A Methodological Construction of an Efficient Sequential Consistency Protocol*

Vicent CHOLVI° Antonio FERNÁNDEZ† Ernesto JIMÉNEZ‡ Michel RAYNAL⋆

° Universidad Jaume I, Castellón, Spain

† Laboratorio de Algoritmia Distribuida, Universidad Rey Juan Carlos, 28933 Móstoles, Spain

‡ EUI, Universidad Politécnica de Madrid, 28031 Madrid, Spain

⋆ IRISA, Université de Rennes, Campus de Beaulieu, 35 042 Rennes, France

vcholvi@lsi.uji.es anto@gsyc.escet.urjc.es ernes@eui.upm.es raynal@irisa.fr

Abstract

A concurrent object is an object that can be concurrently accessed by several processes. Sequential consistency is a consistency criterion for such objects. 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. (Sequential consistency is weaker than atomic consistency -the usual consistency criterion-as it does not refer to real-time.) The paper proposes a simple protocol that ensures sequential consistency when the shared memory abstraction is supported by the local memories of nodes that can communicate only by exchanging messages through reliable channels. Differently from other sequential consistency protocols, the proposed protocol 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 of providing fast writes operations (i.e., a process has never to wait when it writes a new value in a shared object). According to the current local state, some read operations can also be fast. An experimental evaluation of the protocol is also presented. The proposed protocol could be used to manage Web page caching.

1 Introduction

Sequential consistency The definition of a consistency criterion is crucial for the correctness of a multiprocess program. Basically, a consistency criterion defines which value has to be returned when a read operation on a shared object is invoked by a process. The strongest (i.e., most constraining) consistency criterion is *atomic consistency* [15] (also called *linearizability* [10]). It states that a read returns the value written by the last preceding write, "last" referring to real-time occurrence order (concurrent writes being ordered). *Causal consistency* [3, 5] 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 been proposed [1, 21].

This paper focuses on *sequential consistency* [12]. 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 (1) the order of operations in each process is preserved, and (2) each read obtains the last previously written value, "last" referring here to the total order. The difference between atomic consistency and sequential consistency lies in the meaning of the word "last". 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 (as atomic consistency) sequential consistency orders all write operations, while causal consistency does not require to order 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 known atomic consistency protocols is the invalidation-based protocol due to Li and Hudak [13] that has been designed to provide a dis-

*This work is partially supported by the CICYT under grant TIC2001-1586-C03-01 and TIC2001-1586-C03-02.

tributed shared memory on top of a local area network. An update-based atomic consistency protocol is described in [8].

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.

Differently, sequential consistency can be seen as a form of *lazy* atomic consistency [19]. A cached value has not 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 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) and which ones have not. 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 efficient runs of multiprocess programs.

Related work: Sequential consistency protocols. 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 [2] 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 [7] 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 [17] 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 that 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 [4]. The protocol described in [18] uses a token that orders all write operations and piggybacks updated values like one of the protocols described [7] it provides fast (i.e., purely local) read operations [9][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]). Differently, the protocol described in [19] 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.

Content of the paper. This paper presents a methodological construction of a sequential consistency protocol. A variant of this protocol has first been presented in [11], where 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, we show 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 we obtain from the 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 fast, i.e., a write operation is always executed locally without involving global synchronization. Differently, some read operations can be fast, while other cannot. The fact that a read operation is fast or not 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 paper is made up of five sections. Section 2 presents the computation model, and defines sequential consistency. Then, Section 3 derives the protocol from the sequential consistency definition. Section 4 presents experimental results that show the protocol performance. Finally, Section 5 concludes the paper.

2 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

[1] As shown in [7] atomic consistency does not allow protocols in which all read operations (or all write operations) are fast [10, 16]. Differently, causal consistency allows protocols where all operations are fast [3, 5, 20].

[2] E.g., an atomic broadcast facility allows ordering all the write operations, whatever the processes that issue them.

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 [10]. As in such a system an object can be accessed concurrently by several processes, it is necessary to define consistency criteria for concurrent objects.

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 $w_i(x)v$; similarly a read of x by process p_j is denoted $r_j(x)v$ 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 [3, 15, 20], we assume 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).

History concept Histories are introduced to model the execution of shared memory parallel programs. The *local history* (or local computation) $\widehat{h}_i$ of p_i is the sequence of operations issued by p_i. If $op1$ and $op2$ are issued by p_i and $op1$ is issued first, then we say "$op1$ precedes $op2$ in p_i's process-order", which is noted $op1 \rightarrow_i op2$. Let h_i denote the set of operations executed by p_i; the local history $\widehat{h}_i$ is the total order $(h_i, \rightarrow_i)$.

Definition 1 *An* execution history *(or simply history, or computation)* $\widehat{H}$ *of a shared memory system is a partial order* $\widehat{H} = (H, \rightarrow_H)$ *such that:*

- $H = \bigcup_i h_i$
- $op1 \rightarrow_H op2$ *if:*
  - i) $\exists\, p_i\ : op1 \rightarrow_i op2$ *(in that case,* $\rightarrow_H$ *is called* process-order *relation),*
  - *or* ii) $op1 = w_i(x)v$ *and* $op2 = r_j(x)v$ *(in that case* $\rightarrow_H$ *is called* read-from *relation),*
  - *or* iii) $\exists op3\ : op1 \rightarrow_H op3$ *and* $op3 \rightarrow_H op2$.

Two operations $op1$ and $op2$ are *concurrent* in $\widehat{H}$ if we have neither $op1 \rightarrow_H op2$ nor $op2 \rightarrow_H op1$.

Legality notion The legality concept is the key notion on which are based definitions of shared memory consistency criteria [3, 5, 16, 21]. From an operational point of view, it states that, in a legal history, no read operation can get an overwritten value.

[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.

Definition 2 *A read operation* $r(x)v$ *is* legal *if: (i)* $\exists\, w(x)v\ : w(x)v \rightarrow_H r(x)v$ *and (ii)* $\nexists\, op(x)u\ : (u \neq v) \wedge (w(x)v \rightarrow_H op(x)u \rightarrow_H r(x)v)$. *A history* $\widehat{H}$ *is* legal *if all its read operations are legal.*

Sequential consistency has been proposed by Lamport in 1979 to define a correctness criterion for multiprocessor shared memory systems [12]. 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. Let us first recall the definition of *linear extension* of a partial order. A linear extension $\widehat{S} = (S, \rightarrow_S)$ of a partial order $\widehat{H} = (H, \rightarrow_H)$ is a topological sort of this partial order. This means we have the following: (i) $S = H$, (ii) $op_1 \rightarrow_H op_2 \Rightarrow op_1 \rightarrow_S op_2$ ($\widehat{S}$ maintains the order of all ordered pairs of $\widehat{H}$) and (iii) $\rightarrow_S$ defines a total order.

Definition 3 *A history* $\widehat{H} = (H, \rightarrow_H)$ *is* sequentially consistent *if it has a legal linear extension.*

3 Methodological Construction of a Sequential Consistency Protocol

3.1 Underlying Distributed System

Our aim is to implement the 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.

3.2 The Methodology

The usual approach to design sequential consistency protocols consists in first defining a protocol and then proving it is correct. The approach we adopt 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

[4]In his definition, Lamport assumes that the *process-order* relation 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* relation displayed by the execution histories are the ones specified by the programs which gave rise to these execution histories.

$\widehat{H} = (H, \rightarrow_H)$ has an equivalent legal sequential history $\widehat{S} = (H, \rightarrow_S)$, we (1) first, define a base legal sequential history $\widehat{S}$, and (2) then, design a protocol that controls the execution of the multiprocess program in order to produce an actual distributed execution $\widehat{H}$ equivalent to the base history $\widehat{S}$.

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 sequential process can be decomposed in 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 efficiency.

3.3 Step 1 of the Construction: Considering a Trivial Case

Let us start with a multiprocess program where the sequential history $\widehat{h}_i$ of each process p_i has the following very particular structure; namely, $\widehat{h}_i$ is R_i^0 followed by WR_i^1 where R_i^0 is a (possibly empty) sequence containing only read operations, and WR_i^1 is a (possibly empty) sequence starting with a write operation and followed by write operations on any variable and read operations only on variables that have been previously written by p_i (i.e., if $r_i(x)$ appears in WR_i^1 then $w_i(x)$ appears previously in WR_i^1).

Figure 1 shows an example of program as described in the above paragraph (in order to facilitate the presentation and without loss of generality, the figures that illustrate the construction consider a multiprocess program made up of $n = 3$ processes).

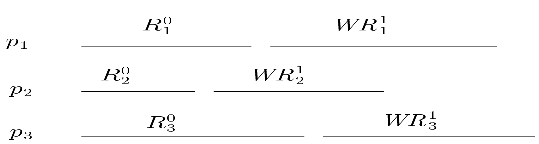

Figure 1. A (very) simple case

As we can see from the very definition of sequential consistency, the parallel execution described in Figure 1 could have been produced by executing sequentially, first R_1^0, R_2^0, and R_3^0 in any order (they contain only read operations that obtain the initial values of the shared variables), and then WR_1^1, WR_2^1, and WR_3^1 in any order (as any read operation appearing in WR_i^1 reads only variables that p_i has previously written)[5].

If follows that, if all the multiprocess programs had the structure previously described ($\widehat{h}_i$ being R_i^0 followed by WR_i^1), an implementation would simply consist in providing each process p_i with a local cache containing all the shared variables. No additional protocol would be necessary. So, we assume in the following that each process p_i has a local cache denoted $C_i[x]$ associated with each shared variable x.

[5] Let us observe that sequential consistency does not require that all the caches containing a copy of a shared variable x have to be equal at the end of the computation.

3.4 Step 2 of the Construction: (General Case) Looking for Correctness

Let us first observe that, in the general case, the history $\widehat{h}_i$ of a sequential process p_i can always be decomposed into consecutive "segments" (subsequences), each segment being of the form R_i^k or WR_i^k, namely ("," stands for "followed by"):

$$\widehat{h}_i = R_i^0 \; , \; WR_i^1 \; , \; R_i^1 \; , \; WR_i^2 \; , \; R_i^2 \; , \; \ldots \; , \; WR_i^k \; , \; R_i^k \; , \; \ldots$$

where, as before, R_i^k is a (possibly empty) sequence of only read operations, WR_i^k is a (possibly empty) sequence starting with a write operation and followed by write operations on any shared variable or read operations on shared variables previously written in WR_i^k. It is important to notice that, for $k > 0$, R_i^k starts (and consequently WR_i^k ends) when p_i reads a shared variable not written in WR_i^k.

Controlling a distributed execution will consist in two types of actions. The first concerns the safety of read operations, namely it consists in blocking the read of a shared variable x issued by a process p_i when the current value of $C_i[x]$ would produce a non-legal read. The second concerns liveness, namely it consists in propagating the new values to ensure that no read can block forever.

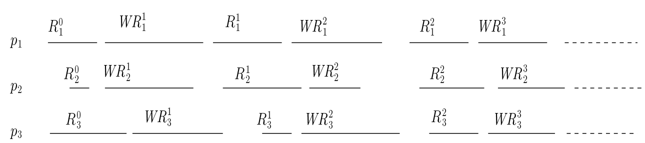

Figure 2. An execution

The decomposition of each process history into segments and the particular case of a single segment examined in Section 3.3, provides us with some hint on how to define a base legal sequential history $\widehat{S}$. Let us consider the execution described in Figure 2. A base legal sequential history $\widehat{S}$ that benefits from the segment decomposition of process histories can be the following:

$$\widehat{S} = R_1^0, R_2^0, R_3^0, \boxed{WR_1^1, R_1^1}, \boxed{WR_2^1, R_2^1}, \boxed{WR_3^1, R_3^1}, \boxed{WR_1^2, R_1^2},$$
$$\boxed{WR_2^2, R_2^2}, \boxed{WR_3^2, R_3^2}, \; \ldots$$

This base sequential execution can be produced by a mono-processor system whose scheduler provides the control first to p_1 to execute R_1^0, to p_2 to execute R_2^0, and to

Proceedings of the Third IEEE International Symposium on Network Computing and Applications (NCA'04)
0-7695-2242-4/04 $ 20.00 IEEE

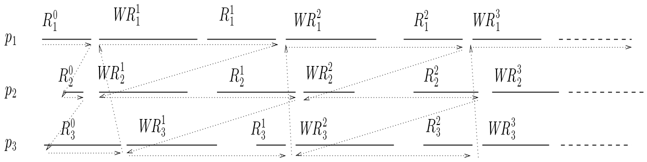

Figure 3. A base sequential execution $\widehat{S}$

p_3 to execute R_3^0, and then to p_1 to execute WR_1^1, R_1^1, then to p_1 to execute WR_2^1, R_2^1, etc. (This execution is indicated with the dotted arrows in Figure 3.)

As indicated, designing a sequential consistency protocol consists of ensuring that the actual distributed execution $\widehat{H}$ is equivalent to the base sequential execution $\widehat{S}$. Let us observe that, in $\widehat{S}$, when p_2 executes R_2^1, it can read the value of a variable x that has been written by p_1 when it executed WR_1^1. Hence, p_2 must be informed of these writes before it executes R_2^1. 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 last value it knows of each shared variable. To carry the new values written in WR_1^1, the token has to be sent after WR_1^1. Moreover, as R_1^1 modifies no shared variables, it can be sent by p_1 before R_1^1. So, when a process p_i has the token, it ends a segment WR_i^k, sends the token and starts a segment R_i^k.

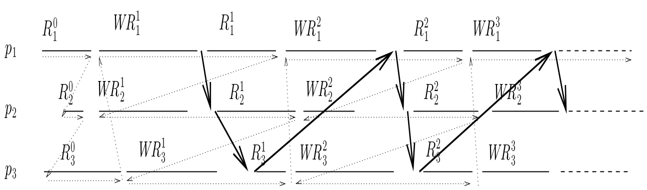

Figure 4. Using a token to disseminate the updates and prevent deadlock

The travel of the token is indicated by the bold arrows in Figure 4. Let us observe that for $\widehat{H}$ (the distributed execution) to be equivalent to $\widehat{S}$, the values carried by the token when it arrives at a process (say p_2 in Figure 4) have to be considered only if they have not been overwritten by WR_2^1. This means that we have to manage the token exactly as if it was received by p_2 just after p_2 has executed R_2^0 and was sent by p_2 to p_3 just after p_2 has terminated R_2^1: logically, the token follows the dotted arrows so that $\widehat{H}$ is equivalent to $\widehat{S}$.

The resulting protocol is described in Figure 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 iff p_i has updated x since the last visit of the token. The boolean no_change_i is a synonym for $\wedge_{y \in X}(\neg updated_i[y])$ (no_change_i is true iff 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. (Let us observe that the arrival of the token at a process always corresponds to the beginning of a new segment for that process.)[6]

init:
    for each $y \in X$ **do**
        $C_i[y] \leftarrow$ initial value of y;$updated_i[y] \leftarrow false$;
    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 WR_1^1;

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

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

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

Figure 5. Protocol for process p_i: token-based version

3.5 Step 3 of the Construction: (General Case) Looking for Efficiency

When we look carefully at the way the token is used in the previous protocol, we observe that it plays actually two distinct roles. On one side, when it is at a process p_i, the token gives p_i the right to disseminate the updates of the shared variables. That is the "control part" associated with the token: it provides an exclusive right to its current owner (a single process at a time can disseminate updates), and establishes an order among the processes to exploit this exclusive right. On another side, when it is sent by p_i to p_j, the token carries updates. That is the "communication" part

[6] The reader familiar with token-based termination detection protocols [14] can see that the protocol described in Figure 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.

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

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

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

Task T:
(1) **loop case** $(next_i = i)$ **then let** $upd = \{(y, C_i[y]) \mid updated_i[y]\}$;
(2)   **for each** $j \neq i$ **do** *send* UPDATES(*upd*) *to* p_j **end do**;
(3)   **for each** $(y, v_y) \in upd$ **do** $updated_i[y] \leftarrow false$ **end do**;
(4) $no\_change_i \leftarrow true$;
(5) $(next_i \neq i)$ **then wait** (UPDATES(*upd*) from $next_i$);
(6)   **for each** $(y, v_y) \in upd$ **do**
(7)   **if** $(\neg updated_i[y])$ **then** $C_i[y] \leftarrow v_y$ **end if**
(8)   **end do**;
(9) **end case**;
(10) $next_i \leftarrow (next_i \mod n) + 1$;
(11) **end loop**

Figure 6. Efficient protocol (for process p_i)

associated with the token. This section shows that it is possible to dissociate these two distinct roles to get a more efficient protocol.

Let us first introduce a local variable $next_i$, such that $next_i = i$ means that p_i (knows that it) has the token and is consequently allowed to disseminate updates. More generally, $next_i = j$ means that, from p_i's point of view, p_j is the process that is currently allowed to disseminate updates. So, circulating the token along the logical ring $p_1, p_2, \ldots, p_n, p_1, \ldots$, is realized by having each $next_i$ variable taking successively the values $1, 2, \ldots, n, 1, \ldots$

To dissociate the two roles of the token, the token itself is suppressed (as just indicated, it is replaced by the variables $next_i$) and the statement associated with its management is replaced by a task denoted T (see Figure 6). (The write and read operations and the task T are executed atomically.) This task defines two distinct behaviors for a process p_i according to the token role. More precisely, when p_i has the token (case $next_i = i$), it is allowed to send to the rest of processes all the updates it has done since the previous visit of the token (lines 1-2). These updates are carried by the message UPDATES(*upd*). After it has sent its updates, p_i resets its local control variables (lines 3-4).

There are two main differences with respect to the previous token-based protocol. First, a process broadcasts only its own updates, and second, this broadcast is done eagerly. (In the previous protocol, the token accumulates and disseminate the updates in a sequential way, following the logical ring.) This eager update dissemination, described in Figure 7, allows a process to be informed of new values earlier than what is done by the protocol of Figure 5 (in this figure, the UPDATES() messages "simulating" the token are described in bold arrows).

For a process p_i, the token passes from p_j to p_{j+1} when, $next_i$ being equal to j, p_i executes $next_i \leftarrow (next_i \mod n) + 1$ (line 10). 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_{i-1}, the process p_i knows that it has the token: no explicit message is necessary to represent the token.

When p_i has not the token (case $next_i \neq i$), it waits for an UPDATES() message from the next process allowed to broadcast its updates (p_{next_i}). When it receives that message (line 5), p_i updates accordingly its local cache (as in the previous protocol, lines 6-7). This constitutes an early refreshing of its local cache with the new values provided by p_{next_i}.

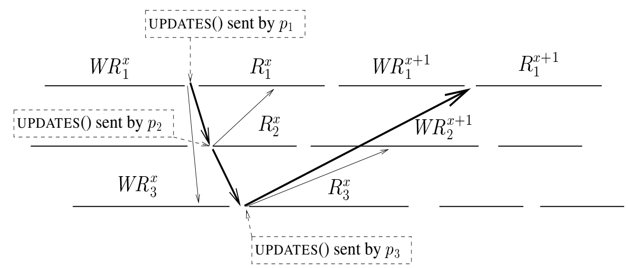

Figure 7. Eager dissemination of the updates

It is important to notice 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 $next_i$. 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. More precisely, let us consider p_i that receives an UPDATES() message from p_j. There are two cases: (1) p_i is executing a WR_i^z segment when it receives an UPDATES() message from p_j. In that case, p_i updates its local cache, but as the updates overwritten by p_i are discarded (line 7), the resulting behavior is exactly the same as if all the updates included in the UPDATES() message had been applied to p_i's local cache before WR_i^z. (2) p_i is executing a R_i^z segment when it receives an UPDATES() message from p_j. Let WR_j^y be the segment that terminated just before p_j sent the UPDATES() message. (Let us remind that such a message is always sent after a WR_j^y segment and before the R_j^y segment that follows it.) Then, R_i^z can be divided in two sub-segments $R1_i^z$ and $R2_i^z$ separated by the processing of p_j's UPDATES() message. The base sequential execution $\widehat{S}$ is now refined as follows: $R1_i^z$ appears before WR_j^y (as p_i does not yet know the new values carried by the UPDATES() message), while $R2_i^z$ appears after WR_j^y (as, after it has processed the UPDATES() message, p_i knows these values).

Finally, it is possible, from an engineering point of view, to adapt this 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 (i.e., when they are such that $next_i = i$). The benefit of such a possibility depends on the read/write access pattern of the upper layer application program.

4 Performance Evaluation

This section presents experiments that show the efficiency of the proposed protocol. Its performance is also compared with that of the popular sequential protocols proposed by Attiya and Welch [7]. The protocol described in Figure 6 is denoted CFJR in the following.

processes	FD	MM	FFT
2	99.53%	99.93%	99.35%
4	99.94%	99.99%	99.95%
8	99.86%	99.99%	99.97%

Figure 8. Ratio of fast read operations per process

Context of the experiment Note first that in our protocols all memory operations are fast except some read operations. (Recall that a memory operation is fast if it can be completed based only on the local state of the process that issued it.) An operation $r_i(x)$ is blocking if, since the last visit of the token, p_i has not updated x (i.e., $updated_i[x]$ is false) while no other variable has been updated (i.e., no_change_i is true). Such a read operation blocks until the process receives the token in the first protocol, or until $next_i = i$ in the second.

An analytic evaluation of how many read operations the protocol allows to be are 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. Our experimental study has considered the protocol of Figure 6. We have implemented this protocol and three typical parallel processing applications: finite differences (FD), matrix multiplication (MM), and fast Fourier transform (FFT). We have implemented FD and MM (as in [22]), and FFT (as in [6]). The code, written in C, uses the *sockets* interface with UDP/IP for computer intercommunication[7].

	DD			MM		
	2	4	8	2	4	8
CFJR	2228.3	2360.0	1450.8	3760.0	3307.5	2813.3
AW-$fast_r$	14133.3	19100.0	22591.7	4816.7	10346.7	8718.3
AW-$fast_w$	12141.7	16400.0	21008.3	4348.3	9720.8	7512.5

	FFT		
	2	4	8
CFJR	554.2	512.5	437.5
AW-$fast_r$	1371.7	14070.0	11304.2
AW-$fast_w$	1227.5	10215.8	9093.3

Figure 9. Execution time of DD, MM and FFT (in seconds)

Experimental results on the protocol efficiency The results that follow concern the protocol described in Figure 6 (denoted CFJR, in short) running with the following application programs: (1) FD with 16384×1024 elements, (2) MM with 1600×1600 matrices, and (3) FFT with 262144 coefficients. The executions have been done in an experimental environment formed by a cluster of 2, 4 and 8 computers connected with a network. Each computer is a PC running Linux Red-Hat with a 1.5GHz AMD CPU, and 512Mbytes of RAM memory. The network is a switched, full-duplex 1Gbps Ethernet. 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. Figure 8 shows the percentage of fast read operations in each process for the previously described FD, MM, and FFT application programs. As it can be observed, almost all read operations are fast in each case.

Comparing the protocol with other protocols We compare our protocol with two sequential consistency protocols proposed by Attiya and Welch [7]. The comparison is done with respect to two important performance measures: (1) the time used to run an application (i.e., its execution time), and (2) the number of messages sent through the network. Attiya and Welch have proposed a sequential consistency protocol where all read operations are fast while the write operations are not fast, which we denote by AW-$fast_r$. They have also proposed a sequential consistency protocol with all write operations are fast while the read operations are not fast, which we denote by AW-$fast_w$. We have executed these protocols with the same set of parallel applications (namely, FD, MM, and FFT), and in the same experimentation cluster.

Figure 9 presents the execution time (in seconds) of running FD, MM, and FFT using each sequential consistency protocol. It can seen that, whatever the case, the execution

[7]The source code can be found at *http://luna.dat.escet.urjc.es/˜ernes*.

DD

	2	4	8
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

	2	4	8
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

	2	4	8
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

Figure 10. Total number (in thousands) of messages+acks sent by each process

time provided by our protocol is much lower than with the other protocols.

Figure 10 presents the total number of messages and acknowledgments (in thousands) 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 our protocol reduces in two orders of magnitude the total number of messages sent by each process. This is due to the fact that while our protocol pieces together several write operations in a single message (in our implementation, up to 100), each other protocol issues one message per write operation. Figure 10 also show that almost each message contains write operations in our protocol. Differently, more than 50% of the messages are acknowledgments in AW-$fast_r$ and AW-$fast_w$.

5 Conclusion

This paper has presented a new sequential consistency protocol. Differently from 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. It provides fast write operations: these operations are always executed "locally" (i.e., without requiring any form of global synchronization). Read operations can also be fast when they are on a variable that has just been previously updated by the same process. An experimental study has been done. It shows that the proposed protocol is particularly efficient for a large class of multiprocess programs.

References

[1] Adve S.V. and Garachorloo K., Shared Memory Models: a Tutorial. *IEEE Computer*, 29(12):66-77, 1997.

[2] Afek Y., Brown G. and Merritt M., Lazy Caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182-205, 1993.

[3] Ahamad M., Hutto P.W., Neiger G., Burns J.E. and Kohli P., Causal memory: Definitions, Implementations and Programming. *Distributed Computing*, 9:37-49, 1995.

[4] Ahamad M. and Kordale R., Scalable Consistency Protocols for Distributed Services. *IEEE Transactions on Parallel and Distributed Systems*, 10(9):888-903, 1999.

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

[6] Akl S.G., The design and analysis of parallel algorithms. *Prentice-Hall*, 1989.

[7] Attiya H. and Welch J.L., Sequential Consistency versus Linearizability. *ACM Transactions on Computer Systems*, 12(2):91-122, 1994.

[8] Bal H. and Tanenbaum A.S., ORCA: a Language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3):190-205, 1992.

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

[10] Herlihy M.P. and Wing J.L., Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492, 1990.

[11] Jimenez E., Fernandez A. and Cholvi V., A parameterized Algorithm that Implements Sequential, Causal and Cache Consistency. *Proc. 10th EUROMICRO Workshop on Parallel, Distributed and Network-Based Processing (PDP'02)*, Islas Canarias (Spain), 2002.

[12] Lamport L., How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C28(9):690-691, 1979.

[13] Li K. and Hudak P., Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321-359, 1989.

[14] Mattern F., Algorithms for Distributed Termination Detection. *Distributed Computing*, 2:161-175, 1987.

[15] Misra J., Axioms for Memory Access in Asynchronous Hardware Systems. *ACM Transactions on Programming Languages and Systems*, 8(1):142-153, 1986.

[16] Mizuno M., Nielsen M.L. and Raynal M., An Optimistic Protocol for a Linearizable Distributed Shared Memory System. *Parallel Processing Letters*, 6(2):265-278, 1996.

[17] Mizuno M., Raynal M. and Zhou J.Z., Sequential Consistency in Distributed Systems. *Proc. Int. Workshop on Theory and Practice of Distributed Systems*, Springer Verlag LNCS #938, pp. 224-241, Dagsthul Castle (Germany), 1994.

[18] Raynal M., Token-Based Sequential Consistency. *Int. Journal of Computer Systems Science and Engineering*, 17(6):359-366, 2002.

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

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

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

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