An Efficient implementation of sequentially consistent distributed shared memories

Masaaki Mizuno, Michel Raynal, Gurdip Singh, Mitchell Neilsen

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

An Efficient Implementation of Sequentially Consistent Distributed Shared Memories

M. Mizuno, M. Raynal, G. Singh and M.L. Neilsen

N° 2085

Novembre 1993

PROGRAMME 1

Architectures parallèles,

bases de données,

réseaux et systèmes distribués

ISSN 0249-6399

An Efficient Implementation of Sequentially Consistent Distributed Shared Memories

M. Mizuno*, M. Raynal**, G. Singh* and M.L. Neilsen***

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet Adp

Rapport de recherche n° 2085 — Novembre 1993 — 16 pages

Abstract: Recently, distributed shared memory systems have received much attention because such an abstraction simplifies programming. In this paper, we present a data consistency protocol for a distributed system which implements *sequentially consistent* memories. The protocol is aimed at an environment where no special support for atomic broadcast exists. As compared to previously proposed protocols, our protocol eliminates the need of atomic broadcast and significantly reduces the amount of information flow among the processors. This is realized by maintaining state information and capturing causal relations among read and write operations.

Key-words: data consistency protocols, causal relations, distributed shared memory, sequential consistency.

(Résumé : tsvp)

This paper will appear in the proceedings, published by North-Holland, of the IFIP WG 10.3 Int. Conf. on Applications of Parallel and Distributed Computing, April 1994.

*Dept. of Computing and Info. Sciences, Kansas State University, Manhattan, KS 66506, USA masaaki@cis.ksu.edu. This work was supported in part by the National Science Foundation under Grant CCR-9201645.

**IRISA, Campus de Beaulieu, 35042 Rennes Cédex, FRANCE raynal@irisa.fr. This work was supported in part by the ESPRIT project BROADCAST (Number 6360) of the Commission of European Communities.

***Computer Science Department, Oklahoma State University, Stillwater, OK 74078, USA neilsen@a.okstate.edu.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 38 38 32

Une mise en oeuvre efficace de la cohérence séquentielle pour les mémoires réparties

Résumé : Cet article présente un protocole qui maintient la cohérence séquentielle pour une mémoire partagée supportée par un ensemble de mémoires locales réparties. Le critère de cohérence séquentiel, proposé par Lamport, garantit que toute exécution parallèle d'un programme est équivalente à une exécution de ce même programme sur un mono-processeur. Le protocole proposé ne nécessite pas de primitive de diffusion, contrairement à la plupart des autres protocoles qui réalisent ce critère de cohérence; il est fondé sur la capture des relations de causalité entre les lectures et les écritures.

Mots-clé : cohérence séquentielle, mémoire distribuée partagée, protocoles de cohérence des données, relations causales.

1 Introduction

In a distributed computing environment, distributed shared memory systems have received much attention since solutions for many classical synchronization problems have been developed for shared memory abstraction, and therefore, such abstraction simplifies programming. In order to improve performance, many proposed and implemented distributed shared memory systems maintain multiple copies for each data item. In such systems, modification to individual copies must be handled carefully to avoid inconsistent system states. Protocols which manage multiple copies in a consistent manner are called *data consistency protocols*. Traditional data consistency protocols require that all copies of an object to be identical at all times [7]. However, this requirement may be unnecessarily stringent for many applications.

Lamport proposed a relaxed form of consistency, called *sequential consistency* [6], which allows more concurrency than the traditional notion of consistency and still can be used in many applications. A shared memory system is sequentially consistent 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.* Note that the real time order of the operations in execution can be different from their order in this sequence. For this reason, in this paper we sometimes refer to a sequential order of operations as an "illusory sequential history."

Three well known protocols to implement sequential consistency are [1, 3, 4]. These protocols assume a system model that consists of several processors with local memory connected via a network. In order to guarantee a unique illusory sequential history to appear to all the processors, all three protocols rely on atomic broadcast to synchronize write operations. Each write operation broadcasts the value to be written (in update protocols) or the data item to be invalidated (in invalidation protocols) to all the processors. Since atomic broadcast orders all the broadcasting messages as preserving the order of messages issued by the same processor, it totally orders all the write operations. If the underlying hardware provides an atomic broadcasting facility such as ethernet or token ring, these protocols can be implemented efficiently. However, without such support, these protocols can be very expensive.

In this paper, we present an efficient protocol for sequentially consistent memories, aiming at an environment where no special support for atomic broadcast exists and the cost of communication is high. The protocol eliminates the need of atomic broadcast and significantly reduces the amount of communication as compared to the above three protocols. In order to form an illusory sequential history, the protocol uses a shared memory module, which is a special processor in the network and

works as a centralized arbitrator. All write operations are sent to the shared memory module, and the shared memory module executes the operations sequentially. Thus, write operations on all objects are synchronized and constitute a total order.

A read operation tries to read a value from its local memory. A local memory may contain obsolete values with respect to an illusory sequential history; that is, if read operations read such values, they would not consistently fit in the sequential order. The system invalidates such out-of-date values. If a read operation tries to access an invalidated value, it is notified, and accesses a valid value from the shared memory module.

Unlike protocols based on atomic broadcasting, a write operation is not automatically propagated to all the processors. Thus, it is not easy for an individual processor to independently detect obsolete values in its local memory. We have developed a unique mechanism for this protocol, in which the shared memory module efficiently detects obsolete values and informs each processor when the processor communicates with the shared memory module. To identify out-of-date data values, the shared memory module analyzes causal relations among the operations that it has processed. In order to capture the causal relations, it maintains special data structures. This is the overhead of the protocol. However, since memories are becoming cheaper, we believe that a strategy appropriate for a distributed system in which the cost of communication is high is to reduce the amount of communication at the expense of memory. Furthermore, this mechanism minimizes the amount of information flow among processors so that only the necessary information to maintain sequential consistency is propagated to each processor. This further reduces the amount of communication. An implementation of this protocol is currently in progress on a distributed memory parallel machine (a hypercube with 32 processors).

In the next section, we review definitions used throughout this paper, and Section 3 presents the protocol.

2 Review of Definitions

In this section, we give definitions of consistency on which our implementation is based.

- A *distributed shared memory system* is a pair $(\{P_1, \cdots, P_N\}, M)$, where $\{P_1, \cdots, P_N\}$ is a set of N processors and M is a shared memory. Each processor P_i sequentially executes read and write operations on data items in M in the order defined by the program running on it. A write operation by processor P_i on a

data item x is denoted by $w_i(x)v$, where v is the value written on x by this operation. A read operation on x by i is denoted by $r_i(x)u$, where u is the value of x returned by this operation. We may omit the parameters of an operation when they are not important.

- A *history* of a distributed shared memory system is a well-ordered[1] partially ordered set (poset) $\widehat{H} = (H, \rightarrow_H)$, where H is a set of read and write operations and $\rightarrow_H$ is an irreflexive and transitive relation on H; that is, $\rightarrow_H$ is a partial order on H.
- A history $\widehat{H} = (H, \rightarrow_H)$ is a *sequential history* if $\rightarrow_H$ is a total order.
- A *processor history* of processor P_i is a sequential history $\widehat{h_i} = (h_i, \rightarrow_{h_i})$, where h_i is the set of read and write operations issued by processor P_i, and total order $\rightarrow_{h_i}$ is specified by the program running on P_i.

  In other words, $\widehat{h_i}$ is the sequence of operations executed by P_i.
- An *execution history* is a history $\widehat{H} = (H, \rightarrow_H)$, where
  1. $H = \cup_{i=1}^{N} h_i$, and
  2. $\rightarrow_H \supseteq \cup_{i=1}^{N} \rightarrow_{h_i}$.
- $\widehat{H}|i$ denotes a restriction of history $\widehat{H}$ to h_i. Thus, if $\widehat{H}$ is an execution history, $\widehat{H}|i = \widehat{h_i}$ for $1 \leq i \leq N$.
- A sequential history $\widehat{H} = (H, \rightarrow_H)$ is *legal* if for every read operation $r(x)v$ in H, there exists a write operation $w(x)v$ such that $w(x)v \rightarrow_H r(x)v$ and there does not exist a write operation $w(x)u$ such that $w(x)v \rightarrow_H w(x)u \rightarrow_H r(x)v$.
- Two histories $\widehat{H_1} = (H_1, \rightarrow_{H_1})$ and $\widehat{H_2} = (H_2, \rightarrow_{H_2})$ are *equivalent* if $H_1 = H_2$ and $\widehat{H_1}|i = \widehat{H_2}|i$ for all i.

Using the above definitions, the notion of sequential consistency proposed by Lamport is formulated as follows:

Definition 1: A distributed memory system $(\{P_1, \cdots, P_N\}, M)$ is *sequentially consistent* if for each of its execution histories $\widehat{H}$, there exists a legal sequential history $\widehat{WR}$ which is equivalent to $\widehat{H}$. □.

For example, consider the following execution history $\widehat{H_1}$ consisting of $\widehat{h_1}$ and $\widehat{h_2}$:

[1] The well-ordered property is that any event in the poset has only finitely many predecessors.

$$\widehat{h_1} = w_1(x)1 \rightarrow_{h_1} w_1(y)2 \rightarrow_{h_1} r_1(x)4$$
$$\widehat{h_2} = w_2(y)3 \rightarrow_{h_2} w_2(x)4 \rightarrow_{h_2} r_2(y)2.$$

$\widehat{H}$ is equivalent to the following legal sequential history $\widehat{WR}_1$:

$$\widehat{WR}_1 = w_1(x)1 \rightarrow_{WR_1} w_2(y)3 \rightarrow_{WR_1} w_2(x)4 \rightarrow_{WR_1} w_1(y)2 \rightarrow_{WR_1} r_2(y)2 \rightarrow_{WR_1} r_1(x)4.$$

In H_1, if $r_1(x)$ reads 1 (instead of 4) and $r_2(x)$ reads 3 (instead of 2), then the execution history is equivalent to the following legal sequential history $\widehat{WR}_2$:

$$\widehat{WR}_2 = w_1(x)1 \rightarrow_{WR_2} w_1(y)2 \rightarrow_{WR_2} r_1(x)1 \rightarrow_{WR_2} w_2(y)3 \rightarrow_{WR_2} w_2(x)4 \rightarrow_{WR_2} r_2(y)3.$$

Now, consider the following execution history $\widehat{H_2}$ consisting of $\widehat{h_1'}$ and $\widehat{h_2'}$:

$$\widehat{h_1'} = w_1(x)1 \rightarrow_{h_1'} r_1(y)4 \rightarrow_{h_1'} w_1(x)2$$
$$\widehat{h_2'} = w_2(y)3 \rightarrow_{h_2'} r_2(x)2 \rightarrow_{h_2'} w_2(y)4.$$

There is no legal sequential history equivalent to $\widehat{H_2}$.

3 A Communication Efficient Implementation

This section describes a data consistency protocol which implements *sequentially consistent* memory. The aim of the protocol is to reduce the amount of communication. This is achieved by maintaining additional information in the shared memory. Furthermore, the protocol does not require atomic broadcasting.

3.1 System model

We assume that the system consists of a shared memory module which is implemented on a dedicated processor and multiple job processors to run application processes. The shared memory module, denoted $SMem$, stores all the data objects. Each processor has a local cache memory, which stores a subset of the data objects, and can reliably communicate with $SMem$.

We assume that a manager process exists on $SMem$ (called the $SMem$ manager) and on each of the job processors (called a processor manager). A processor manager handles read and write requests issued by application processes running on the processor. It also communicates with the $SMem$ manager. The $SMem$ manager handles messages sent by processor managers.

A processor manager processes a write operation issued by an application process by sending a message to $SMem$ and updating its local cache memory. It then awaits response indicating completion of the operation[2]. When a processor manager performs a read operation on an object, it looks the object up in its local memory. If the object is found, its value is returned. Otherwise, it reads the current object value from $SMem$, updates its local memory, and returns the value. A processor manager is informed by the $SMem$ manager of the set of values to be invalidated in its local memory, when it accesses $SMem$.

3.2 Overview of the protocol

Since multiple copies on a data object are maintained, some of the values in a local memory may become out-of-date with respect to the write operations performed on $SMem$. If a processor reads such values, consistency might be violated. Thus, the system must detect and invalidate these out-of-date memory values. The mechanism to detect such values is explained as follows: since the $SMem$ manager processes one request at a time, the operations processed by $SMem$ constitutes a total order, which is the basis of an illusory sequential history $\widehat{WR}$. Consider an execution shown in Figure 1.

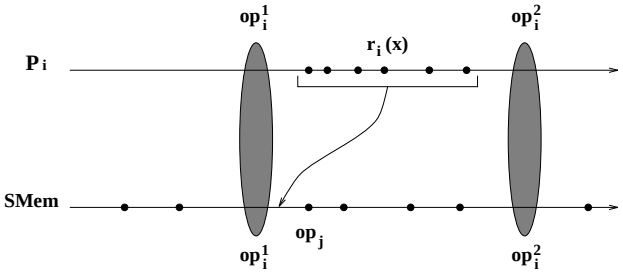

Figure 1: Principle of the protocol.

Assume that P_i sends a message to $SMem$ to perform an operation, say op_i^1. At $SMem$, op_i^1 is ordered after all the previous operations that $SMem$ has processed. Let op_i^2 be the next operation of P_i which results in accessing $SMem$. Let op_j be the operation by any processor P_j which directly follows op_i^1 at $SMem$. Thus, if $i = j$, then $op_j = op_i^2$; otherwise op_j precedes op_i^2 at $SMem$. Let $r_i(x)$ be a read operation of P_i executed in between op_i^1 and op_i^2 (thus, $r_i(x)$ accesses P_i's local memory). The

[2] In Section 3.4, we discuss modification to the protocol in which the processor manager is not blocked by waiting for response from $SMem$.

protocol constructs an illusory sequential history $\widehat{WR}$ by scheduling all such $r_i(x)$ in between op_i^1 and op_j in $\widehat{WR}$. In order to satisfy legality (*i.e.*, consistency of read operations), $r_i(x)$ must read the value written by the most recent write operation on x which precedes op_i^1 at $SMem$. Thus, if the local memory of P_i contains a value written by any other previous write operation on x, the value must be invalidated when $SMem$ performs op_i^1. Such an out-of-date value can be identified by capturing causal relations among read and write operations.

To easily capture the causal relations, each write operation on an object is assigned a unique *version number*. The first write operation on an object is assigned the version number one. The version number assigned to a subsequent write operation is one greater than the previous write operation on the same object. Then, each write operation may be uniquely identified by a pair consisting of its version number and the object, such as "version 4 of object x." Similarly, a value in a data item can be identified by the version number of the write operation which wrote the value. Thus, we also identify a value as the "value of version 4 of object x."

3.3 Description of the protocol

The data structures maintained in $SMem$ are:

- Memory area $M[Object\_Range]$.
- Two-dimensional array $Cache\_Ver[Processor\_Range, Object\_Range]$, where $Cache\_Ver[i, x]$ stores the version number of object x which processor P_i holds in its local memory. The value is zero if the corresponding memory location has been invalidated. Each element of $Cache\_Ver$ is initialized to zero.

- One-dimensional array $Causal[Object\_Range]$, where $Causal[x]$ keeps the version number of the most recent write on object x in the shared memory. Each element of $Causal$ is initialized to zero. Note that $Causal\ [x] = Max_{i=1}^{Processor\_Range} Cache\_Ver[i, x]$ (therefore, this array need not be explicitly maintained. It is included for ease of exposition).

The data structures maintained in processor P_i are:

- A set of valid objects $Valid_i$. $Valid_i$ is initialized to be an empty set.
- Local cache memory area $C_i[x]$ for each object $x \in Valid_i$.

Operations executed by the process manager at processor P_i are described below:

Write(x,v)::
(a) send $[write, x, v]$ message to $SMem$;
(b) $C_i[x] := v$;
(c) receive $[Invalid]$ message from $SMem$;
(d) $Valid_i := (Valid_i - Invalid) \cup \{x\}$;

Read(x)::
  if x $\notin Valid_i$
    then
      send $[read, x]$ message to $SMem$;
      receive $[v, Invalid]$ message from $SMem$;
      $Valid_i := (Valid_i - Invalid) \cup \{x\}$; $C_i[x] := v$;
  fi;
  return $C_i[x]$;

Operations executed by the $SMem$ manager are described below:

Local Procedure Invalidate (**var** $Invalid$) ::
  $Invalid := \emptyset$;
  for each y **in** $Object\_Range, y \neq x$ **do**
    if $Cache\_Ver[i,y] \neq 0$ **and** $Cache\_Ver[i,y] < Causal[y]$
      /* check if object y in local memory at processor P_i is out-of-date
         relative to the current version of y */
      then $Invalid := Invalid \cup \{y\}$; $Cache\_Ver[i,y] := 0$;
  fi;

Receive $[write, x, v]$ **message from processor** P_i::
  $M[x] := v$; increment($Causal[x]$); $Cache\_Ver[i,x] = Causal[x]$;
  Invalidate($Invalid$); return $[Invalid]$ to processor P_i;

Receive $[read, x]$ **message from processor** i::
  $Cache\_Ver[i,x] := Causal[x]$;
  Invalidate($Invalid$); return $[M[x], Invalid]$ to processor P_i;

Each operation must be performed indivisibly. We assume that a write operation on an object is performed before the first read request on the object is issued.

Note that as long as a processor performs only read operations after its local memory is initialized, all local memory values are consistent and it does not need to obtain values from $SMem$. However, this would violate the well-ordered property of a history. For example, assume the following scenario:

- Data item x is initialized to 0.
- Processor P_i executes statement "**while** $(x = 0)$ **do** skip **od** ."

  In the first execution of condition "$x = 0$", it reads x from $SMem$.
- Then, processor P_j executes statement "$x := 1$."

Processor P_i will never know about the execution by P_j and keep looping in the while statement. In this situation, all of the (infinitely many) read operations performed by P_i are scheduled before the write operation by P_j in the illusory sequential history $\widehat{WR}$ and violates the well-ordered property. In order to enforce the well-ordered property on a history, each processor must recognize updates performed by other processors within a finite number of its execution steps. Thus, if a processor P_i does not access $SMem$ for a certain length of time, it must initiate an $SMem$ access operation even though data structure '$Valid_i$' indicates it is not necessary.

Theorem 1: The protocol implements a memory system which is sequentially consistent. □

The proof of Theorem 1 is given in the Appendix.

3.4 A modified protocol

In the above protocol, a processor manager is required to wait for a response from $SMem$ for each write operation (refer to (c) in **Write(x,v)**). This section discusses a simple modification to the protocol to remove the requirement. In this modification, when a processor issues a write operation, the processor manager does not wait for a response. Instead, after sending a message to $SMem$ and updating its local memory (refer to (a) and (b)), it starts executing the next operation. The remaining part of the operation (refer to (d)) is executed when the processor manager receives a response from $SMem$. If the next operation is write, it can immediately send another message to $SMem$. If the next operation is read, it must wait for responses for all the preceding write operations before executing it. Note that the modification requires the communication network between each processor and $SMem$ to be a FIFO channel.

The original and modified protocols show two different ways to implement the global synchronization necessary for sequential consistency. In the original protocol, each write operation has to wait for a response from $SMem$, whereas the waiting is associated with a read operation in the modified protocol. The choice between the two protocols depends on the characteristics of the application.

3.5 Capturing causal relations

In order to efficiently implement data consistency protocols, it is sometimes helpful to capture causal relations between write operations. In message passing systems, vector clocks are often used to identify causal relations among events, each of which is an execution of a send, receive, or internal operation [5, 8]. A vector clock consists of a set of local clocks, one for each processor. In an implementation of a shared memory system, we have to order operations on each object. For this purpose, we can employ a separate vector clock for each object. Secondly, we also have to capture causal relations among operations on different objects. Thus, it is natural to use a vector of vector clocks. Therefore, we find that a two-dimensional array $CR2[Processor\_Range, Object\_Range]$ is appropriate for shared memory systems. When processor P_i writes on object z, the values in $CR2$ are interpreted as follows:

$CR2[j,x] = v$ indicates that all the write operations up to and including the v^{th} write operation by processor P_j on object x causally precede this write on z. Now, we will demonstrate that $CR2$ generalizes the data structures used in our protocol and the protocol proposed by Ahamad *et al.* [2] to capture causal relations among write and read operations.

We can view array $Causal$ used by the protocol described in Section 3.3 as $CR2$ being collapsed into a one-dimensional array, such that $Causal[x] = \sum_{\mathrm{j \in Processor\_Range}} CR2[j,x]$, for $x \in Object\_Range$. This is possible by assigning system wide version numbers to all write operations on an object, instead of processor-level version numbers.

Ahamad *et al.* proposed a protocol for a weaker notion of consistency than sequential consistency, called *causal consistency* [2]. The protocol maintains a one-dimensional array VT [$Processor\_Range$]. We can view VT as $CR2$ being collapsed into one-dimensional array such that $VT[j] = \sum_{\mathrm{x \in Object\_Range}} CR2[j,x]$, for $j \in Processor\_Range$. Thus, $VT[j]$ represents the total number of write operations issued by P_j on any object.

In order to compare $Causal$ and VT, consider the following example:

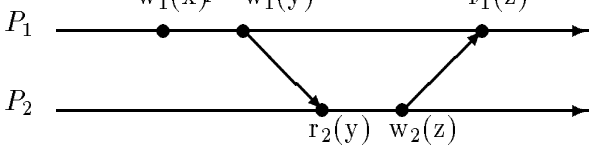

Figure 2. How to capture causal relations.

When P_1 performs $r_1(z)$, VT only records that "P_2 has updated *some* object" but does not know which object. On the other hand, if $Causal$ is used, it captures that "*some* processor has updated z, not x or y," but does not know which processor. Therefore, $Causal$ is more appropriate to capture causal relations among operations and may allow a protocol to minimize unnecessary invalidations as compared to VT.

4 Conclusion

We presented an efficient data consistency protocol for sequential consistency, which aims at an environment where no special support for atomic broadcast exists and the cost of communication is high. Global synchronization among operations is enforced by the shared memory module.

The protocol efficiently detects obsolete values in local memories. This is done by maintaining, within the shared memory module, the state information of local cache memories and causal information among the write operations. Based on the

information, the set of out-of-date values is computed. Thus, the protocol requires additional memory space, and this is the overhead of the protocol. However, the shared memory is used only to exchange information among processors and not to store programs and data structures private to a processor. Thus, we expect that the size of the shared memory is not unreasonably large, and neither is the amount of memory overhead. Furthermore, the amount of additional memory space depends on the granularity of the shared memory. Thus, it may be controlled by choosing a proper granularity. Since memories are becoming cheaper, we believe that a strategy appropriate for a distributed system in which the cost of communication is high is to reduce the amount of communication at the expense of memory.

We also discussed a general framework to capture causal relations among operations in a shared memory system and showed that the mechanism used in our protocol is appropriate.

An implementation of the protocol is currently in progress on a distributed memory parallel machine. Besides sequential consistency, many other consistency criteria have been proposed for distributed shared memory systems. Readers will find a survey on these criteria in [9].

References

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

[2] Ahamad, M., Hutto, P., and John, R. Implementing and programming causal distributed shared memory. In *Proceedings of the IEEE Int'l Conference on Distributed Computing Systems*, pages 274–281, 1991.

[3] Attiya, H. and Welch, J. Sequential consistency versus linearizability. In *Proceedings of the 3rd ACM Sym. Parallel Algorithms and Architectures*, pages 304–315, 1991.

[4] Brown, G. Asynchronous multicaches. *Distributed Computing*, 4:31–36, 1990.

[5] C.J. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, 1991.

[6] Lamport, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28:690–691, 1979.

[7] Li, K. and Hudak, P. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, 1989.

[8] F. Mattern. Virtual time and global states of distributed systems. In Cosnard, Quinton, Raynal, and Robert, editors, *International Workshop on Parallel and Distributed Algorithms*, pages 215–226, North-Holland, Bonas, France, 1989.

[9] Raynal, M. and Mizuno, M. How to find his way in the jungle of consistency criteria for distributed shared memories (or how to escape from Minos' labyrinth). In *Proceedings of International Conference on Future Trends of Distributed Computing Systems*, Lisboa (Portugal), 1993.

Appendix: Proof of Theorem 1

Let $\widehat{H}$ be an execution history of the protocol described in Section 3. In order to show that the protocol implements a sequentially consistent memory defined by Definition 1, we have to show that: there exists a legal sequential history $\widehat{WR} = (WR, \rightarrow_{WR})$, where $WR = H$ and $\widehat{H}|i = \widehat{WR}|i$ for all i.

The construction of $\widehat{WR}$ is completed as follows (refer to Figure 1):

(1) Suppose that $SMem$ processes operation op_i followed by operation op_j. Then, we order $op_i \rightarrow_{WR} op_j$.

(2) Let $r_{i_1}, r_{i_2}, \cdots, r_{i_n}$ be a sequence of consecutive internal read operations performed by P_i immediately after an operation op_i which accesses $SMem$ (the operation after r_{i_n} at P_i also accesses $SMem$). Then, we order $op_i \rightarrow_{WR} r_{i_1} \rightarrow_{WR} r_{i_2} \rightarrow_{WR} \cdots \rightarrow_{WR} r_{i_n}$.

Let the operation after op_i in $\widehat{WR}$ ordered by construction rule (1) be op_j. Then, we order $r_{i_n} \rightarrow_{WR} op_j$.

Note that all of the operations ordered between op_i and op_j are issued only by P_i. Thus, $\widehat{WR}$ enforces a total order.

It is easy to see $H = WR$. Since a processor issues operations sequentially without any overlap and construction rule (2) respects the order of operations issued by the same processor, $\widehat{H}|i = \widehat{WR}|i$ holds for each P_i.

Next, we will show that $\widehat{WR}$ is legal. Assume that, on the contrary, WR is not legal. Then, there must exist a read operation $r_i(x)v$ such that $w(x)v \rightarrow_{WR} w(x)u \rightarrow_{WR} r_i(x)v$ and there does not exist $w(x)s$ such that $w(x)u \rightarrow_{WR} w(x)s \rightarrow_{WR} r_i(x)v$. There are two cases to consider:
Case 1: $r_i(x)$ accesses $SMem$. Clearly, from the protocol, $w(x)u$ writes u to $M[x]$, and $r_i(x)$ is performed after $w(x)u$ on $SMem$. Thus, $r(x)$ does not return v, and history $w(x)v \rightarrow_{WR} w(x)u \rightarrow_{WR} r_i(x)v$ never occurs.
Case 2: $r_i(x)$ is a local read. Let op_i be the last operation before $r_i(x)$ by processor P_i which accesses $SMem$. Thus, $w(x)v \rightarrow_{WR} w(x)u \rightarrow_{WR} r_i(x)v$, and $op_i \rightarrow_{WR} r_i(x)v$. Note that for any $r_i(x)$, there exists such op_i since $Valid_i$ is initially empty. There are three cases to consider based on the position of op_i in $\widehat{WR}$:

1. op_i follows $w(x)u$ (that is, $w(x)v \rightarrow_{WR} w(x)u \rightarrow_{WR} op_i \rightarrow_{WR} r_i(x)v$):

   In this case, there are two cases to consider:

(a) $w(x)u$ is issued by processor P_i: Then $w(x)u$ sets $M[x] = u$ and $C_i[x] = u$, and includes x in $Valid_i$. Since there does not exist $w(x)s$ ordered by $\rightarrow_{WR}$ in between $w(x)u$ and $r_i(x)$, $x \in Valid_i$ and $C_i[x]$ stays unchanged at least until $r_i(x)$ is performed. Thus, $r_i(x)$ locally reads value u from $C_i[x]$, and history $w(x)v \rightarrow_{WR} w(x)u \rightarrow_{WR} r_i(x)v$ never occurs.

(b) $w(x)u$ is not issued by processor P_i: Execution of $w(x)u$ changes $M[x] = u$ and $Cache\_Ver[i,x] < Causal[x]$. Since $r_i(x)$ is a local read, $x \in Valid_i$ when $r_i(x)$ is performed by processor P_i. This means $Cache\_Ver\ [i,x]$ had been changed to $Causal[x]$ before op_i is completed. ¿From the protocol, the change can occur only if write or read operation on x by processor P_i is performed on $SMem$. By the assumption, there does not exist $w(x)s$ ordered in between $w(x)u$ and $r_i(x)$ by $\rightarrow_{WR}$. Therefore, there must be a read operation by processor P_i which reads $M[x]$ in $SMem$ between $w(x)u$ and op_i, including op_i. This read operation also includes x in $Valid_i$ and sets $C_i[x] = u$. Thus, $r_i(x)$ returns u, and history $w(x)v \rightarrow_{WR} w(x)u \rightarrow_{WR} r_i(x)v$ never occurs.

2. op_i is $w(x)u$:

   Then, Operation $w(x)u$ includes x in $Valid_i$ and sets $C_i[x] = u$. Since there is no operation by processor P_i which accesses $SMem$ between $w(x)u$ and $r_i(x)$, $r_i(x)$ returns u. Thus, history $w(x)v \rightarrow_{WR} w(x)u \rightarrow_{WR} r_i(x)v$ never occurs.

3. op_i precedes $w(x)u$ (that is, $op_i \rightarrow_{WR} w(x)u$):

   In this case, construction rule (2) above orders $r_i(x)$ in between op_i and $w(x)u$. Hence, history $w(x)v \rightarrow_{WR} w(x)u \rightarrow_{WR} r_i(x)v$ never occurs.

Unité de recherche Inria Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 Villers Lès Nancy
Unité de recherche Inria Rennes, Irisa, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Unité de recherche Inria Rhône-Alpes, 46 avenue Félix Viallet, 38031 Grenoble Cedex 1
Unité de recherche Inria Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex
Unité de recherche Inria Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 Sophia-Antipolis Cedex

Éditeur
Inria, Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
ISSN 0249-6399