fig. 1: Sequential consistency is not composable: registers X and Y are both sequentially consistent but their composition is not.

In this paper we focus on message-passing distributed systems. In such systems a shared memory is not a physical object; it has to be built using the underlying message-passing communication network. Several bounds have been found on the cost of sequential consistency and linearizability in synchronous distributed systems, where the transit time for any message is in a range $[d-u, d]$, where d and u are called respectively the *latency* and the *uncertainty* of the network. Let us consider an implementation of a shared memory, and let r (resp. w) be the worst case latency of any read (resp. write) operation. Lipton and Sandberg proved in [3] that, if the algorithm implements a sequentially consistent memory, the inequality $r + w \geq d$ must hold. Attiya and Welch refined this result in [4], proving that each kind of operations could have a 0-latency implementation for sequential consistency (though not both in the same implementation) but that the time duration of both kinds of operations has to be at least linear in u in order to ensure linearizability.

Therefore the following questions arise. Are there applications for which the lack of composability of sequential consistency is not a problem? For these applications, can we expect the same benefits in weaker message-passing models, such as asynchronous failure-prone systems, from using sequentially consistent objects rather than linearizable objects?

To illustrate the contributions of the paper, we also address a higher level operation: a snapshot operation [5] that allows to read in a single operation a whole set of registers. A sequentially consistent snapshot is such that the set of values it returns may be returned by a sequential execution. This operation is very useful as it has been proved [5] that linearizable snapshots can be wait-free implemented from single-writer/multi-reader registers. Indeed, assuming a snapshot operation does not bring any additional power with respect to shared registers. Of course this induces an additional cost: the best known simulation needs $O(n \log n)$ basic read/write operations to implement each of the snapshot operations and the associated update operation [6]. Such an operation brings a programming comfort as it reduces the "noise" introduced by asynchrony and failures [7] and is particularly used in round-based computations [8] we consider for the study of the composability of sequential consistency.

Contributions. This paper has three major contributions. (1) It identifies two contexts that can benefit from the use of sequential consistency: round-based algorithms that use a different shared object for each round, and asynchronous shared-memory systems, where programs can not differentiate a sequentially consistent memory from a linearizable memory. (2) It proposes an implementation of a sequentially consistent memory where waiting is only required when a write is immediately followed by a read. This extends the result presented in [4], which only applies to synchronous failure-free systems, to failure-prone asynchronous systems. (3) The proposed algorithm also implements a sequentially consistent snapshot operation the cost of which compares very favorably with the best existing linearizable implementation to our knowledge (the stacking of the snapshot algorithm of Attiya and Rachman [6] over the ABD simulation of linearizable registers).

Outline. The remainder of this article is organized as follows. In Section 2, we define more formally sequential consistency, and we present special contexts in which it becomes composable. Then, in Section 3, we present our implementation of shared memory and study its complexity. Finally, Section 4 concludes the paper.

2 Sequential Consistency and Composability

2.1 Definitions

In this section we recall the definitions of the most important notions we discuss in this paper: two consistency criteria, sequential consistency (SC, Def. 2, [1]) and linearizability (L, Def. 3, [2]), as well as composability (Def. 4). A consistency criterion associates a set of admitted *histories* to the *sequential specification* of each given object. A history is a representation of an execution. It contains a set of operations, that are partially ordered according to the sequential order of each process, called *process order*. A sequential specification is a language, i.e. a set of sequential (finite and infinite) words. For a consistency criterion C and a sequential specification T, we say that an algorithm implements a $C(T)$-consistent object if all its executions can be modelled by a history that belongs to $C(T)$, that contains all returned operations and only invoked operations. Note that this implies that if a process crashes during an operation, then the operation will appear in the history as if it was complete or as if it never took place at all.

Definition 1 (Linear extension). *Let H be a history and T be a sequential specification. A* linear extension $\leq$ *is a total order on all the operations of H, that contains the process order, and such that each event e has a finite past $\{e' : e' \leq e\}$ according to the total order.*

Definition 2 (Sequential Consistency). *Let H be a history and T be a sequential specification. The history H is* sequentially consistent *regarding T, denoted $H \in SC(T)$, if there exists a linear extension $\leq$ such that the word composed of all the operations of H ordered by $\leq$ belongs to T.*

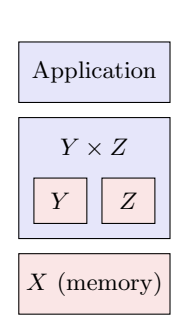
Definition 3 (Linearizability). *Let H be a history and T be a sequential specification. The history H is* linearizable *regarding T, denoted $H \in L(T)$, if there exists a linear extension $\leq$ such that (1) for two operations a and b, if the end of a precedes the beginning of b in real time, then $a \leq b$ and (2) the word formed of all the operations of H ordered by $\leq$ belongs to T.*

Let T_1 and T_2 be two sequential specifications. We define the *composition* of T_1 and T_2, denoted by $T_1 \times T_2$, as the set of all the interleaved sequences of a word from T_1 and a word from T_2. An interleaved sequence of two words l_1 and l_2 is a word composed of the disjoint union of all the letters of l_1 and l_2, that appear in the same order as they appear in l_1 and l_2. For example, the words ab and cd have six interleaved sequences: $abcd$, $acbd$, $acdb$, $cabd$, $cadb$ and $cdab$.

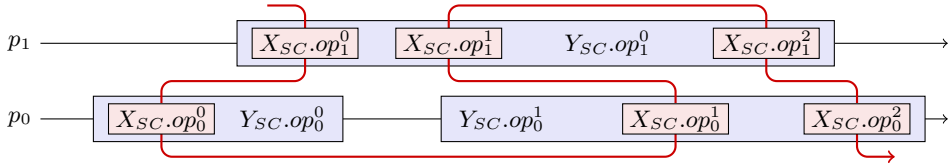
A consistency criterion C is composable (Def. 4) if the composition of a $C(T_1)$-consistent object and a $C(T_2)$-consistent object is a $C(T_1 \times T_2)$-consistent object. Linearizability is composable, and sequential consistency is not.

Definition 4 (Composability). *For a history H and a sequential specification T, let us denote by H_T the sub-history of H that only contains the operations belonging to T.*

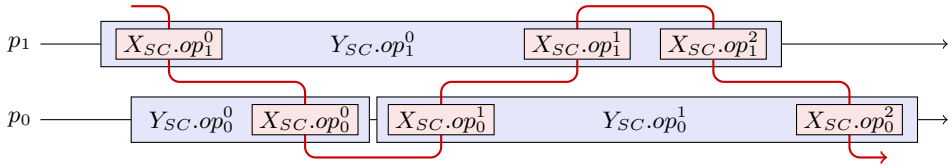
A consistency criterion C is composable *if, for all sequential specifications T_1 and T_2 and all histories H containing only events on T_1 and T_2, ($H_{T_1} \in C(T_1)$ and $H_{T_2} \in C(T_2)$) imply $H \in C(T_1 \times T_2)$.*



(a) Layer based architecture.



(b) The implementation of upper layer objects call operations on objects from lower layers.



(c) An asynchronous process cannot differentiate this history from the one on Figure 2b.

Fig. 2: In layer based program architecture running on asynchronous systems, local clocks of different processes can be distorted such that it is impossible to differentiate a sequentially consistent execution from a linearizable execution.

2.2 From Linearizability to Sequential Consistency

Software developers usually abstract the complexity of their system gradually, which results in a layered software architecture: at the top level, an application is built on top of several objects specific to the application, themselves built on top of lower levels. Such an architecture is represented in Fig. 2a. The lowest layer usually consists of one or several objects provided by the system itself, typically a shared memory. The system can ensure sequential consistency globally on all the provided objects, therefore composability is not required for this level. Proposition 1 expresses the fact that, in asynchronous systems, replacing a linearizable object by a sequentially consistent one does not affect the correctness of the programs running on it circumventing the non composability of sequential consistency. This result may have an impact on parallel architectures, such as modern multi-core processors and, to a higher extent, high performance supercomputers, for which the communication with a linearizable central shared memory is very costly, and weak memory models such as cache consistency [9] make the writing of programs tough.

Proposition 1. *Let A be an algorithm that implements an $SC(Y)$-consistent object when it is executed on an asynchronous system providing an $L(X)$-consistent object. Then A also implements an $SC(Y)$-consistent object when it is executed in an asynchronous system providing an $SC(X)$-consistent object.*

Proof. Let A be an algorithm that implements an $SC(Y)$-consistent object when it is executed on an asynchronous system providing an $L(X)$-consistent object.

Let us consider a history H_{SC} obtained by the execution of A in an asynchronous system providing a $SC(X)$-consistent object. Such a history is depicted on Fig. 2b. The history H_{SC} contains operations on X (in red in Fig. 2b), as well as on Y (in blue in Fig. 2b).

We will now build another history H_L, in which the operations on X are linearizable, and the operations on Y consist in the same calls to operations on X. Such a history is depicted on Fig. 2c. The only difference between the histories on Fig. 2b and 2c is the way the two processes experience time. As the system is asynchronous, it is impossible for them to distinguish them.

Let us enumerate all the operations made on X in their linear extension $\leq$ required for sequential consistency. Now, we build the execution H_L in which the i^{th} operation on X of H_{SC} is called on an $L(X)$-consistent object at time $2i$ seconds and lasts for one second. As no two

operations overlap, and the operations happen in the same order in H_L and in the linearization of H_{SC}, $\leq$ is the only linear extension accepted by linearizability. Therefore, all operations can return the same values in H_L and in H_{SC} (and they will if X is deterministic). Now let us assume all operations on X in H_L were called by algorithm A, in the same pattern as in H_{SC}. When considering the operations on Y, H_L is $SC(Y)$-consistent. Moreover, as A works on asynchronous systems and the same values were returned by X in H_{SC} and in H_L, A returns the same values in both histories. Therefore, H_{SC} is also $SC(Y)$-consistent.

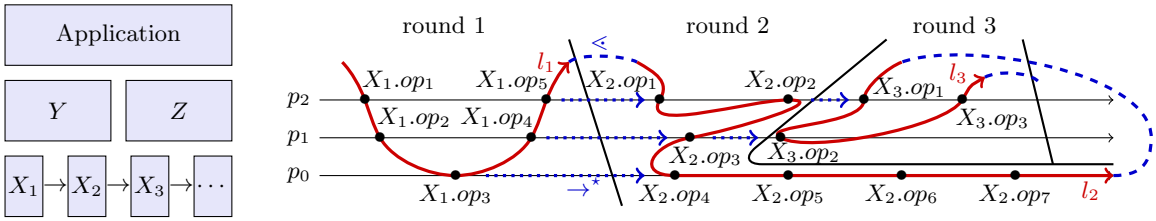
An interesting point about Proposition 1 is that it allows sequentially consistent — but not linearizable — objects to be composable. Let A_Y and A_Z be two algorithms that implement $L(Y)$-consistent and $L(Z)$-consistent objects when they are executed on an asynchronous system providing an $L(X)$-consistent object, like on Fig. 2a. As linearizability is stronger than sequential consistency, according to Proposition 1, executing A_Y and A_Z on an asynchronous system providing an $SC(X)$-consistent object would implement sequentially consistent — yet not linearizable — objects. However, in a system providing the linearizable object X, by composability of linearizability, the composition of A_Y and A_Z implements an $L(Y \times Z)$-consistent object. Therefore, by Proposition 1 again, in a system providing the sequentially consistent object X, the composition also implements an $SC(Y \times Z)$-consistent object. In this example, the sequentially consistent versions of Y and Z derive their composability from an anchor to a *common time*, given by the sequentially consistent memory, that can differ from *real time*, required by linearizability.

2.3 Round-Based Computations

Even at a single layer, a program can use several objects that are not composable, but that are used in a fashion so that the non-composability is invisible to the program. Let us illustrate this with round-based algorithms. The synchronous distributed computing model has been extensively studied and well-understood leading the researchers to try to offer the same comfort when dealing with asynchronous systems, hence the introduction of synchronizers [10]. A synchronizer slices a computation into phases during which each process executes three steps: send/write, receive/read and then local computation. This model has been extended to failure prone systems in the round-by-round computing model [8] and to the Heard-Of model [11] among others. Such a model is particularly interesting when the termination of a given program is only eventual. Indeed, some problems are undecidable in failure prone purely asynchronous systems. In order to circumvent this impossibility, eventually or partially synchronous systems have been introduced [12]. In such systems the termination may hold only after some finite but unbounded time, and the algorithms are implemented by the means of a series of asynchronous rounds each using its own shared objects.

In the round-based computing model, the execution is sliced into a sequence of asynchronous rounds. During each round, a new data structure (usually a single-writer/multi-reader register per process) is created and it is the only shared object used to communicate during the round. At the end of the round, each process destroys its local accessor to the object, so that it can no more access it. Note that the rounds are asynchronous: the different processes do not necessarily start and finish their rounds at the same time. Moreover, a process may not terminate a round, and keep accessing the same shared object forever or may crash during this round and stop executing. A round-based execution is illustrated in Fig. 3b.

In Proposition 2, we prove that sequentially consistent objects of different rounds behave well together: as the ordering added between the operations of two different objects always follows the round numbering, that is consistent with the program order already contained in the linear extension of each object, the composition of all these objects cannot create loops



(a) Round-based program architecture.

(b) As the ordering between different objects follows the process order, that is contained into the serialization order of each object, no loop can appear.

Fig. 3: The composition of sequentially consistent objects used in different rounds is sequentially consistent.

(Figure 3b). Putting together this result and Proposition 1, all the algorithms that use a round-based computation model can benefit of any improvement on the implementation of an array of single-writer/multi-reader register that sacrifices linearizability for sequential consistency. Note that this remains true whatever is the data structure used during each round. The only constraint is that a sequentially consistent shared data structure can be accessed during a unique round. If each object is sequentially consistent then the whole execution is consistent.

Proposition 2. *Let $(T_r)_{r\in\mathbb{N}}$ be a family of sequential specifications and $(X_r)_{r\in\mathbb{N}}$ be a family of shared objects such that, for all r, X_r is $SC(T_r)$-consistent. Let H be a history that does not contain two operations $X_r.a$ and $X_{r'}.b$ with $r > r'$ such that $X_r.a$ precedes $X_{r'}.b$ in the process order. Then H is sequentially consistent with respect to the composition of all the T_r.*

Proof. Let $(T_r)_{r\in\mathbb{N}}$ be a family of sequential specifications and $(X_r)_{r\in\mathbb{N}}$ be a family of shared object such that, for all r, X_r is $SC(T_r)$-consistent. Let H be a history that does not contain two operations $X_r.a$ and $X_{r'}.b$ with $r > r'$ such that $X_r.a$ precedes $X_{r'}.b$ in the process order.

For each X_r, there exists a linearization l_r that contains the operations on X_r and respects T_r. For each operation op, let us denote by $op.r$ the index of the object X_r on which it is made and by $op.i$ the number of operations that precede op in the linearization l_r. Let us define two binary relations $\lessdot$ and $\rightarrow$ on the operations of H. For two operations op and op', $op \lessdot op'$ if $op.r < op'.r$, or $op.r = op'.r$ and $op.i \leq op'.i$. Note that $\lessdot$ is the concatenation of all the linear extensions, so it is a total order on all the operations of H, but it may not be a linear extension as an operation can have an infinite past if a process does not finish its round. For two operations op and op', $op \rightarrow op'$ if op and op' were done in that order by the same process, or $op.r = op'.r$ and $op.i \leq op'.i$. Let $\rightarrow^\star$ be the transitive closure of $\rightarrow$.

Notice that, according to the round based model, $\rightarrow$ is contained into $\lessdot$, and so is $\rightarrow^\star$ because $\lessdot$ is transitive. The relation $\rightarrow^\star$ is transitive and reflexive by construction. Moreover, if $op \rightarrow^\star op' \rightarrow^\star op$, we have $op.r \leq op'.r \leq op.r$ and therefore $op.i \leq op'.i \leq op.i$, so $op = op'$ (antisymmetry), which proves that $\rightarrow^\star$ is a partial order. Moreover, let us suppose that an operation contains an infinite past according to $\rightarrow^\star$. There is a smallest such operation, $op_{\min}$, according to $\lessdot$. The direct predecessors of $op_{\min}$ according to $\rightarrow$ are smaller than $op_{\min}$ according to $\lessdot$, so they have a finite past. Moreover, they precede $op_{\min}$ either in the process order or in the linearization $l_{op.r}$, so there is a finite number of them. This is a contradiction, so all operations have a finite past according to $\rightarrow^\star$. It is possible to extend $\rightarrow^\star$ to a total order $\leq$ such that all operations have a finite past according to $\leq$. As $\leq$ contains the total orders defined by all the l_r, the execution of all the operations in the order $\leq$ respects the sequential specification of the composition of all the X_r.

3 Implementation of a Sequentially Consistent Memory

In this section we will describe the computation model that we consider for the implementation of a sequentially consistent shared memory (Section 3.1). In Section 3.2 we will discuss the characteristics of such a memory and, finally, in Section 3.3 we will present the proposed implementation of the discussed data structure. Finally, in Section 3.5 we discuss the complexity of the proposed implementation.

3.1 Computation Model

The computation system consists of a set Π of n sequential processes which are denoted $p_0, p_1, \ldots, p_{n-1}$. The processes are asynchronous, in the sense that they all proceed at their own speed, not upper bounded and unknown to all other processes.

Among these n processes, up to t may crash (halt prematurely) but otherwise execute correctly the algorithm until the moment of their crash. We call a process *faulty* if it crashes, otherwise it is called *correct* or *non-faulty*. In the rest of the paper we will consider the above model restricted to the case $t < \frac{n}{2}$.

The processes communicate with each other by sending and receiving messages through a complete network of bidirectional communication channels. This means that a process can directly communicate with any other process, including itself (p_i receives its own messages instantaneously), and can identify the sender of the message it received. Each process is equipped with two operations: **send** and **receive**.

The channels are reliable (no losses, no creation, no duplication, no alteration of messages) and asynchronous (finite time needed for a message to be transmitted but there is no upper bound). We also assume the channels are FIFO: if p_i sends two messages to p_j, p_j will receive them in the order they were sent. As stated in [13], FIFO channels can always be implemented on top of non-FIFO channels. Therefore, this assumption does not bring additional computational power to the model, but it allows us to simplify the writing of the algorithm. Process p_i can also use the macro-operation **FIFO broadcast**, that can be seen as a multi-send that sends a message to all processes, including itself. Hence, if a faulty process crashes during the broadcast operation some processes may receive the message while others may not, otherwise all correct processes will eventually receive the message.

3.2 Single-Writer/Multi-Reader Registers and Snapshot Memory

The shared memory considered in this paper, called a *snapshot memory*, consists of an array of shared registers denoted $\mathsf{REG}[1..n]$. Each entry $\mathsf{REG}[i]$ represents a single-writer/multi-reader (SWMR) register. When process p_i invokes $\mathsf{REG}.\mathtt{update}(v)$, the value $\mathtt{v}$ is written into the SWMR register $\mathsf{REG}[i]$ associated with process p_i. Differently, any process p_i can read the whole array REG by invoking a single operation namely $\mathsf{REG}.\mathtt{snapshot}()$. According to the sequential specification of the snapshot memory, $\mathsf{REG}.\mathtt{snapshot}()$ returns an array containing the most recent value written by each process or the initial default value if no value is written on some register. Concurrency is possible between snapshot and writing operations, as soon as the considered consistency criterion, namely linearizability or sequential consistency, is respected. Informally in a sequentially consistent snapshot memory, each snapshot operation must return the last value written by the process that initiated it, and for any pair of snapshot operations, one must return values at least as recent as the other for all registers.

Compared to read and write operations, the snapshot operation is a higher level abstraction introduced in [5] that eases program design without bringing additional power with respect to shared registers. Of course this induces an additional cost: the best known simulation, above

SWMR registers proposed in [6], needs $O(n \log n)$ basic read/write operations to implement each of the snapshot and the associated update operations.

Since the seminal paper [14] that proposed the so-called ABD simulation that emulates a linearizable shared memory over a message-passing distributed system, most of the effort has been put on the shared memory model given that a simple stacking allows to translate any shared memory-based result to the message-passing system model. Several implementations of linearizable snapshot have been proposed in the literature some works consider variants of snapshot (e.g. immediate snapshot [15], weak-snapshot [16], one scanner [17]) others consider that special constructions such as test-and-set (T&S) [18] or load-link/store-conditional (LL/SC) [19] are available, the goal being to enhance time and space efficiency. In this paper, we propose the first message-passing sequentially consistent (not linearizable) snapshot memory implementation directly over a message-passing system (and consequently the first sequentially consistent array of SWMR registers), as traditional read and write operations can be immediately deduced from snapshot and update with no additional cost.

3.3 The Proposed Algorithm

Algorithm 1 proposes an implementation of the sequentially consistent snapshot memory data structure presented in Section 3.2. Process p_i can write a value $\mathtt{v}$ in its own register $\mathsf{REG}[i]$ by calling the operation $REG.\mathtt{update}(v)$, implemented by the lines 6-9. It can also call the operation $REG.\mathtt{snapshot}()$, implemented by the lines 10-11. Roughly speaking, the principle of this algorithm is to maintain, on each process, a local view of the object that reflects a set of *validated* update operations. To do so, when a value is written, all processes label it with their own timestamp. The order in which processes timestamp two different update operations define a *dependency relation* between these operations. For two operations a and b, if b depends on a, then p_i cannot validate b before a.

More precisely, each process p_i maintains five local variables:

- $\mathsf{X}_i \in \mathbb{N}^n$ represents the array of most recent validated values written on each register.
- $\mathsf{ValClock}_i \in \mathbb{N}^n$ represents the timestamps associated with the values stored in X_i, labelled by the process that initiated them.
- $\mathsf{SendClock}_i \in \mathbb{N}$ is an integer clock used by p_i to timestamp all the update operations. $\mathsf{SendClock}_i$ is incremented each time a message is sent, which ensures all timestamps from the same process are different.
- $\mathsf{G}_i \subset \mathbb{N}^{3+n}$ encodes the dependencies between the update operations that have not been validated yet, as they are known by p_i. An element $g \in \mathsf{G}_i$, of the form $(g.\mathtt{v}, g.\mathtt{k}, g.\mathtt{t}, g.\mathtt{cl})$, represents the update operation of value $g.\mathtt{v}$ by process $p_{g.\mathtt{k}}$ labelled by process $p_{g.\mathtt{k}}$ with timestamp $g.\mathtt{t}$. For all $0 \leq j < n$, $g.\mathtt{cl}[j]$ contains the timestamp associated by p_j if it is known by p_i, and ∞ otherwise.
  All updates of a history can be uniquely represented by a pair of integers (k, t), where p_k is the process that invoked it, and t is the timestamp associated to this update by p_k. Considering a history and a process p_i, we define the dependency relation $\rightarrow_i$ on pairs of integers (k, t), by $(k, t) \rightarrow_i (k', t')$ if for all g, g' ever inserted in G_i with $(g.\mathtt{k}, g.\mathtt{t}) = (k, t)$, $(g'.\mathtt{k}, g'.\mathtt{t}) = (k', t')$, we have $|\{j : g'.\mathtt{cl}[j] < g.\mathtt{cl}[j]\}| \leq \frac{n}{2}$ (i.e. the dependency does not exist if p_i knows that a majority of processes have seen the first update before the second). Let $\rightarrow_i^\star$ denote the transitive closure of $\rightarrow_i$.
- $\mathsf{V}_i \in \mathbb{N} \cup \{\perp\}$ is a buffer register used to store a value written while the previous one is not yet validated. This is necessary for validation (see below).

The key of the algorithm is to ensure the inclusion between sets of validated updates on any two processes at any time. Remark that it is not always necessary to order all pairs of update

Algorithm 1: Implementation of a sequentially consistent memory (code for p_i)

```
   /* Local variable initialization                                                    */
 1 X_i ← [0, ..., 0];                          // X_i ∈ ℕ^n: X_i[j] is the last validated value written by p_j
 2 ValClock_i ← [0, ..., 0];        // ValClock_i ∈ ℕ^n: ValClock_i[j] is the stamp given by p_j to value X_i[j]
 3 SendClock_i ← 0;                                  // SendClock_i ∈ ℕ: used to stamp all the updates
 4 G_i ← ∅;          // G_i ⊂ ℕ^{3+n}: contains a g = (g.v, g.k, g.t, g.cl) per non-val. update of g.v by p_{g.k}
 5 V_i ← ⊥;           // V_i ∈ ℕ ∪ {⊥}: stores updates that have not yet been proposed to validation
   ──────────────────────────────────────────────────────────────────────────────────
   operation update(v) /* v ∈ ℕ: written value; no return value                       */
 6   if ∀g ∈ G_i : g.k ≠ i then                                  // no non-validated update by p_i
 7     SendClock_i++;
 8     FIFO broadcast message(v, i, SendClock_i, SendClock_i);
 9   else V_i ← v;                                                  // postpone the update
   ──────────────────────────────────────────────────────────────────────────────────
   operation snapshot() /* return type: ℕ^n                                          */
10   wait until V_i = ⊥ ∧ ∀g ∈ G_i : g.k ≠ i ;                  // make sure p_i's updates are validated
11   return X_i;
   ──────────────────────────────────────────────────────────────────────────────────
   when a message message(v, k, t, cl) is received from p_j
     /* v ∈ ℕ: written value, k ∈ ℕ: writer id, t ∈ ℕ: stamp by p_k, cl ∈ ℕ: stamp by p_j  */
12   if t > ValClock_i[k] then                                          // update not validated yet
13     if ∃g ∈ G_i : g.k = k ∧ g.t = t then                               // update already known
14       g.cl[j] ← cl;
15     else                                                        // first message for this update
16       if k ≠ i then
17         SendClock_i++;
18         FIFO broadcast message(v, k, t, SendClock_i);              // forward with own stamp
19       var g ← (g.v = v, g.k = k, g.t = t, g.cl = [∞, ..., ∞]);
20       g.cl[j] ← cl;
21       G_i ← G_i ∪ {g};                                  // create an entry in G_i for the update
22   var G' = {g ∈ G_i : |{l : g'.cl[l] < ∞}| > n/2};        // G' contains updates that can be validated
23   while ∃g ∈ G_i \ G', g' ∈ G' : |{l : g'.cl[l] < g.cl[l]}| ≠ n/2 do  G' ← G' \ {g'};
24   G_i ← G_i \ G';                                                  // validate updates of G'
25   for g ∈ G' do
26     if ValClock_i[g.k] < g.t then ValClock_i[g.k] = g.t; X_i[g.k] = g.V;
27   if V_i ≠ ⊥ ∧ ∀g ∈ G_i : g.k ≠ i then   // start validation process for postponed update if any
28     SendClock_i++;
29     FIFO broadcast message(V_i, i, SendClock_i, SendClock_i);
30     V_i ← ⊥;
```

operations to implement a sequentially consistent snapshot memory: for example, two update operations on different registers commute. Therefore, instead of validating both operations on all processes in the exact same order (which requires Consensus), we can validate them at the same time to prevent a snapshot to occur between them. Therefore, it is sufficient to ensure that, for all pairs of update operations, there is a dependency agreed by all processes (possibly in both directions). This property is expressed by Lemma 2 from Section 3.4.

This is done by the mean of messages of the form $\texttt{message}(\mathsf{v}, \mathsf{k}, \mathsf{t}, \mathsf{cl})$ containing four integers: v the value written, k the identifier of the process that initiated the update, t the timestamp given by p_{k} and cl the timestamp given by the process that sent this message. Timestamps of successive messages sent by p_i are unique and totally ordered, thanks to variable $\mathsf{SendClock}_i$, that is incremented each time a message is sent by p_i. When process p_i wants to submit a value v for validation, it FIFO-broadcasts a message $\texttt{message}(\mathsf{v}, i, \mathsf{SendClock}_i, \mathsf{SendClock}_i)$ (lines 8 and 29). When p_i receives a message $\texttt{message}(\mathsf{v}, \mathsf{k}, \mathsf{t}, \mathsf{cl})$, three cases are possible. If p_i has

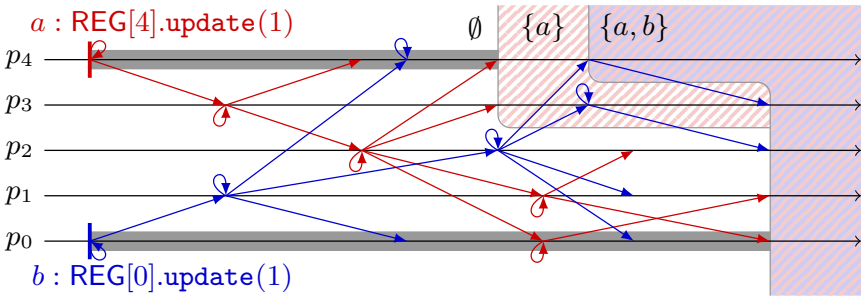


Fig. 4: An execution of Algorithm 1. An update is validated by a process when it has received enough messages for this update, and all the other updates it depends of have also been validated.

already validated the corresponding update ($\mathsf{t} > \mathsf{ValClock}_i[\mathsf{k}]$), the message is simply ignored. Otherwise, if it is the first time p_i receives a message concerning this update (G_i does not contain any piece of information concerning it), it FIFO-broadcasts a message with its own timestamp and adds a new entry $g \in \mathsf{G}_i$. Whether it is its first message or not, p_i records the timestamp cl, given by p_j, in $g.\mathtt{cl}[j]$ (lines 14 or 20). Note that we cannot update $g.\mathtt{cl}[\mathsf{k}]$ at this point, as the broadcast is not causal: if p_i did so, it could miss dependencies imposed by the order in which p_k saw concurrent updates. Then, p_i tries to validate update operations: p_i can validate an operation a if it has received messages from a majority of processes, and there is no operation $b \rightarrow_i^\star a$ that cannot be validated. For that, it creates the set G' that initially contains all the operations that have received enough messages, and removes all operations with unvalidatable dependencies from it (lines 22-23), and then updates X_i and $\mathsf{ValClock}_i$ with the most recent validated values (lines 24-26).

This mechanism is illustrated in Fig. 4, featuring five processes. Processes p_0 and p_4 initially call operation $\mathsf{REG}.\mathtt{update}(1)$. Messages that have an impact in the algorithm are represented by arrows, and messages that do not appear on the figure are received later. Several situations may occur. The simplest case is process p_3, that received three messages concerning a (from p_4, p_3 and p_2, with $3 > \frac{n}{2}$) before its first message concerning b, allowing it to validate a. The case of process p_4 is similar: even if it knows that process p_1 saw b before a, it received messages concerning a from three *other* processes, which allows it to ignore the message from p_1. At first sight, the situation of processes p_0 and p_1 may look similar to the situation of p_4. However, the message they received concerning a and one of the messages they received concerning b come from the same process p_2, which forces them to respect the dependency $a \rightarrow_0 b$. Note that the same situation occurs on process p_2 so, even if a has been validated before b by other processes, p_2 must respect the dependency $b \rightarrow_2 a$.

Sequential consistency requires the total order to contain the process order. Therefore, a snapshot of process p_i must return values at least as recent as its last updated value. In other words, it is not allowed to return from a snapshot between an update and the time when it is validated (grey zones in Fig. 4). There are two ways to implement this: we can either wait at the end of each update until it is validated, in which case all snapshot operations are done for free, or wait at the beginning of all snapshot operations that immediately follow an update operation. This extends the remark of [4] to crash-prone asynchronous systems: to implement a sequentially consistent memory, it is necessary and sufficient to wait either during read or during write operations. In Algorithm 1, we chose to wait during read/snapshot operations (line 10). This is more efficient for two reasons: first, it is not necessary to wait between two consecutive updates, which can not be avoided if we wait at the end of the update operation, and second

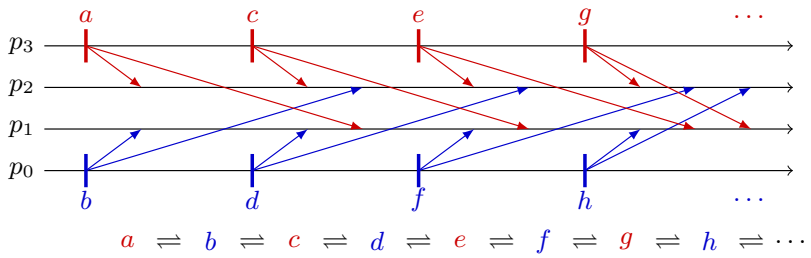


Fig. 5: If we are not careful, infinite chains of dependencies may occur. We must avoid infinite chains of dependencies in order to ensure termination

the time between the end of an update and the beginning of a snapshot counts in the validation process, but it can be used for local computations. Note that when two snapshot operations are invoked successively, the second one also returns immediately, which improves the result of [4] according to which waiting is necessary for all the operations of one kind.

In order to obtain termination of the snapshot operations (and progress in general), it is necessary to ensure that all update operations are eventually validated by all processes. This property is expressed by Lemma 3 from Section 3.4. Figure 5 illustrates what could happen. On the one hand, process p_2 receives a message concerning a and a message concerning c before a message concerning b. On the other hand, process p_1 receives a message concerning b before messages concerning a and c. Therefore, it may create dependencies $a \rightarrow_i b \rightarrow_i c \rightarrow_i b \rightarrow_i a$ on some process p_i, which means p_i will be forced to validate a and c at the same time, even if they are ordered by the process order. The pattern in Fig. 5 shows that it can result in an infinite chain of dependencies, blocking validation of any update operation. To break this chain, we force process p_3 to wait until a is validated locally before it proposes c to validation, by storing the value written by c in a local variable V_i until a is validated (lines 6 and 9). When a is validated, we start the same validation process for c (lines 27-30). Remark that, if several updates (say c and e) happen before a is validated, the update of c can be dropped as it will eventually be overwritten by e. In this case, c will happen just before e in the final linearization required for sequential consistency.

This algorithm could be adjusted to implement multi-writer/multi-reader registers. Only three points must be changed. First, the identifier of the register written should be added to all messages and all $g \in \mathsf{G}_i$. Second, concurrent updates on the same register must be ordered; this can be done, for example, by replacing $\mathsf{SendClock}_i$ by a Lamport Clock, that respects the order in which updates are validated, and using a lexicographic order on pairs $(\mathsf{cl}, \mathsf{k})$. Third, variable V_i must be replaced by a set of update operations, and so does the value contained in the messages. All in all, this greatly complexifies the algorithm, without changing the way concurrency is handled. This is why we only focus on collections of SWMR registers here.

3.4 Correctness

In order to prove that Algorithm 1 implements a sequentially consistent snapshot memory, we must show that two important properties are verified by all histories it admits. These two properties correspond to lemmas 2 and 3. In Lemma 2, we show that it is possible to totally order the sets of updates validated by two processes at different moments. This allows us to build a total order on all the operations. In Lemma 3, we prove that all update operations are eventually validated by all processes. This is important to ensure termination of snapshot operations, and to ensure that update operations can not be ignored forever. Before that, Lemma 1 expresses a central property on how the algorithm works: the fact that each correct process broadcasts a

message corresponding to each written value proposed to validation. Finally, Property 3 proves that all histories admitted by Algorithm 1 are sequentially consistent.

In the following and for each process p_i and local variable x_i used in the algorithm, let us denote by x_i^t the value of x_i at time t. For example, $\mathsf{ValClock}_i^0$ is the initial value of $\mathsf{ValClock}_i$. For arrays of n integers cl and cl', we also denote by $cl \leq cl'$ the fact that, for all i, $cl[i] \leq cl'[i]$ and $cl < cl'$ if $cl \leq cl'$ and $cl \neq cl'$.

Lemma 1. *If a message* $\mathtt{message}(\mathsf{v}, \mathsf{k}, \mathsf{t}, \mathsf{cl})$ *is broadcast by a correct process* p_i*, then each correct process* p_j *broadcasts a unique message* $\mathtt{message}(\mathsf{v}, \mathsf{k}, \mathsf{t}, \mathsf{cl}')$*.*

In the following, for all processes p_j *and pairs* (k, t)*, let us denote by* $M_j(k, t)$ *the message* $\mathtt{message}(\mathsf{v}, \mathsf{k}, \mathsf{t}, \mathsf{cl}')$ *and by* $CL_j(k, t) = \mathsf{cl}'$ *the stamp that* p_j *put in this message.*

Proof. Let p_i and p_j be two correct processes, and suppose p_i broadcasts a message $M_i(\mathsf{k}, \mathsf{t})$.

First, we prove that p_j broadcasts a message $M_j(\mathsf{k}, \mathsf{t})$. As p_i is correct, p_j will eventually receive the message sent by p_i. At that time, if $\mathsf{t} > \mathsf{ValClock}_j[\mathsf{k}]$, after the condition on line 13 and whatever its result, G_i contains a value g with $g.\mathtt{k} = \mathsf{k}$ and $g.\mathtt{t} = \mathsf{t}$. That g was inserted on line 13 (possibly after the reception of a different message), just after p_j sent a message $M_j(\mathsf{k}, \mathsf{t})$ at line 18. Otherwise, $\mathsf{ValClock}_j[\mathsf{k}]$ was incremented on line 26, when validating some g', that was added in G_j after p_j received a (first) message $M_l(g'.\mathtt{k}, g'.\mathtt{t})$, with $g'.\mathtt{k} = \mathsf{k}$ and $g'.\mathtt{t} = \mathsf{ValClock}_j[\mathsf{k}]$. Remark that, as FIFO reception is used, p_k sent message $M_\mathsf{k}(\mathsf{k}, \mathsf{t})$ before $M_\mathsf{k}(\mathsf{k}, \mathsf{ValClock}_j[\mathsf{k}])$, and all other processes only forward messages, p_j received message $M_l(\mathsf{k}, \mathsf{t})$ before $M_l(\mathsf{k}, \mathsf{ValClock}_j[\mathsf{k}])$, and at that time, $\mathsf{t} > \mathsf{ValClock}_j[\mathsf{k}]$, so the first case applies.

Now, we prove that p_i will broadcast no other message with the same k and t later. If $i = \mathsf{k}$, the message would be sent on line 8 or 29, just after $\mathsf{SendClock}_i$ is incremented, which would lead to a different t. Otherwise, the message would be sent on line 18, which would mean the condition of line 13 is false. As p_i broadcast a first message, a corresponding g was present in $Algog_i$, deleted on line 24, which would make the condition of line 12 to be false.

Lemma 2. *Let* p_i*,* p_j *be two processes and* t_i*,* t_j *be two time instants, and let us denote by* $\mathsf{ValClock}_i^{t_i}$ *(resp.* $\mathsf{ValClock}_j^{t_j}$*) the value of* $\mathsf{ValClock}_i$ *(resp.* $\mathsf{ValClock}_j$*) at time* t_i *(resp.* t_j*). We have either, for all* k*,* $\mathsf{ValClock}_i^{t_i}[k] \leq \mathsf{ValClock}_j^{t_j}[k]$ *or for all* k*,* $\mathsf{ValClock}_j^{t_j}[k] \leq \mathsf{ValClock}_i^{t_i}[k]$*.*

Proof. Let p_i, p_j be two processes and t_i, t_j be two instants. Let us suppose (by contradiction) that there exist k and k' such that $\mathsf{ValClock}_j^{t_j}[k] < \mathsf{ValClock}_i^{t_i}[k]$ and $\mathsf{ValClock}_i^{t_i}[k'] < \mathsf{ValClock}_j^{t_j}[k']$.

As $\mathsf{ValClock}_i$ is only updated on line 26, at some time $t_i^k \leq t_i$, there was $g_i^k \in G'$ with $g_i^k.\mathtt{k} = k$ and $g_i^k.t = \mathsf{ValClock}_i^{t_i}[k]$. According to line 22, we have $|\{l : g_i^k.\mathtt{cl}[l] < \infty\}| > \frac{n}{2}$ and according to lines 14 and 20, each finite field $g_i^k.\mathtt{cl}[l]$ corresponds to the reception of a message $M_l(k, \mathsf{ValClock}_i^{t_i}[k])$. Similarly, process p_j received messages $M_l(k', \mathsf{ValClock}_j^{t_j}[k'])$ from more than $\frac{n}{2}$ processes. Since the number of processes is n, the intersection of these two sets of processes is not empty.

Let p_c be a process that belongs to both sets, i.e. p_c broadcast messages $M_c(k, \mathsf{ValClock}_i^{t_i}[k])$ and $M_c(k', \mathsf{ValClock}_j^{t_j}[k'])$. Process p_c sent these two messages in a given order, let us say $M_c(k', \mathsf{ValClock}_j^{t_j}[k'])$ before $M_c(k, \mathsf{ValClock}_i^{t_i}[k])$ (the other case is symmetric). As $\mathsf{SendClock}_c$ is never decremented and it is incremented before all sendings, $CL_c(k', \mathsf{ValClock}_j^{t_j}[k']) < CL_c(k, \mathsf{ValClock}_i^{t_i}[k])$. Moreover, as the protocol uses FIFO ordering, p_i received the two messages in the same order.

According to line 26, $\mathsf{ValClock}_i$ can only increase, so $\mathsf{ValClock}_i^{t'_i}[k'] \leq \mathsf{ValClock}_i^{t_i}[k']$ and $\mathsf{ValClock}_i^{t_i}[k'] < \mathsf{ValClock}_j^{t_j}[k']$. It means that the condition on line 12 was true when p_i received

$M_c(k', \mathsf{ValClock}_j^{t_j}[k'])$. Then, after the execution of the condition starting on line 13 and whatever the result of this condition, there was a $g_i^{k'} \in \mathsf{G}_i$ with $g_i^{k'}.\mathtt{k} = k'$, $g_i^{k'}.\mathtt{t} = \mathsf{ValClock}_j^{t_j}[k']$ and $g_i^{k'}.\mathtt{cl}[c] = CL_c(k', \mathsf{ValClock}_j^{t_j}[k'])$.

At time t_i, if $g_i^{k'} \notin G_i$, it was removed on line 24, which means $\mathsf{ValClock}_i^t[k'] \geq g_i^{k'}.\mathtt{t} = \mathsf{ValClock}_j^{t_j}[k']$ by lines 25 and 26, which is absurd by our hypothesis. Otherwise, after line 23 was executed at time t_i^k, we have $g_i^k \in G'$ and $g_i^{k'} \notin G'$, which is impossible as $g_i^{k'}.\mathtt{cl}[c] \leq g_i^k.GCL[c]$.

This is a contradiction. Therefore $\mathsf{ValClock}_i^{t_i} \leq \mathsf{ValClock}_j^{t_j}$ or $\mathsf{ValClock}_j^{t_j} \leq \mathsf{ValClock}_i^{t_i}$.

Lemma 3. *If a message* $\mathtt{message}(v, i, t, t)$ *is sent by a correct process* p_i*, then beyond some time* t'*, for each correct process* p_j*,* $\mathsf{ValClock}_j^{t'}[i] \geq t$.

Proof. Let us suppose a message $M_i(i, t)$ is sent by a correct process p_i.

Let us suppose (by contradiction) that there exists a process p_j such that the pair (i, t) has an infinity of predecessors according to $\rightarrow_j^\star$. As the number of processes is finite, an infinity of these predecessors correspond to the same process, let us say $(k, t_l)_{l \in \mathbb{N}}$. As p_j is correct, p_k eventually receives message $M_j(i, t)$, which means an infinity of messages $m_k(k, t_l)$ were sent after p_k receives message $m_j(i, t)$, and for all of them, $(k, t_l) \rightarrow_i^\star (i, t) \rightarrow_i (k, t_l)$. Therefore, there exists a sequence $(k_1, t'_1) \rightarrow_i (k_2, t'_2) \rightarrow_i \cdots \rightarrow_i (k_m, t'_m)$ with $k_1 = k_m = k$ and $t'_m > t'_1$. Two cases are possible for (k_2, t'_2):

- If p_k received a message $M_x(k_2, t'_2)$ (from any p_x) before it sent $M_k(k, t'_1)$, then p_k also send $M_k(k_2, t'_2)$ before it sent $M_k(k, t'_1)$, and all processes received these messages in the same order (and possibly a message $M_x(k_2, t'_2)$ even before from another process), which is in contradiction with the fact that $(k, t'_1) \rightarrow_i (k_2, t'_2)$.
- Otherwise, there is an index l such that process p_k received a message $M_x(k_{l'}, t'_{l'})$ (from any p_x) for all $l' > l$ but not for $l' = l$, before it sent message $M_k(k, t'_1)$. Whether it finally sends it on line 8 or line 29, there was no $g \in \mathsf{G}_i$ corresponding to (k_m, t'_m) so, by lines 22-23, p_k received messages $M_x(k_{l'}, t'_{l'})$ for all $l' > l$, from a majority of processes p_x, and all of them sent $M_x(k_{l'}, t'_{l'})$ before $M_x(k_l, t'_l)$. As $(k_l, t'_l) \rightarrow_i (k_{l+1}, t'_{l+1})$ and FIFO reception is used, a majority of processes sent $M_x(k_l, t'_l)$ before $M_x(k_{l'}, t'_{l'})$. This is impossible as two majorities always have a non-empty intersection. Therefore, this case is also impossible.

Finally, for all correct processes p_j, there exists a finite number of pairs (k, t') such that $(k, t') \rightarrow_j (i, t)$. As p_j is correct, according to Lemma 1, p_j will eventually receive a message $M_x(k, t')$ for all of them from all correct processes, which are in majority. At the last message, on line 25, G' will contain a g with $g.\mathtt{k} = i$ and $g.\mathtt{t} = t$ and after it executed line 26, it will have $\mathsf{ValClock}_j[i] \geq t$. As $\mathsf{ValClock}_j[i]$ can only grow and what precedes is true for all j, eventually it will be true for all correct processes.

Finally, given Lemmas 2 and 3, it is possible to prove that Algorithm 1 implements a sequentially consistent snapshot memory (Proposition 3). The idea is to order snapshot operations according to the order given by Lemma 2 on the value of $\mathsf{ValClock}_i$ when they were made and to insert the update operations at the position where $\mathsf{ValClock}_i$ changes because they are validated. It is possible to complete this order into a linearization order, thanks to Lemma 3, and to show that the execution of all the operations in that order respects the sequential specification of the snapshot memory data structure.

Proposition 3. *All histories admitted by Algorithm 1 are sequentially consistent.*

Proof. Let H be a history admitted by Algorithm 1. For each operation op, let us define $op.clock$ as follows:

- If op is a snapshot operation done on process p_i, $op.clock$ is the value of $\mathsf{ValClock}_i$ when p_i executes line 11.
- If op is an update operation done on process p_i, let us remark that the call to op is followed by the sending of a message $\texttt{message}(v, i, cl_i, cl_i)$, either directly on line 8 or later on line 29 as lemma 3 prevents the condition of line 27 to remain false forever (in this case, the value v may be more recent from the one written in op). Let us consider the clock cl_i of the first such message sent by p_i. We pose $op.clock$ as the smallest value taken by variable $\mathsf{ValClock}_j$ for any j (according to the total order given by lemma 2) such that $op_i \leq op.clock[i]$ (such a clock exists according to lemma 3).

Let $\lessdot$ be any total order on all the operations, that contains the process order, and such that all operation has a finite past according to $\lessdot$ ($\lessdot$ is only used to break ties). We define the relation $\leq$ on all operations of H by $op \leq op'$ if

1. $op.clock < op'.clock$, or
2. $op.clock = op'.clock$, op is an update operation and op' is a snapshot operation, or
3. $op.clock = op'.clock$, op and op' are either two snapshot or two update operations and $op \lessdot op'$.

Let us prove that $\leq$ is a total order.

reflexivity: for all op, the third point in the definition is respected, as $\lessdot$ is a total order.

antisymmetry: let op, op' be two operations such that $op \leq op' \leq op$. We have $op.clock = op'.clock$, op and op' are either two snapshot or two update operations and, as $\lessdot$ is antisymmetric, $op = op'$.

transitivity: let op, op', op'' be three operations such that $op \leq op' \leq op''$. If $op.clock \leq op'.clock$ or $op'.clock \leq op''.clock$, then $op.clock \leq op''.clock$. Otherwise, $op.clock = op'.clock = op''.clock$. If the three operations are all update or all snapshot operations, $op.clock \lessdot op''.clock$ so $op.clock \leq op''.clock$. Otherwise, op is an update and op' is a snapshot so $op.clock \leq op''.clock$.

total: let op, op' be two operations. If $op.clock \neq op'.clock$, they are ordered according to lemma 2. Otherwise, they are ordered by one of the last two points.

Let us prove that $\leq$ contains the process order. Let op and op' be two operations that occurred on the same process p_i, on which op preceded op'. According to lemma 2, $op.clock$ and $op'.clock$ are ordered.

- If $op.clock < op'.clock$ then $op \leq op'$.
- Let us suppose $op.clock = op'.clock$. It is impossible that op is a read operation and op' is an update operation: as $\mathsf{SendClock}_i$ is always increased before p_i sends a message, $op.clock[i] < op'.clock[i]$. If op is an update operation and op' is a snapshot operation, then $op \leq op'$. In the other cases, $op \lessdot op'$ so $op \leq op'$.
- We now prove case $op.clock > op'.clock$ cannot happen. As above, it is impossible that o is a read operation and o' is an update operation. It is also impossible that op and op' are two read or two update operations because $\mathsf{ValClock}_i$ can only grow. Finally, if op is an update operation and op' is a snapshot operation, $op.clock[i] \leq op'.clock[i]$ thanks to line 10, and by definition of $op.clock$ for update operations, $op.clock \leq op'.clock$.

	Read		Write		Snapshot		Update	
	# messages	latency	# messages	latency	# messages	latency	# messages	latency
ABD [14]	$\mathcal{O}(n)$	4	$\mathcal{O}(n)$	2	$\sim$	$\sim$	$\sim$	$\sim$
ABD + AR [14,6]	$\sim$	$\sim$	$\sim$	$\sim$	$\mathcal{O}\left(n^2 \log n\right)$	$\mathcal{O}\left(n \log(n)\right)$	$\mathcal{O}\left(n^2 \log n\right)$	$\mathcal{O}\left(n \log(n)\right)$
Algorithm 1	0	0 — 4	$\mathcal{O}(n^2)$	0	0	0 — 4	$\mathcal{O}(n^2)$	0

Fig. 6: Complexity of several algorithms to implement a shared memory.

Let us prove that all operations have a finite past according to $\leq$. Let op be an operation of the history. Let us first remark that, for each process p_i, $op.clock[i]$ corresponds to a message $M_i(i, op.clock[i])$. According to lemma 3, eventually, for all processes p_i, $\mathsf{ValClock}_i \geq op.clock$. Only a finite number of operations have been done before that, therefore $\{op' : op'.clock < op.clock\}$ is finite. Moreover, all the updates op' with $op'.clock \leq op.clock$ are done before that time, so there is a finite number of them. If op is an update operation, then its antecedents op' verify either $op'.clock < op.clock$ or $op'.clock = op.clock$ and op' is a write operation. In both cases, there is a finite number of them. If op is a snapshot operation, its antecedents op' verify either (1) $op'.clock < op.clock$, (2) $op'.clock = op.clock$ and op' is an update operation or (3) $op'.clock = op.clock$ and op' is a snapshot operation. Cases (1) and (2) are similar as above, and antecedents that verify case (3) also are its antecedents by $\lessdot$ so there is a finite number of them. Finally, in all cases, op has a finite number of antecedents.

Let us prove that the execution of all the operations in the order $\leq$ respects the sequential specification of memory. Let op be a snapshot operation invoked by process p_i and let p_j be a process. According to line 26, the value of $\mathsf{X}_i[j]$ corresponds to the value contained in a message $M_j(j, op.clock[j])$. Let op' be the last update operation invoked by process p_j before it sent this message. Whether the message was sent on line 8 or 29, $\mathsf{X}_i[j]$ is the value written by op'. Moreover, $op'.clock \leq op.clock$ so $op' \leq o'$ and for all update operations op'' done by process p_j after op', $op.clock < op''.clock$ so $op \leq op''$. All in all, op returns the last values written on each register, according to the order $\leq$.

Finally, $\leq$ defines a linearization of all the events of the history that respects the sequential specification of the shared object. Therefore, H is sequentially consistent.

3.5 Complexity

In this section, we analyze the algorithmic complexity of Algorithm 1 in terms of the number of messages and latency for snapshot and update operations. Fig. 6 sums up this complexity and compares it with the standard implementation of linearizable registers [14], as well as with the construction of a snapshot object [6] implemented on top of registers.

In an asynchronous system as the one we consider, the latency d and the uncertainty u of the network can not be expressed by constants. We therefore measure the complexity as the length of the longest chain of causally related messages to expect before an operation can complete. For example, if a process sends a message to another process and then waits for its answer, the complexity will be 2.

According to Lemma 1, it is clear that each update operation generates at most n^2 messages. The time complexity of an update operation is 0, as update operations return immediately. No message is sent for snapshot operations. Considering its latency, in the worst case, a snapshot operation is called immediately after two update operations a and b. In this case, the process must wait until its own message for a is received by the other processes, then to receive their acknowledgements, and then the same two messages must be routed for b, which leads to a complexity of 4. However, in the case of two consecutive snapshots, or if enough time has elapsed between a snapshot and the last update, the snapshot can also return immediately.

In comparison, the ABD simulation uses solely a linear number of messages per operation (reads as well as writes), but waiting is necessary for both kinds of operations. Even in the case of the read operation, our worst case corresponds to the latency of the ABD simulation. Moreover, our solution directly implements the snapshot operation. Implementing a snapshot operation on top of a linearizable shared memory is actually more costly than just reading each register once. The AR implementation [6], that is (to our knowledge) the implementation of the snapshot that uses the least amount of operations on the registers, uses $\mathcal{O}(n \log n)$ operations on registers to complete both a snapshot and an update operation. As each operation on memory requires $\mathcal{O}(n)$ messages and has a latency of $\mathcal{O}(1)$, our approach leads to a better performance in all cases.

Algorithm 1, like [14], uses unbounded integer values to timestamp messages. Therefore, the complexity of an operation depends on the number m of operations executed before it, in the linear extension. All messages sent by Algorithm 1 have a size of $\mathcal{O}\left(log(nm)\right)$. In comparison, ABD uses messages of size $\mathcal{O}\left(log(m)\right)$ but implements only one register, so it would also require messages of size $\mathcal{O}\left(log(nm)\right)$ to implement an array of n registers.

Considering the use of local memory, due to asynchrony, it is possible in some cases that G_i contains an entry g for each value previously written. In that case, the space occupied by G_i may grow up to $\mathcal{O}(mn \log m)$. Remark however that, according to Lemma 2, an entry g is eventually removed from G_i (in a synchronous system, after 2 time units if $g.\mathtt{k} = i$ or 1 time unit if $g.\mathtt{k} \neq i$). Therefore, this maximal bound is not likely to happen. Moreover, if all processes stop writing (which is the case in the round based model we discussed in Section 2.3), then eventually G_i becomes empty and the space occupied by the algorithm drops down to $\mathcal{O}(n \log m)$, which is comparable to ABD. In comparison, the AR implementation keeps a tree containing past values from all registers, in each register, which leads to a much higher size of messages and local memory.

4 Conclusion

In this paper, we investigated the advantages of focusing on sequential consistency. Because of its non composability, sequential consistency has received little focus so far. However, we show that in many applications, this limitation is not a problem. The first case concerns applications built on a layered architecture. If one layer contains only one object, then it is impossible for objects built on top of it to determine if this object is sequentially consistent or linearizable. The other example concerns round-based algorithms: if processes access to one different sequentially consistent object in each round, then the overall history is also sequentially consistent.

Using sequentially consistent objects instead of their linearizable counterpart can be very profitable in terms of execution time of operations. Whereas waiting is necessary for both read and write operations when implementing linearizable memory, we presented an algorithm in which waiting is only required for read operations when they follow directly a write operation. This extends the result of Attiya and Welch (that only concerns synchronous failure-free systems) to asynchronous systems with crashes. Moreover, the proposed algorithm implements a sequentially consistent snapshot memory for the same cost, which results in a better message and time comlexity, for both kinds of operations, than the best known implementation of a snapshot memory.

Exhibiting such an algorithm is not an easy task for two reasons. First, as write operations are wait-free, a process may write before its previous write has been acknowledged by other processes, which leads to "concurrent" write operations by the same process. Second, proving that an implementation is sequentially consistent is more difficult than proving it is linearizable since the condition on real time that must be respected by linearizability highly reduces the number of linear extensions that need to be considered.

5 Acknowledgments

This work has been partially supported by the Franco-German ANR project DISCMAT under grant agreement ANR-14-CE35-0010-01. The project is devoted to connections between mathematics and distributed computing.

References

1. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. Computers, IEEE Transactions on **100**(9) (1979) 690–691
2. Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems (TOPLAS) **12**(3) (1990) 463–492
3. Lipton, R.J., Sandberg, J.S.: PRAM: A scalable shared memory. Princeton University, Department of Computer Science (1988)
4. Attiya, H., Welch, J.L.: Sequential consistency versus linearizability. ACM Transactions on Computer Systems (TOCS) **12**(2) (1994) 91–122
5. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic snapshots of shared memory. J. ACM **40**(4) (1993) 873–890
6. Attiya, H., Rachman, O.: Atomic snapshots in o(n log n) operations. SIAM J. Comput. **27**(2) (1998) 319–340
7. Gafni, E.: Distributed Computing: a Glimmer of a Theory, in Handbook of Computer Science. CRC Press (1998)
8. Gafni, E.: Round-by-round fault detectors: Unifying synchrony and asynchrony (extended abstract). In: Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998. (1998) 143–152
9. Goodman, J.R.: Cache consistency and sequential consistency. University of Wisconsin-Madison, Computer Sciences Department (1991)
10. Awerbuch, B.: Complexity of network synchronization. J. ACM **32**(4) (1985) 804–823
11. Charron-Bost, B., Schiper, A.: The heard-of model: computing in distributed systems with benign faults. Distributed Computing **22**(1) (2009) 49–71
12. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM **35**(2) (1988) 288–323
13. Birman, K.P., Joseph, T.A.: Reliable communication in the presence of failures. ACM Transactions on Computer Systems (TOCS) **5**(1) (1987) 47–76
14. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in message-passing systems. Journal of the ACM (JACM) **42**(1) (1995) 124–142
15. Borowsky, E., Gafni, E.: Immediate atomic snapshots and fast renaming (extended abstract). In: Proceedings of the Twelth Annual ACM Symposium on Principles of Distributed Computing, Ithaca, New York, USA, August 15-18, 1993. (1993) 41–51
16. Dwork, C., Herlihy, M., Plotkin, S.A., Waarts, O.: Time-lapse snapshots. In: Theory of Computing and Systems. Springer (1992) 154–170
17. Kirousis, L.M., Spirakis, P.G., Tsigas, P.: Reading many variables in one atomic operation: Solutions with linear or sublinear complexity. IEEE Trans. Parallel Distrib. Syst. **5**(7) (1994) 688–696
18. Attiya, H., Herlihy, M., Rachman, O.: Atomic snapshots using lattice agreement. Distributed Computing **8**(3) (1995) 121–132
19. Riany, Y., Shavit, N., Touitou, D.: Towards a practical snapshot algorithm. Theor. Comput. Sci. **269**(1-2) (2001) 163–201