Advance Access publication on November 3, 2009
doi:10.1093/comjnl/bxp090

A Methodological Construction of an Efficient Sequentially Consistent Distributed Shared Memory[1]

VICENT CHOLVI[1,*], ANTONIO FERNÁNDEZ[2], ERNESTO JIMÉNEZ[3], PILAR MANZANO[3], AND MICHEL RAYNAL[4]

[1]*Universitat Jaume I, Castellón, Spain*
[2]*LADyR, GSyC, Universidad Rey Juan Carlos, 28933 Móstoles, Spain*
[3]*Universidad Politécnica de Madrid, 28031 Madrid, Spain*
[4]*IRISA, Université de Rennes, Campus de Beaulieu, 35 042 Rennes, France*
**Corresponding author: vcholvi@uji.es*

The paper proposes a simple protocol that ensures sequential consistency. The protocol assumes that the shared memory abstraction is supported by the local memories of nodes that can communicate only by exchanging messages through reliable channels. Unlike other sequential consistency protocols, the one proposed here does not rely on a strong synchronization mechanism, such as an atomic broadcast primitive or a central node managing a copy of every shared object. From a methodological point of view, the protocol is built incrementally starting from the very definition of sequential consistency. It has the noteworthy property that a process that issues a write operation never has to wait for other processes. Depending on the current local state, most read operations issued also have the same property.

Keywords: distributed systems; distributed shared memory; sequential consistency

Received 10 September 2008; revised 25 September 2009
Handling editor: Peter Harrison

1. INTRODUCTION

The definition of a consistency criterion is crucial for the correctness of a multiprocess program [2, 3]. Basically, a consistency criterion defines which value has to be returned when a read operation on a shared object is invoked by a process [4–6]. The strongest (i.e. most constraining) consistency criterion is *atomic consistency* [7], also called *linearizability* [8]. It states that a read returns the value written by the latest preceding write, 'latest' referring to real-time occurrence order (concurrent writes being totally ordered). *Causal consistency* [9, 10] is a weaker criterion, stating that a read does not get an overwritten value. Causal consistency allows concurrent writes; consequently, it is possible that concurrent read operations on the same object get different values. This occurs when those values have been produced by concurrent writes. Other consistency criteria weaker than causal consistency have also been proposed [11, 12].

This paper focuses on *sequential consistency* [13]. This criterion lies between atomic consistency and causal consistency. Informally, it states that a multiprocess program executes correctly if its results could have been produced by executing that program on a single processor system. This means that an execution is correct if we can totally order its operations in such a way that:

(i) the order of operations in each process is preserved;
(ii) each read operation obtains the latest previously written value, 'latest' referring here to the total order.

The difference between atomic consistency and sequential consistency lies in the meaning of the word 'latest'. This word refers to real-time when we consider atomic consistency, while it refers to a logical time notion when we consider sequential consistency. Namely, the logical time defined by the total order. The main difference between sequential consistency and causal consistency lies in the fact that, like atomic consistency, sequential consistency orders all write operations,

[1]A preliminary version of this paper was published in [1].

while causal consistency does not require the ordering of concurrent writes.

Atomic consistency is relatively easy to implement in a distributed message-passing system. Each process p_i maintains in a local cache the current value v of each shared variable x, and such a cached value v is systematically invalidated (or updated) each time a process p_j writes x. The conflicts due to multiple accesses to a shared variable x are usually handled by associating a manager M_x with every shared variable x. One of the most popular atomic consistency protocols is the invalidation-based protocol due to Li and Hudak [14] that has been designed to provide a distributed shared memory on top of a local area network. An update-based atomic consistency protocol is described in [15].

Due to its very definition, atomic consistency requires that the value of a variable x cached at p_i be invalidated (or updated) each time a process p_j issues a write on x. In that sense, the atomic consistency criterion (that is an abstract property of a computation) is intimately related to an *eager* invalidation (or update) mechanism that concerns the operational side. Said in another way, atomic consistency is a consistency criterion that can be too *conservative* for some applications.

Put in another way, sequential consistency can be seen as a form of *lazy* atomic consistency [16]. A cached value does not need to be systematically invalidated each time the corresponding shared variable is updated. Old and new values of a shared variable can coexist at different processes as long as the resulting execution could have been produced by running the multiprocess program on a single multiprogrammed processor system. Of course, a protocol implementing sequential consistency can be more involved than a protocol implementing atomic consistency, as it has to keep track of global information allowing it to know, for each process p_i, which old values currently used by p_i have to be invalidated (or updated). This global information tracking, which is at the core of sequential consistency protocols, is the additional price that has to be paid to replace eager invalidation by lazy invalidation, thereby providing the possibility for more efficient runs of multiprocess programs.

This paper presents a methodological construction of a sequential consistency protocol. A variant of this protocol has first been presented in [17] as a dynamically adaptive and parameterized algorithm that implements sequential consistency, cache consistency or causal consistency, according to the setting of some parameter. This parameterized algorithm is presented 'from scratch', without exhibiting or relying on basic underlying principles. Here, it is shown that a variant of its sequential consistency instantiation can be obtained from a simple derivation starting from the very definition of sequential consistency.

The algorithm obtained here from this derivation not only is surprisingly simple, but—as it is based on the very essence of sequential consistency—it reveals to be particularly efficient for some classes of applications. The protocol has the nice property to allow the write operations to be always executed locally without involving external synchronization. Alternatively, some read operations can be executed in the same fashion, while others cannot. Whether a read is executed locally depends on the variable that is read and the set of variables that have been previously written by the process issuing the read operation, so it is context-dependent.

The derived algorithm has been implemented and used to run typical parallel programming applications, namely finite differences (FD), matrix multiplication (MM), and fast Fourier transform (FFT), in a cluster of workstations. In this context, the performance of this implementation of the algorithm has been compared with implementations of the sequential consistency algorithms proposed by Attiya and Welch [18]. The results of this comparison show that the implementation of our algorithm runs faster and requires smaller number of messages than the other two. Furthermore, unlike the algorithms from [18], with our algorithm a large majority of the messages carry information about written values.

The paper is made up of six sections. Section 2 presents some related work. Section 3 presents the computation model, and defines sequential consistency. Then, Section 4 derives the protocol from the sequential consistency definition. Section 5 provides a performance evaluation of such a protocol. Finally, Section 6 concludes the paper.

2. RELATED WORK

Several protocols providing a sequentially consistent shared memory abstraction on top of an asynchronous message passing distributed system have been proposed. The protocol described in [19] implements a sequentially consistent shared memory abstraction on top of a physically shared memory and local caches. It uses an atomic n-queue update primitive. Attiya and Welch [18] present two sequential consistency protocols. Both protocols assume that each local memory contains a copy of the whole shared memory abstraction. They order the write operations using an atomic broadcast facility: all the writes are sent to all processes and are delivered in the same order by each process. Read operations issued by a process are appropriately scheduled to ensure their correctness.

The protocol described in [20] considers a server site that has a copy of the whole shared memory abstraction. The local memory of each process contains a copy of a shared memory abstraction, but the state of some of its objects can be invalid. When a process wants to read an object, it reads its local copy if it is valid. When a process wants to read an object whose state is invalid, or wants to write an object, it sends a request to the server. In this way, the server orders all write operations. An invalidation mechanism ensures that the reading by p_i of an object that is locally valid is correct. A variant of this protocol is described in [21]. The protocol described in [22] uses a token that orders all write operations and piggybacks updated values. This protocol, like one of the protocols described in [18], provides purely local read operations [23].[1]

Most of the previous protocols rely on a strong synchronization mechanism that has a scope spanning the whole system (atomic broadcast facility, navigating token or central manager[2]). However, the protocol described in [16] is fully distributed in the sense that it does not rely on an underlying global mechanism: each object x is managed by its own object manager M_x and there is no synchronization primitive whose scope is the entire system.

3. THE SEQUENTIALLY CONSISTENT SHARED MEMORY ABSTRACTION

A parallel program defines a set of processes interacting through a set of concurrent objects. This set of shared objects defines a *shared memory abstraction*. Each object is defined by a sequential specification and provides processes with operations to manipulate it. When it is running, the parallel program produces a concurrent system [8]. As in such a system an object can be accessed concurrently by several processes, it is necessary to define consistency criteria for concurrent objects.

3.1. Shared memory abstraction

A shared memory system is composed of a finite set of sequential processes $p_1, \ldots, p_n$ that interact via a finite set X of shared objects. Each object $x \in X$ can be accessed by read and write operations. A write into an object defines a new value for the object; a read allows to obtain a value of the object. A write of value v into object x by process p_i is denoted by $w_i(x)v$; similarly, a read of x by process p_j is denoted by $r_j(x)v$ where v is the value returned by the read operation; op will denote either r (read) or w (write). To simplify the analyses, as in [7, 9, 25], we assume that all values written into an object x are distinct.[3] Moreover, the parameters of an operation are omitted when they are not important. Each object has an initial value (it is assumed that this value has been assigned by an initial fictitious write operation).

3.2. Programs, histories and legality

A *program* is a set of read and write operations to be issued by the processes that form the program. The *local program* of process p_i is the set of operations to be issued by p_i. If $op1$ and $op2$ are going to be issued by p_i and $op1$ is going to be issued first, then we say that '$op1$ precedes $op2$ in p_i's process–order', which is denoted by $op1 \rightarrow_i op2$. Note that nothing has been said about the read or written values, nor about the order between operations from different processes.

In order to model concrete executions of programs, we introduce the concept of history. The *local history* (or local computation) $\hat{h}_i$ of p_i is the sequence of operations issued by p_i in process order such that each operation has an associated (read or written) value. If h_i denotes the set of operations executed by p_i, then $\hat{h}_i$ is the total order $(h_i, \rightarrow_i)$.

DEFINITION 3.1. *An* execution history (*or simply history, or computation*) $\hat{H}$ *of a shared memory system is a partial order* $\hat{H} = (H, \rightarrow_H)$ *such that:*

(i) $H = \bigcup_i h_i$;
(ii) $op1 \rightarrow_H op2$ *if*:

  (a) $\exists p_i : op1 \rightarrow_i op2$ (*in that case*, $\rightarrow_H$ *is called* process–order *relation*), *or*
  (b) $op1 = w_i(x)v$ *and* $op2 = r_j(x)v$ (*in that case* $\rightarrow_H$ *is called* read-from *relation*), *or*
  (c) $\exists op3 : op1 \rightarrow_H op3$ *and* $op3 \rightarrow_H op2$.

Two operations $op1$ and $op2$ are *concurrent* in $\hat{H}$ if we have neither $op1 \rightarrow_H op2$ nor $op2 \rightarrow_H op1$. Table 1 shows some of the nomenclature used.

The legality concept is the key notion on which the definitions of shared memory consistency criteria are based [9, 10, 12, 24]. From an operational point of view, it states that, in a legal history, no read operation can get an overwritten value.

DEFINITION 3.2. *A read operation* $r(x)v$ *of a history* $\hat{H}$ *is* legal *if*:

(i) $\exists w(x)v : w(x)v \rightarrow_H r(x)v$;
(ii) $\nexists op(x)u : (u \neq v) \wedge (w(x)v \rightarrow_H op(x)u \rightarrow_H r(x)v)$.

A history $\hat{H}$ *is* legal *if all its read operations are legal.*

TABLE 1. Nomenclature.

Symbol	Description
p_i	Sequential process i
$r_i(x)v$	Read of value v of object x by process p_i
$w_i(x)v$	Write of value v into object x by process p_i
$op \rightarrow_i op'$	Operation op precedes operation op' in p_i
$op \rightarrow_H op'$	Operation op causally precedes operation op'
h_i	Set of operation executed by p_i
$\hat{h}_i$	Total order $(h_i, \rightarrow_i)$
H_i	$\bigcup_i h_i$
$\hat{H}_i$	Total order $(H_i, \rightarrow_H)$

[1]As shown in [18], atomic consistency does not allow protocols in which all read operations (or all write operations) can be executed locally without involving global synchronization[8, 24]. Alternatively, causal consistency allows protocols where this happens [9, 10, 25].

[2]For example, an atomic broadcast facility allows ordering all the write operations, independently of the processes that issue them.

[3]Intuitively, this hypothesis can be seen as an implicit tagging of each value by a pair composed of the identity of the process that issued the write plus a sequence number. Such a tagging is only conceptual and not required for the correctness of the algorithm.

3.3. Sequential consistency

Sequential consistency was proposed by Lamport in 1979 to define a correctness criterion for multiprocessor shared memory systems [13]. A system is sequentially consistent with respect to a multiprocess program if *'the result of any execution is the same as if (1) the operations of all the processors were executed in some sequential order, and (2) the operations of each individual processor appear in this sequence in the order specified by its program.'*

This informal definition states that the execution of a program is sequentially consistent if it could have been produced by executing this program on a single processor system.[4] More formally, we define sequential consistency in the following way. We first recall the definition of *linear extension* of a partial order. A linear extension $\hat{S} = (S, \rightarrow_S)$ of a partial order $\hat{H} = (H, \rightarrow_H)$ is a topological sort of this partial order. This means that it satisfies the following:

(i) $S = H$;
(ii) $op_1 \rightarrow_H op_2 \Rightarrow op_1 \rightarrow_S op_2$ ($\hat{S}$ maintains the order of all ordered pairs of $\hat{H}$);
(iii) $\rightarrow_S$ defines a total order.

DEFINITION 3.3. *A history* $\hat{H}$ *is* sequentially consistent *if it has a legal linear extension* $\hat{S}$. *We also say that* $\hat{S}$ *is a* base legal sequentially consistent history *of* $\hat{H}$.

As an example, we consider the history $\hat{H}$ depicted in Fig. 1. Each process has issued three operations on the shared objects x and y. The write operations $w_1(x)0$ and $w_2(x)1$ are concurrent. It is easy to see that $\hat{H}$ is sequentially consistent by building a legal linear extension $\hat{S}$ including first the operations issued by p_2 and then the ones issued by p_1.

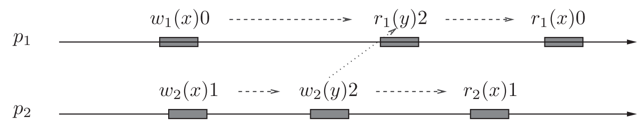


FIGURE 1. A sequentially consistent execution $\hat{H}$. Transitivity edges come from *process–order* relations (represented by dashed arrows) and *read–from* (represented by dotted arrows) relations. Only the edges that are not due to transitivity are shown.

4. THE METHODOLOGICAL CONSTRUCTION

The aim of this work is to implement a sequentially consistent shared memory abstraction on top of an underlying message-passing distributed system. Such a system is a distributed system made up of n reliable sites, one per process. Hence, without ambiguity, p_i denotes both a process and the associated site. Each p_i has a local memory. The processes communicate through reliable channels by sending and receiving messages. There are no assumptions neither on process speed, nor on message transfer delay. Hence, the underlying distributed system is reliable but asynchronous.

4.1. The methodology

The usual approach to design sequential consistency protocols consists on first defining a protocol and then proving it is correct. The approach adopted here is different, in the sense that we start from the very definition of sequential consistency, and *derive* from it a sequential consistency protocol.

More precisely, to ensure that a distributed execution has a base legal sequentially consistent history, we perform the following steps.

(i) First define a base legal sequentially consistent history $\hat{S}$.
(ii) Then, design a protocol that controls the execution of the multiprocess program in order to produce an actual distributed execution $\hat{H}$ that has $\hat{S}$ as a base legal sequentially consistent history.

The first subsection that follows derives a trivial sequential consistency protocol that works for a very particular type of multiprocess programs; these particular multiprocess programs have the nice property that all operations can be executed locally. Then, by observing that the history of each sequentially consistent process can be decomposed into segments, such as those considered in the previous type of multiprocess programs, a new sequential consistency protocol is derived that works for the general case. Finally, the last subsection shows how to enhance such a general protocol in order to achieve higher performance. The key idea behind the above-mentioned algorithms is disseminating updates only at the end of the different segments into which the distributed execution is decomposed. This solution reduces the necessary number of messages used to guarantee sequential consistency, thus improving the overall system performance.

4.2. Step 1 of the construction: the trivial case

We start with a multiprocess program where the local program of each process p_i has the following very particular structure. Namely, it is formed by a (possibly empty) sequence containing only read operations (denoted as SR_i), followed by a (possibly empty) sequence of write and read operations (denoted as SWR_i) such that the read operations are issued only on variables that have been previously written by p_i. Note that SWR_i ends when p_i stops issuing operations.

Consider a concrete execution $\hat{H}$ of such a program, produced by executing sequentially the SR_i sequences in any order, and

[4] In his definition, Lamport assumes that the *process–order* relations defined by the program (point 2 of the definition) is maintained in the equivalent sequential execution, but not necessarily in the execution itself. As we do not consider programs, but only executions, we implicitly assume that the *process–order* relations displayed by the execution histories are the ones specified by the programs which gave rise to these execution histories.

then the SWR_i sequences also in any order, and make each read operation return the closest previously written value, by the same process, in the corresponding variable (or the initial value, if it has not written any value). Since the SR_i sequences contain only read operations that obtain the initial values of the shared variables and the read operations in the SWR_i sequences read only variables previously written by process p_i, from the very definition of sequential consistency, it is immediate to find a base legal sequentially consistent history $\hat{S}$ of $\hat{H}$. Namely,

$$\hat{S} = SR_1 \ldots SR_n\ SWR_1 \ldots SWR_n. \tag{1}$$

Figure 2 shows an example of a parallel execution of a program made of $n = 3$ processes, as described in the above paragraph. Our goal now is to design a protocol that ensures that any history of the multiprocess programs considered is of the previously defined form (i.e. has $\hat{S}$ as a base legal sequentially consistent history).

4.2.1. *Implementation of the trivial case protocol*

From the above presented reasoning, it follows that an implementation would simply provide each process p_i with a local cache containing all the shared variables and perform

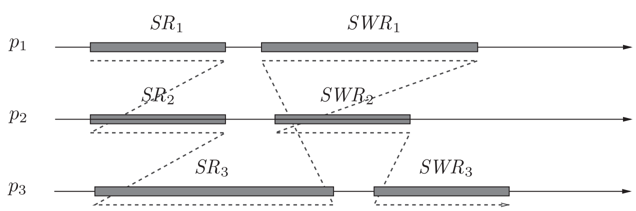


FIGURE 2. Example of a program execution $\hat{H}$ that is 'trivially' sequentially consistent. Reads are assumed to return the closest previously written value, by the same process, in the corresponding variable (or the initial value, if it has not been written yet). The ordering of the base legal sequentially consistent history $\hat{S}$ is indicated with the dashed arrow.

both reads and writes locally. Clearly, all the resulting histories will have the above presented $\hat{S}$ as a base legal sequentially consistent history. Consequently, no additional protocol would be necessary. Figure 3 shows an implementation of the described protocol.

4.3. Step 2 of the construction: the general case (looking for correctness)

We first observe that, in the general case, the local program of p_i (for each process) can always be expressed as follows:

$$SR_i^0\ SWR_i^1\ SR_i^1\ SWR_i^2\ SR_i^2 \ldots SWR_i^k\ SR_i^k \ldots,$$

where SR_i^k is a (possibly empty) sequence of only read operations and SWR_i^k is a (possibly empty) sequence of write and read operations such that read operations are performed only on variables that have been previously written in SWR_i^k. Note that SWR_i^k ends immediately before there is a read operation, by process p_i, on a variable not previously written in SWR_i^k. The superscript k is used to associate a SR_i sequence with its immediately preceding SWR_i sequence.

The decomposition of each process history into sequences and the particular case of a single sequence examined in the previous step of the construction, provides us with some hint on how to proceed. Indeed, we define a history $\hat{S}$ formed first by the sequences $SR_1^0, SR_2^0, \ldots, SR_n^0$, in this order, and then by the sequences $\boxed{SWR_1^1\ SR_1^1}, \boxed{SWR_2^1\ SR_2^1}, \ldots, \boxed{SWR_n^1\ SR_n^1}$, in this order. We find that $\hat{S}$ will contain additional subsequent phases, similar to the second one, until completing the execution. Also, make read operations to return the closest previously written value, by any process, in the corresponding variable (or the initial value, if it has not been written yet).

Clearly, for the definition of sequential consistency, $\hat{S}$ will be a base legal sequentially consistent history of 'some' of the histories of the general program. Figure 4a shows an example,

```
init:
   for each x ∈ X do
      C_i[x] ← initial value of x; % C_i[x] denotes a local cache associated with each shared variable x %
   end do

operation w_i(x)v:
   C_i[x] ← v;
   return()

operation r_i(x):
   return(C_i[x])
```

FIGURE 3. Trivial case protocol for process p_i.

in the case where there are $n = 3$ processes, of the parallel execution of a program $\hat{H}$ that has the above defined history $\hat{S}$ as a base legal sequential history.

Now, the goal is to design a sequential consistency protocol that ensures that 'any' possible program execution has $\hat{S}$ as a base legal sequentially consistent history.

4.3.1. *Implementation of the general case protocol*

For the design of the protocol, we observe that, in $\hat{S}$, when p_{i+1} executes SR^1_{i+1}, it can read the value of a variable x that has been written by p_i when it executed SWR^1_i. Hence, p_{i+1} must be informed of these writes before it executes SR^1_{i+1}. A simple way to attain this goal consists of using a token traveling along a logical ring, so that no process misses updates (e.g. $p_1, p_2, \ldots, p_n, p_1$) and carrying the latest known value of each shared variable. Therefore, we have to manage the token exactly as if it was received by p_{i+1} just after p_{i+1} executed SR^0_{i+1} and was sent by p_{i+1} to p_{i+2} just after p_{i+1} terminated SR^1_{i+1}. Logically, the token follows the dashed arrow in Fig. 4a, so that $\hat{H}$ will have $\hat{S}$ as a base legal sequentially consistent history. Then, in the algorithm to carry the new values written in SWR^1_i, the token has to be sent after SWR^1_i finishes. Moreover, as SR^1_i modifies no shared variables, the token can be sent by p_i before SR^1_i. So, when a process p_i receives the token, it ends a segment SWR^k_i, sends the token and starts a segment SR^k_i.

The resulting protocol is described in Fig. 5. As already indicated, X denotes the set of shared variables, and $C_i[x]$ is p_i's local cache containing the value of the shared variable x. Each process p_i maintains a boolean array $updated_i$ such that $updated_i[x]$ is true if and only if p_i has updated x since the last visit of the token. The boolean no_change_i is a synonym for $\wedge_{x\in X}(\neg updated_i[x])$ (no_change_i is true if and only if no shared variable has been updated since the last visit of the token at p_i). The write operation and the statements associated with the token reception are executed atomically. We observe that the arrival of the token at a process always corresponds to the beginning of a new segment SR^k_i for that process.[5] Figure 4b shows the actual travel of the token with this algorithm in the example used in this step. Observe that, in this protocol the token could be replaced by a list containing only the modifications. This 'improvement', together with one dealing with the dissemination of updates, is incorporated in the algorithm presented in the next step.

[5]The reader familiar with token-based termination detection protocols [26] can see that the protocol described in Fig. 5 and these termination detection protocols share the same underlying mechanism combining token and flags (here, the flags no_change_i). The corresponding flags in a termination detection protocol are usually called $cont\_passive_i$, and are used to know if a process p_i stayed continuously passive between two consecutive visits of the token. This flag is set to *false* when p_i receives a message. It is reset to *true* when p_i owns the token, becomes passive and sends the token to its successor.

4.4. Step 3 of the construction: the general case (looking for efficiency)

When we look at the form of the sequences SR^j_i, as defined in Step 2, we also observe that they can always be decomposed as follows:

$$SR^j_i = \begin{cases} SR^0_{i,1} \ldots SR^0_{i,i} & \text{when } j = 0, \\ SR^j_{i,i \pmod n)+1} \ldots SR^j_{i,(i+n-1) \pmod n)+1} & \text{when } j > 0. \end{cases}$$

Note that SR^0_i is decomposed into i sequences, whereas $SR^{j>0}_i$ is always decomposed into n sequences.

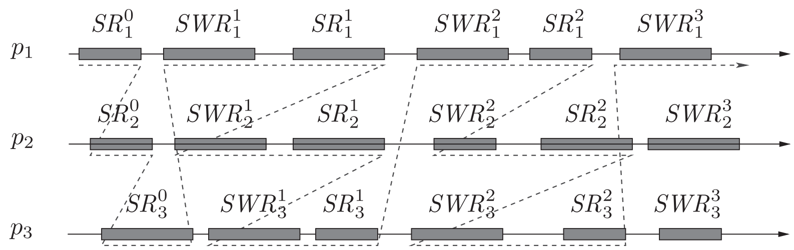
The rationale behind the form we have decomposed SR^j_i into n subsequences (except for the start-up phase, where it is split into i subsequences) can be explained as follows. By using such a decomposition, the goal is to allow a process p_i, during its sequence of reads in SR^j_i, to obtain the updated values as quickly as possible. Namely, those updates will take place at the beginning of each one of the $SR^j_{i,k}$ subsequences. With this in mind, in the new base legal sequentially consistent history $\hat{S}$, the updated values within SWR^j_i will have to be 'disseminated' to all processes at the same time, contrary to what is done at Step 2, where the updated values were disseminated sequentially. Clearly, this type of 'eager' dissemination allows processes to be informed of new values earlier. Furthermore, this also allows processes to disseminate only their own updates (in the protocol in Step 2, the token accumulates all the updates), thus reducing the transfer of data between processes.

Therefore, by substituting the new decomposed sequences into the local history of p_i, we obtain the following:

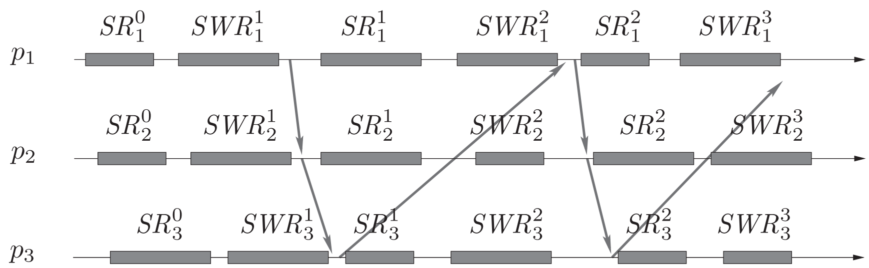
$$\begin{array}{l} SR^0_{i,1} \ldots SR^0_{i,i} \\ SWR^1_i \;\; SR^1_{i,i \pmod n)+1} \ldots SR^1_{i,(i+n-1) \pmod n)+1} \\ SWR^2_i \;\; SR^2_{i,i \pmod n)+1} \ldots SR^2_{i,(i+n-1) \pmod n)+1} \\ \vdots \\ SWR^j_i \;\; SR^j_{i,i \pmod n)+1} \ldots SR^j_{i,(i+n-1) \pmod n)+1} \\ \vdots \end{array}$$

Now, we define the form of the base legal sequentially consistent history $\hat{S}$. To do that, first we order the different sequences of the local programs as in Fig. 6.

Then, we make read operations to return the closest previously written value, by any process, in the corresponding variable (or the initial value, if it has not been written yet). Clearly, for the definition of sequential consistency, $\hat{S}$ will be a base legal sequentially consistent history of 'some' of the histories of the general program. Figure 7a shows an example, in the case where there are $n = 3$ processes, of the parallel execution of a program $\hat{H}$ that has the above defined history $\hat{S}$ as a base legal sequential history. Figure 7b also illustrates how the dissemination of writes is performed. Now, as in the



(a) Example of a program's execution $\widehat{H}$ that is sequentially consistent. The ordering of the base legal sequentially consistent history $\widehat{S}$ is indicated with the dashed arrow. Reads are assumed to return the closest previously written value (according to $\rightarrow_S$), by any process, in its corresponding variable (or the initial value, if it has not been written yet).



(b) Travel of the token in the implemented protocol. Observe that for the distributed execution $\widehat{H}$ to have $\widehat{S}$ as a base legal sequential history, the values carried by the token when it arrives at a process, say p_2, for the first time, have to be considered only if they have not been overwritten by SWR_2^1.

FIGURE 4. Example of a general case program's execution that is sequentially consistent.

previous cases, the goal is to design a sequential consistency protocol that ensures that 'any' possible program execution has $\hat{S}$ as a base legal sequentially consistent history.

4.5. Implementation of the efficient general case protocol

The design of the protocol is based on the protocol in Step 2. However, in order to dissociate the two different roles of the token (namely, dissemination and gathering of updates), the token itself is replaced by the local variables $token_i$. By $token_i = j$ we mean that, from p_i's point of view, p_j is the process that is currently allowed to disseminate updates. So, circulating the token around the logical ring, $p_1, p_2, \ldots, p_n, p_1, \ldots,$ is realized by having each $token_i$ variable taking successively the values $1, 2, \ldots, n, 1, \ldots$. Note that $token_i = i$ means that p_i (knows that it) has the token and is consequently allowed to disseminate updates.

The task associated with the management of the token is presented in Fig. 8. This task defines two distinct behaviors for a process p_i according to the token position. More precisely, when p_i has the token (case $token_i = i$), it is allowed to send to the rest of processes information about all the write operations (*updates*) it has executed since the previous visit of the token (Lines 3 and 4). This set of updates upd is carried in the message UPDATES(upd). After broadcasting its updates, p_i resets its local control variables (Lines 5 and 6).

When p_i does not have the token (case $token_i \neq i$), it waits for an UPDATES() message from the next process allowed to broadcast its updates (p_{token_i}). When it receives that message (Line 8), p_i updates accordingly its local cache (as in the previous protocol, Lines 9 and 10). This constitutes an early refreshing of its local cache with the new values provided by p_{token_i}.

Note that, for a process p_i, the token moves from p_j to p_{j+1} when, being $token_i$ equal to j, p_i executes $token_i \leftarrow$

init:
  for each $x \in X$ **do**
    $C_i[x] \leftarrow$ initial value of x;
    $updated_i[x] \leftarrow false$;
  end do;
  $no\_change_i \leftarrow true$;
  The token (with initial values) is initially at p_1 that simulates its arrival at the end of SWR_1^1

operation $w_i(x)v$: % $w_i(x)v$ always belongs to some segment SWR_i^z %
  $C_i[x] \leftarrow v$;
  $updated_i[x] \leftarrow true$;
  $no\_change_i \leftarrow false$;
  return()

operation $r_i(x)$:
  wait until $(no\_change_i \vee updated_i[x])$;
  % $no\_change_i \Rightarrow r_i(x) \in SR_i^z \wedge updated_i[x] \Rightarrow r_i(x) \in SWR_i^z$ %
  return $(C_i[x])$

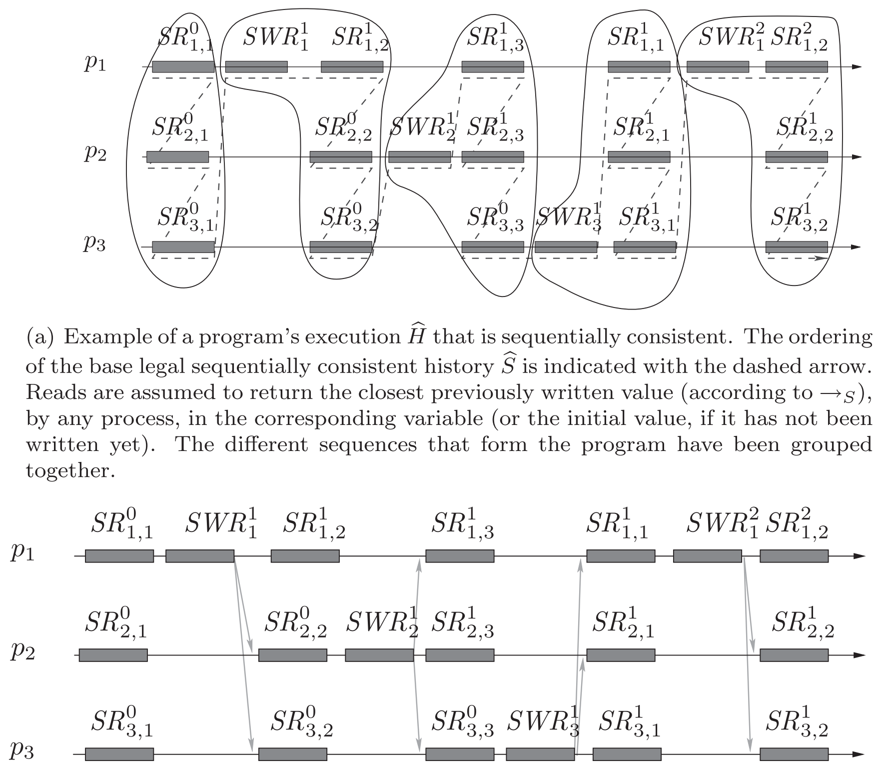
Task T_i**:** (activated upon reception of $token[X]$)
  for each $x \in X$ **such that** $\neg updated_i[x]$ **do**
    $C_i[x] \leftarrow token[x]$;
  end do;
  for each $x \in X$ **such that** $updated_i[x]$ **do**
    $token[x] \leftarrow C_i[x]$;
    $updated_i[x] \leftarrow false$;
  end do;
  send $token[X]$ **to** the next process on the logical ring;
  $no\_change_i \leftarrow true$;
  % we have here: $\forall x \in X: \ updated_i[x] = false$ %

FIGURE 5. General case protocol for process p_i.

$$
\begin{array}{l}
SR^0_{1,1}\ SR^0_{2,1}\ \ldots\ SR^0_{n,1} \\
SWR^1_1\ SR^1_{1,2}\ SR^0_{2,2}\ \ldots\ SR^0_{n,2} \\
SWR^1_2\ SR^1_{1,3}\ SR^1_{2,3}\ SR^0_{3,3}\ \ldots\ SR^0_{n,3} \\
\vdots \\
SWR^1_i\ SR^1_{1,i\,(mod\ n)+1}\ \ldots\ SR^1_{i,i\,(mod\ n)+1}\ SR^0_{i+1,i\,(mod\ n)+1}\ \ldots\ SR^0_{n,i\,(mod\ n)+1} \\
\vdots \\
SWR^1_n\ SR^1_{1,n\,(mod\ n)+1}\ SR^1_{2,n\,(mod\ n)+1}\ \ldots\ SR^1_{n,n\,(mod\ n)+1} \\
SWR^2_1\ SR^2_{1,1\,(mod\ n)+1}\ SR^1_{2,1\,(mod\ n)+1}\ \ldots\ SR^1_{n,1\,(mod\ n)+1} \\
SWR^2_2\ SR^2_{1,2\,(mod\ n)+1}\ SR^2_{2,2\,(mod\ n)+1}\ SR^1_{3,2\,(mod\ n)+1}\ \ldots\ SR^1_{n,2\,(mod\ n)+1} \\
\vdots \\
SWR^2_i\ SR^2_{1,i\,(mod\ n)+1}\ \ldots\ SR^2_{i,i\,(mod\ n)+1}\ SR^1_{i+1,i\,(mod\ n)+1}\ \ldots\ SR^1_{n,i\,(mod\ n)+1} \\
\vdots \\
SWR^2_n\ SR^2_{1,n\,(mod\ n)+1}\ SR^2_{2,n\,(mod\ n)+1}\ \ldots\ SR^2_{n,n\,(mod\ n)+1} \\
\vdots
\end{array}
$$

FIGURE 6. Efficient ordering of the sequences for the general case.

$(token_i \mod n) + 1$ (Line 13). All the processes have the same view of the order in which the token visits the processes. Consequently, after it has received and processed an UPDATES() message from p_j, the process p_{j+1} knows that it has the token: no explicit message is necessary to represent the token.



(a) Example of a program's execution $\widehat{H}$ that is sequentially consistent. The ordering of the base legal sequentially consistent history $\widehat{S}$ is indicated with the dashed arrow. Reads are assumed to return the closest previously written value (according to $\rightarrow_S$), by any process, in the corresponding variable (or the initial value, if it has not been written yet). The different sequences that form the program have been grouped together.

(b) Eager dissemination of the updates.

FIGURE 7. Example of an efficient general case program execution that is sequentially consistent.

It is important to note that all the processes update their local caches (with the new values coming from the other processes) in the same order. This is an immediate consequence of the fact that each process p_i delivers the UPDATES() messages in the order defined by the successive values of $token_i$. As in the base token-based protocol, p_i's own updates are done at the time p_i issues the corresponding write operations and tracked with the boolean array $updated_i$. These boolean flags are used to maintain the consistency of p_i's local cache each time it receives and processes an UPDATES() message.

5. PERFORMANCE EVALUATION

This section presents experiments that show the efficiency of the proposed protocol. The protocol described in Fig. 8 is denoted by *CFJR* in the following. First, we show that in our efficient general case protocol most of the operations are performed in a fast manner. An operation (either read or write) is said to be *fast* if it can be executed locally at the process where it is issued without involving global synchronizations. This is a nice property since a process has never to wait when it writes or reads a new value in a shared object. This implies that such operations can be served almost immediately.

Furthermore, the performance of *CFJR* is also compared with two sequential consistency protocols proposed by Attiya and Welch [18]. Such protocols are two of the most widely known sequential consistency protocols. In the first protocol proposed by Attiya and Welch (denoted as AW-$fast_r$), all read operations are fast while write operations are not fast. In the second one (denoted as AW-$fast_w$), all write operations are fast while read operations are not fast.

An exact analytic evaluation of how many read operations the protocol allows to be fast is not possible, as it depends on the read/write patterns of the upper layer distributed application. Hence, we have used real benchmark implementations to estimate the number of fast reads and, more generally, to evaluate the protocol performance. So, we have implemented three typical parallel processing applications:

(i) FD with 16 384 × 1024 elements,
(ii) MM with 1600 × 1600 matrices,
(iii) FFT with 262 144 coefficients

FD and MM have been implemented as in [27], while FFT as been implemented as in [28]. The code, written in C, uses the *sockets* interface with UDP/IP for computer intercommunication. The executions have been done in an experimental environment formed by a cluster of 2, 4 and

init:
  for each $x \in X$ **do**
    $C_i[x] \leftarrow$ initial value of x;
    $updated_i[x] \leftarrow false$;
  end do;
  $no\_change_i \leftarrow true$;
  $token_i \leftarrow 1$;

operation $w_i(x)v$: % $w_i(x)v$ always belongs to some segment SWR_i^z %
  $C_i[x] \leftarrow v$;
  $updated_i[x] \leftarrow true$;
  $no\_change_i \leftarrow false$;
  return()

operation $r_i(x)$:
  wait until $(no\_change_i \vee updated_i[x])$;
  % $no\_change_i \Rightarrow r_i(x) \in SR_i^z \wedge updated_i[x] \Rightarrow r_i(x) \in SWR_i^z$ %
  return $(C_i[x])$

Task T_i:
loop
  case $(token_i = i)$ **then**
      $upd = \{(x, C_i[x]) \mid updated_i[x]\}$;
      for each $j \neq i$ **do send** UPDATES(upd) **to** p_j **end do**;
      for each $(x, v_x) \in upd$ **do** $updated_i[x] \leftarrow false$ **end do**;
      $no\_change_i \leftarrow true$;
    $(token_i \neq i)$ **then**
      wait (UPDATES(upd) from $token_i$);
      for each $(x, v_x) \in upd$ **do**
        if $(\neg updated_i[x])$ **then** $C_i[x] \leftarrow v_x$ **end if**
      end do;
  end case;
  $token_i \leftarrow (token_i \mod n) + 1$;
end loop

FIGURE 8. Efficient general case protocol for process p_i.

8 computers connected with a switched full-duplex 1Gbps Ethernet network. Each computer is a PC running Linux Red-Hat with a 1.5 GHz AMD CPU and 512 Mbytes of RAM memory. We have mapped one process to each computer and have restricted our implementation to a maximum of 100 memory operations carried in one single message.

5.1. Percentage of fast operations in *CFJR*

In Table 2, the percentages of observed fast read and fast write operations per process in *CFJR* are shown. As it can be readily seen, all write operations are fast, while in all cases, almost 100% of the read operations are fast. This makes evident that the main goal of our protocol (i.e. to maximize the local operations) is certainly achieved.

5.2. Comparing *CFJR* with other protocols

In this section, we compare *CFJR* with AW-$fast_r$ and AW-$fast_w$. First, we compare the execution time measured with the three protocols for each one of the considered applications. As it can be readily seen in Table 3, whatever the case, the execution time provided by the *CFJR* protocol is much smaller than the execution time provided by both AW-$fast_r$ and AW-$fast_w$. In the case of FD, the execution time is up to 14.5 times lower; in the case of MM the execution time is up to 3.12 times lower and in the case of FFT the execution time is up to 27.5 times lower.

Table 4 presents the total number of messages and acknowledgments sent by each process when executing FD, MM, and FFT. By *acknowledgments* we mean all the messages sent to preserve the correct behavior of the protocol but without containing write operations. We can see that *CFJR* reduces

TABLE 2. Percentage of fast read and write operations per process in *CFJR*.

	Nodes (%)								
	2 nodes			4 nodes			8 nodes		
Operations	MM	FD	FFT	MM	FD	FFT	MM	FD	FFT
Reads	99.21	99.57	99.46	99.99	99.82	99.95	99.99	99.87	99.98
Writes	100	100	100	100	100	100	100	100	100

TABLE 3. Execution time of FD, MM and FFT (in seconds)

	FD			MM			FFT		
	2	4	8	2	4	8	2	4	8
CFJR	2228.3	2360.0	1450.8	3760.0	3307.5	2813.3	554.2	512.5	437.5
AW-$fast_r$	14133.3	19100.0	22591.7	4816.7	10346.7	8718.3	1371.7	14070.0	11304.2
AW-$fast_w$	12141.7	16400.0	21008.3	4348.3	9720.8	7512.5	1227.5	10215.8	9093.3

TABLE 4. Total number (in thousands) of messages + acknowledgments sent by each process.

	2	4	8
FD			
CFJR	2667/50	960/29	579/11
AW-$fast_r$	366361/190201	352321/264241	312321/308281
AW-$fast_w$	346613/170453	342284/254204	338782/294742
MM			
CFJR	3004/63	396/0.4	208/1.5
AW-$fast_r$	110400/51520	110080/76800	109847/89367
AW-$fast_w$	110080/51200	106587/73307	108239/87590
FFT			
CFJR	5206/3357	376/87	194/15
AW-$fast_r$	19922/4980	20970/8388	21068/9731
AW-$fast_w$	19546/4604	19766/7184	19559/8426

up to two orders of magnitude the total number of messages sent by each process. This is due to the fact that while *CFJR* allows several write operations to be disseminated in a single message (in our implementation, up to 100), both the AW-$fast_r$ and the AW-$fast_w$ protocols issue one message per write operation. Table 4 also show that in *CFJR*, almost each message contains write operations, unlike the AW-$fast_r$ and the AW-$fast_w$ protocols, where up to 50% of the messages are acknowledgments.

6. CONCLUSION

This paper has presented a new sequential consistency protocol. Unlike the previous protocols we are aware of, this one has been derived from the very definition of the sequential consistency criterion. Due to its design principles, the protocol we have obtained is particularly simple. Additionally, it provides write operations that can be executed locally (i.e. without requiring any form of global synchronization). Read operations can also be executed locally when they read a variable that has just been previously updated by the same process. The proposed protocol is very efficient in terms of achieving a high rate of memory operations that can be executed locally. Finally, we note that it is possible, from an engineering point of view, to adapt the globally efficient protocol to particular environments. A simple adaptation would consist in allowing some processes p_i to keep the token for some time when they have it. The benefit of such a possibility depends on the read/write access pattern of the upper layer application program.

FUNDING

This work has been partially supported by the Spanish MEC under grants TIN2008-03687, TIN2007-67353-C02-01, TIN2008-06735-C02-01 and PR2008-0015; Bancaixa Grant P1-1B2007-44; CAM Grant S-0505/TIC/0285; Comunidad de Madrid grant S-0505/TIC/0285.

REFERENCES

[1] Cholvi, V., Fernández, A., Jiménez, E. and Raynal, M. (2004) A methodological construction of an efficient sequential consistency protocol. *IEEE Int. Symp. Netw. Comput. Appl.* (NCA'04), pp. 141–148.

[2] Cholvi, V. and Bernabéu, J. (2004) Relationships between memory models. *Inf. Process. Lett.*, **90**, 53–58.

[3] Steinke, R.C. and Nutt, G.J. (2004) A unified theory of shared memory consistency. *J. ACM*, **51**, 800–849.

[4] Haldar, S. and Vidyasankar, K. (2007) On specification of read/write shared variables. *J. ACM*, **54**, 1–31.

[5] Higham, J. and Jackson, L. and Kawash, J. (2007) Specifying memory consistency of write buffer multiprocessors. *ACM Trans. Comput. Syst.*, **25**, 1–42.

[6] Manovit, C. and Hangal, S. (2005) Efficient Algorithms for Verifying Memory Consistency, *SPAA'05: Proc. 17th Annual ACM Symp. Parallelism in Algorithms and Architectures*, Las Vegas, NV, USA, pp. 245–252. ACM Press, New York, USA.

[7] Misra, J. (1986) Axioms for memory access in asynchronous hardware systems. *ACM Trans. Program. Lang. Syst.*, **8**, 142–153.

[8] Herlihy, M.P. and Wing, J.L. (1990) Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, **12**, 463–492.

[9] Ahamad, M., Hutto, P.W., Neiger, G., Burns, J.E. and Kohli, P. (1995) Causal memory: definitions, implementations and programming. *Distrib. Comput.*, **9** 37–49.

[10] Ahamad, M., Raynal, M. and Thia-Kime, G. (1998) An Adaptive Protocol for Implementing Causally Consistent Distributed Services. *Proc. 18th IEEE Int. Conf. Distributed Computing Systems*, Amsterdam, The Netherland, pp. 86–93. IEEE Computer Society Press.

[11] Adve, S.V. and Garachorloo, K. (1997) Shared memory models: a tutorial. *IEEE Comput.*, **29**, 66–77.

[12] Raynal, M. and Schiper, A. (1996) A Suite of Formal Definitions for Consistency Criteria in Distributed Shared Memories. *Proc. 9th Int. IEEE Conf. Parallel and Distributed Computing Systems* (*PDCS'96*), Dijon, France, pp. 125–131. IEEE Computer Society Press.

[13] Lamport, L. (1979) How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, **C28**, 690–691.

[14] Li, K. and Hudak, P. (1989) Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, **7**, 321–359.

[15] Bal, H. and Tanenbaum, A.S. (1992) ORCA: a language for parallel programming of distributed systems. *IEEE Trans. Softw. Eng.*, **18**, 190–205.

[16] Raynal, M. (2002) Sequential Consistency as Lazy Linearizability. *Proc. 14th ACM Symp. Parallel Algorithms and Architectures* (*SPAA'02*), Winnipeg, pp. 151–152.

[17] Jiménez, E., Fernández, A. and Cholvi, V. (2008) A parameterized algorithm that implements sequential, causal and cache memory consistencies. *J. Syst. Softw.*, **81**, 120–131.

[18] Attiya, H. and Welch, J.L. (1994) Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, **12**, 91–122.

[19] Afek, Y., Brown, G. and Merritt, M. (1993) Lazy caching. *ACM Trans. Program. Lang. Syst.*, **15**, 182–205.

[20] Mizuno, M., Raynal, M. and Zhou, J.Z. (1994) Sequential consistency in distributed systems. *Proc. Int. Workshop on Theory and Practice of Distributed Systems*, Dagsthul Castle, Germany, pp. 224–241. Lecture Notes in Computer Science 938, Springer.

[21] Ahamad, M. and Kordale, R. (1999) Scalable consistency protocols for distributed services. *IEEE Trans. Parallel Distrib. Syst.*, **10**, 888–903.

[22] Raynal, M. (2002) Token-based sequential consistency. *Int. J. Comput. Syst. Sci. Eng.*, **17**, 359–366.

[23] Herlihy, M.P. (1991) Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, **13**, 124–149.

[24] Mizuno, M., Nielsen, M.L. and Raynal, M. (1996) An optimistic protocol for a linearizable distributed shared memory system. *Parallel Process. Lett.*, **6**, 265–278.

[25] Raynal, M. and Schiper, A. (1995) From Causal Consistency to Sequential Consistency in Shared Memory Systems. *Proc. 15th Int. Conf. Foundations of Software Technology and Theoretical Computer Science* (*FST&TCS'95*), Bangalore, India, pp. 180–194. Lecture Notes in Computer Science 1026, Springer.

[26] Mattern, F. (1987) Algorithms for distributed termination detection. *Distrib. Comput.*, **2**, 161–175.

[27] Wilkinson, B. and Allen, M. (1999) *Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers*. Prentice-Hall.

[28] Akl, S.G. (1989) *The Design and Analysis of Parallel Algorithms*. Prentice-Hall.