A suite of definitions for consistency criteria in distributed shared memories

Michel RAYNAL *
André SCHIPER **

Abstract

A shared memory built on top of a distributed system constitutes a distributed shared memory (DSM). *If a lot of protocols implementing* DSMs *in various contexts have been proposed, no set of homogeneous definitions has been given for the many semantics offered by these implementations. This paper provides a suite of such definitions for atomic, sequential, causal,* PRAM *and a few others consistency criteria. These definitions are based on a unique framework : a parallel computation is defined as a partial order on the set of read and write operations invoked by processes, and a consistency criterion is defined as a constraint on this partial order. Such an approach provides a simple classification of consistency criteria, from the more to the less constrained one. This paper can also be considered as a survey on consistency criteria for* DSMs.

Key words : Distributed system, Ressource sharing, Memory, Computer theory, Semantics, Logics relation, Consistency.

UN ENSEMBLE HIÉRARCHIQUE DE DÉFINITIONS POUR LES CRITÈRES DE COHÉRENCE DANS LES MÉMOIRES DISTRIBUÉES PARTAGÉES

Résumé

Cet article présente un ensemble de critères de cohérence pour les données accédées par des processus concurrents. La cohérence atomique, la cohérence séquentielle et la cohérence causale sont plus particulièrement étudiées. Grâce à l'utilisation d'un formalisme unique (fondé sur la théorie des ordres partiels) pour définir ces divers critères, on montre que ceux-ci s'imbriquent naturellement les uns dans les autres. Des protocoles implémentant ces divers critères sont également cités. Cet article peut être vu comme un survey *de critères de cohérence. L'originalité de l'approche consiste à les présenter dans un formalisme unique. Ceci permet de mieux comprendre et apprécier les points où se situent leurs similitudes et leurs différences.*

Mots clés : Système réparti, Partage ressource, Mémoire, Informatique théorique, Sémantique, Relation logique, Cohérence.

Contents

I. INTRODUCTION

Since the end of the eighties, the distributed shared memory abstraction (a shared memory built on top of a distributed system) is receiving more and more atten-

* IRISA, Campus de Beaulieu, F-35042 Rennes Cedex, France. raynal@irisa.fr.
** EPFL, Dept d'Informatique, CH-1015 Lausanne, Switzerland. schiper@di.epfl.ch.

tion. One of its very first implementation has been done in the IVY system designed by Li and Hudak [24]. The distributed shared memory abstraction (DSM for short) has many advantages. First, at the application level, DSM frees the programmer from the underlying support as he has to consider only the well known *shared variables programming* paradigm to design a solution to his problem, independently of the system that will run his program (be it a centralized shared memory or a distributed one). Additionally, this facilitates his programming task as a lot of problems (especially related to numerical analysis or image processing) are easier to solve by using the shared variables paradigm than by using the message passing one. Second, at the system level, DSM makes transparent transport of programs, load balancing and process migration.

So, numerous protocols implementing a DSM on top of a distributed memory parallel machine or on top of a distributed system have been proposed. References [30] and [32] survey systems offering a DSM to their users. DSM implementations have common points with multiprocessor caches, networked file systems and distributed databases. Basically, the shared memory is supported by local memories of processors and copies of a data item can simultaneously be present in several local memories. Due to characteristics of the distributed context (asynchronous communications, existence of several copies, etc.), some protocols implementing a shared memory on top of a distributed system offer to users a shared memory whose semantics is lightly different (sometimes in a very subtle way) from the classic semantics associated with a centralized shared memory, namely the *atomic* semantics.

Semantics of a shared memory is expressed by a *consistency criterion*. Such a criterion defines the value returned by every read operation invoked by a process. In nearly all DSMs [30, 25, 32], this consistency criterion is not formally defined and has to be deduced from the protocol implementing the shared memory. This makes study of properties of DSMs difficult and facilitates neither their understanding nor their comparison.

We propose, in this paper, a set of formal definitions for the following consistency criteria : atomic consistency, sequential consistency, causal consistency, PRAM consistency and a few others. These definitions consider a shared memory computation as a partial order on the set of read and write operations issued by processes, and a particular consistency criterion is expressed as a constraint that the partial order has to satisfy [1]. A protocol implementing a DSM with some consistency criterion C has to ensure all computations will satisfy the associated constraint. Such an approach has several advantages. First, as these definitions are independent of particular implementations, they exhibit intrinsic properties associated with consistency criteria; so, this approach follows the abstract data type one by clearly distinguishing the semantics of the *object* offered to users (a shared memory with some semantics) from particular implementations. Second, the set of definitions given in this paper constitutes a hierarchical suite in the following sense : as they all are expressed by using the same formalism, it is possible to order them (from the more to the less constrained); consequently it is easy to see what are the additional constraints required by one consistency criterion with respect to another by comparing their positions within the hierarchy.

The paper is divided into 7 main sections. Section II presents the basic shared memory model. Then, Sections III, IV, V and VI give formal definitions for sequential, atomic, causal and PRAM consistency, respectively. Basic principles of protocols implementing these criteria are also given. Section VII completes the panorama by examining other consistency criteria, namely hybrid, mixed, release and entry consistencies. Finally, Section VIII exhibits similarities between the shared memory model and the message passing model (atomic, sequential, causal and PRAM consistencies, in the shared memory model, are *equivalent*, in the message passing model, to rendez-vous, logically instantaneous, causally ordered and Fifo communications, respectively).

II. SHARED MEMORY MODEL

II.1. Notations.

A shared memory system is composed of a finite set of sequential processes $P_1, \cdots, P_n$ 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). For simplicity, as in [27, 4, 34], we assume all values written into an object x are distinct[2]. 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.

1. Moreover, it is worth noting that this set of formal definitions is based on very few (and simple) mathematical notions, namely : partial order, linear extension, suborder and legality (of read operations).

2. This hypothesis is usual in the domain of database where it is given greater place to *conflict equivalence* than to *view equivalence* when defining *serializability* [31]. Intuitively, it 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.

II.2. Histories.

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 $(h_i, \rightarrow_i)$.

An *execution history* (or simply a history, or a computation) $\widehat{H}$ of a shared memory system is a partial order $\widehat{H} = (H, \rightarrow_H)$ such that[3] :

- $H = \cup_i h_i$,
- $op1 \rightarrow_H op2$ 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 $op_2 \rightarrow_H op1$.

II.3. Legality.

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.

The legality concept is the key notion on which are based our definitions of shared memory consistency criteria. In a legal history no read operation can get an overwritten value. In the following sections, the definition of every consistency criterion follows the same pattern :

- First, according to the consistency criterion considered, one or several histories are defined from the computation $\widehat{H}$,
- Then, $\widehat{H}$ is claimed to satisfy the consistency criterion if and only if this (these) associated history (-ies) is (are) legal.

III. SEQUENTIAL CONSISTENCY

III.1. Definition.

Sequential consistency has been proposed by Lamport in 1979 to define a correctness criterion for multiprocessor shared memory systems [23]. 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 monoprocessor system[4]. More formally, we define sequential consistency in the following way.

Definition. Sequential consistency. A history $\widehat{H} = (H, \rightarrow_H)$ is *sequentially consistent* if it admits a linear extension[5] in which all reads are legal.

As an example let us consider the history $\widehat{H_1}$ (Fig. 1)[6]. Each process P_i, $(i = 1, 2)$, 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 $\widehat{H_1}$ is sequentially consistent by building a legal linear extension $\widehat{S}$ including first the operations issued by P_2 and then the ones issued by P_1. It is also easy to see that the history $\widehat{H_2}$ (Fig. 2) is not sequentially consistent, as no legal linear extension of $\widehat{H_2}$ can be built.

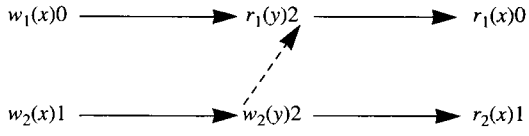

FIG. 1. — A sequentially consistent history $\widehat{H_1}$.

Une histoire séquentiellement cohérente $\widehat{H_1}$.

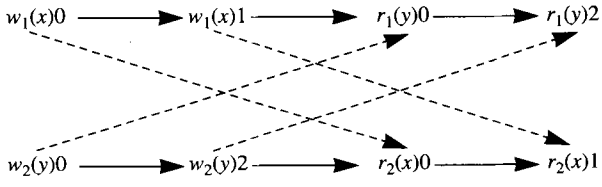

FIG. 2. — A causally consistent history $\widehat{H_2}$.

Une histoire causalement cohérente $\widehat{H_2}$.

3. Section VIII briefly compares definition of computations in the shared memory model and in the message-passing model.

4. In his definition, Lamport assumes that the *process-order* relation defined by the program (see 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.

5. 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, *i.e.*, (i) $S = H$, (ii) $op1 \rightarrow_H op2 \Rightarrow op1 \rightarrow_s op2$ ($\widehat{S}$ maintains the order of all ordered pairs of $\widehat{H}$) and (iii) $\rightarrow_s$ defines a total order.

6. In all figures, only the edges that are not due to transitivity are indicated (transitivity edges come from *process-order* and *read-from* relations). Moreover, (intra-process) *process-order* edges are denoted by continuous arrows and (inter-process) *read-from* edges by dotted arrows.

III.2. Protocols.

Various cache-based protocols implementing sequential consistency have been proposed in the context of parallel machines [2, 7, 28]. In most of these protocols, every processor local memory contains a copy of the whole shared memory. So, each read operation is executed locally, while write operations issued by processes are globally synchronized to get a total order.

In [2] and in the *fast read* protocol of [7], this total order is built by an underlying *atomic broadcast* primitive (messages sent with this primitive are delivered in the same order to each processor [14]). Read operations issued by a process are appropriately scheduled by its processor in order to ensure their legality.

In the protocol presented in [28], a process P_i issuing a write operation sends a write message to a central manager and waits for an answer. The central manager totally orders write operations. After receiving a write message from a process P_i, the central manager sends back an answer informing P_i about its set of copies of variables whose values are out of date and consequently whose future reads will no more be legal. Two versions of the protocol are described; in the first one, variables whose future reads by P_i will be illegal are invalidated in its local memory; in the second one, the manager informs process P_i of the current values associated with these variables.

In the context of distributed systems, where each object is supported by several permanent copies, non cache-based protocols implementing sequential consistency have been proposed. Usually these protocols use votes [37] or quorums [18] mechanisms and, consequently, implement actually atomic consistency which is stronger than sequential consistency (see Section IV). Some systems [11] consider copies as cached values and employ techniques (invalidation *vs* update) similar to the ones used in the management of cache mechanisms [6] ([25] provides an empirical comparison of these techniques).

IV. ATOMIC CONSISTENCY

IV.1. Definition.

Atomic consistency is the *oldest* consistency criterion and the one that is the most encountered in distributed systems. While sequential consistency does not consider real-time, atomic consistency does. So, the underlying model for atomic consistency is asynchrony+real-time. Informally, atomic consistency adds to sequential consistency the following constraint : any two non-overlapping operations must appear in their real-time order within $\widehat{H}$.

Expressed in the previous model this means that executions of operations can no longer be considered as instantaneous. In order to take into account the real-time occurrence of operations, a *real-time precedence* relation, denoted $\prec_{RT}$, is defined in the following way. Let e_i^s and e_j^t be two operations belonging to H; if e_i^s was terminated before (with respect to physical time) e_j^t began, then we have, by definition : $e_i^s \prec_{RT} e_j^t$. Relation $\prec_{RT}$ is a partial order relation : two operations overlapping in real-time are not ordered.

Definition. Atomic consistency. A history $\widehat{H} = (H, \rightarrow_H)$ is *atomically consistent* if it admits a linear extension $\widehat{S} = (H, \rightarrow_S)$ (i) whose all reads are legal (*i.e.*, $\widehat{S}$ is sequentially consistent) and (ii) which is a linear extension of $(H, \rightarrow_{RT})$ (*i.e.*, $e_i^s \prec_{RT} e_j^t \Rightarrow e_i^s \rightarrow_s e_j^t$).

As soon as reads are legal (point i of the definition), they return the *last* value of a variable. The fundamental difference between sequential consistency and atomic consistency lies in the meaning of the word *last*. In the case of sequential consistency *last* refers to logical time, while it refers to physical time in the case of atomic consistency (point *ii* of the definition).

The interested reader will find in [27, 7] a theory of atomic consistency. In [19], under the name of *linearizability*, atomic consistency theory is generalized to objects.

IV.2. Protocols.

The most representative protocol implementing atomic consistency on top of distributed memory parallel machines is the Li-Hudak's one [24]. This protocol uses an invalidation approach. Each data (a page in this protocol) is owned by a process, namely the last process that wrote into it. When a process, wants to read a page for which it has not a copy, it sends a request to the manager of this page that forwards this request to the current owner. When the owner receives such a request, it sends a copy of the page to the requesting process and invalidates its write access right associated with the page. When a process wants to write a page, it sends, through the manager of the corresponding page, a request to the current owner; when receiving such a request, the owner first invalidates all – except his own – copies it has previously disseminated, and then sends its copy to the requesting process. After this, the requesting process is the new owner of the page, and no one else has a copy of the page. These mechanisms ensure atomicity (*i.e.* mutual exclusion) between any couple of read and write operations, and any couple of write operations.

Operating systems, especially distributed file systems, have mainly considered atomic consistency. This criterion is implemented by using a majority voting protocol [37], or a more general quorum protocol [18]. A quorum can be seen as a set of permissions owned by processes and granted to a requesting process. After having executed the operation for which the quorum was necessary, the requesting process gives back permissions to their

owners. To read (write) a data x, a process P_i must get a read quorum $QR_{i,x}$ (write quorum $QW_{i,x}$). Read and write quorums guaranteing atomic consistency are defined by the two following rules which implement the classic readers-writers discipline :

(1) $\forall i \neq j : \forall x \in X : QR_{i,x} \cap QW_{j,x} \neq \emptyset$,

(2) $\forall i, j : \forall x \in X : QW_{i,x} \cap QW_{j,x} \neq \emptyset$,

Rule (1) realizes readers-writers mutual exclusion. It states that if P_i wants to read x, it must get permissions from all processes belonging to $QR_{i,x}$. Similarly, when P_j wants to write x, it must get permissions from all processes belonging to $QW_{j,x}$. As any process in $QR_{i,x} \cap QW_{j,x}$ can grant its permission either to P_i or to P_j (*i.e.*, it can not satisfy simultaneously both of them), the desired exclusion follows. In the same way rule (2) realizes writer-writer mutual exclusion.

V. CAUSAL CONSISTENCY

V.1. Definition.

Causal consistency has first been introduced by Ahamad *et al.* in 1991 [4], and then studied by several authors [5, 3, 34]. It defines a consistency criterion strictly weaker than sequential consistency, and allows for a wait-free implementation of read and write operations in a distributed environment (*i.e.*, causal consistency allows for cheap read/write operations).

With sequential consistency, all processes agree on a same legal linear extension $\widehat{S}$. The agreement defined by causal consistency is weaker. Given a history $\widehat{H}$, it is not required that two processes P_i and P_j agree on the same ordering for the write operations which are not ordered in $\widehat{H}$. The reads are however required to be legal.

The set of operations that may affect a process P_i are the operations of P_i plus the set of all write operations issued by other processes. Let $\widehat{H_i}$ be the sub-history of $\widehat{H}$ from which all read operations not issued by P_i have been removed[7].

Definition. Causal consistency. Let $\widehat{H} = (H, \rightarrow_H)$ be a history. $\widehat{H}$ is *causally consistent* if, for each process P_i, all the read operations of $\widehat{H_i}$ are legal (Fig. 2).

Said another way, in a causally consistent history, all processes see the same partial order on operations but, as processes are sequential, each of them might see a different legal linear extension of this partial order.

So, in a causally consistent history, no read operation of a process P_i can get a value that, from P_i's point of view, has been overwritten by a more *recent* write. As an example consider history $\widehat{H_2}$ (Fig. 2). This history is causally consistent as all its read operations are legal. The history $\widehat{H_3}$ (Fig. 3) is not causally consistent as the read operation $r_3(x)1$ issued by P_3 is not legal : $w_1(x)1 \rightarrow_H r_3(x)2 \rightarrow_H r_3(x)1$. Said another way : when P_3 has issued its first read operation on x (namely $r_3(x)2$), it has got the value 2, and consequently for this process, the value 1 of x has logically been overwritten.

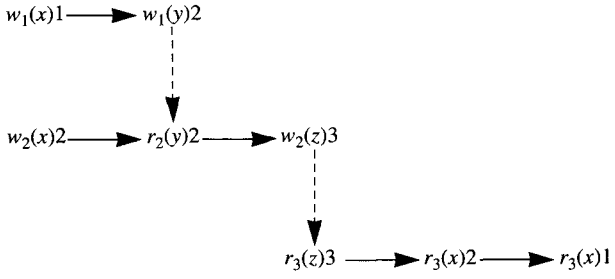

FIG. 3. — A non causally (but PRAM) consistent history $\widehat{H_3}$.

Une histoire causalement cohérente mais non PRAM $\widehat{H_3}$.

Actually, when considering read and write operations as equivalent to receive and send operations in message passing systems, causal consistency is equivalent, in the shared memory model, to causal ordering [14, 33] for the delivery of messages in the message passing model (other *similarities* between both models are addressed in Section VIII).

V.2. Protocols.

Reference [5] presents an implementation of causal memory and studies programming with such a memory. It is shown that, when executed on a causally consistent memory, concurrent-write free[8] or data-race free[9] parallel programs behave as if the underlying shared memory was sequentially consistent. This is particularly interesting as these programs, intended to be executed on a sequentially consistent memory, remain correct when executed on a *less synchronized memory.*

This approach has been generalized in [34] where a synchronization condition, called MSC, less restrictive than concurrent-write freeness or data-race freeness, is proposed. Informally, MSC introduces a new constraint called *concurrent-read freeness* and states a combination

7. More formally, $\widehat{H_i}$ is the sub-relation of $\widehat{H}$ *induced* by the set of all the writes of H and all the reads issued by P_i.

8. A program is *concurrent-write free* if, in all its executions, all its write operations are totally ordered.

9. A program is *data-race free* if all accesses to each variable follow the readers-writer discipline [1].

of (concurrent-write freeness, concurrent-read freeness and data-race freeness) constraints which allows concurrent writes on a same object while ensuring sequential consistency.

This condition can be used when executing, on a causally consistent memory, a program designed for a sequentially consistent memory. For the computation to be correct, it is only necessary to add on top of the causally consistent memory, a protocol implementing the MSC condition. Such a condition is interesting as it provides a layered approach to design a sequentially consistent memory : the library can contain (i) a first protocol implementing a causally consistent memory, and (ii) a second one implementing the MSC condition.

VI. PRAM CONSISTENCY

PRAM (pipelined RAM) consistency [26] is a consistency criterion weaker than causal consistency. The difference, in the shared memory model, between PRAM consistency and causal consistency is the same as the one between fifo ordering and causal ordering for message deliveries in the message passing model [14, 35, 33]. PRAM and fifo are only concerned by *direct relations* between pairs of *adjacents* processes and do not take into account transitivity due to intermediary processes. More precisely, in a message passing system with fifo ordering, two messages sent to a same process by two distinct senders can be delivered in any order, even if the send events are causally related [22] (this is not the case with causal ordering : if the send events are causally related, messages must be delivered in their sending order to the destination process). In the same way, in a PRAM consistent shared memory system, two updates of objects by two distinct processes can be known in any order by a third one[10] (this is not the case in a causally consistent shared memory : if $w_k(x)u \rightarrow_H w_j(x)v$, a process P_i reading x can never get v and then u). As an example consider history $\widehat{H_3}$ which is not causally consistent (Fig. 3). This history is PRAM consistent : as $w_1(x)1$ and $w_2(x)2$ have been issued by distinct processes, values 1 and 2 of x can be known by P_3 in any order.

Let $\widehat{H}$ be a history and let $\widehat{H'} = (H', \rightarrow_{H'})$ be a history defined from $\widehat{H}$ in the following way ($\widehat{H'}$ differs from $\widehat{H}$ only in point *iii* defining transitivity – see Section II.2 – where $\rightarrow_i$ is used instead of $\rightarrow_H$)[11] :

- $H' = H$ (so $H' = \cup_i h_i$)
- $op1 \rightarrow_{H'} op2$ if :

i) $\exists P_i : op1 \rightarrow_i op2$ (*process-order* relation),

or ii) $op1 = w_i(x)v$ and $op2 = r_j(x)v$ (*read-form* relation),

or iii) $\exists op3 : op1 \rightarrow_i op3$ and $op3 \rightarrow_i op2$.

Let $\widehat{H'_i}$ be the sub-history of $\widehat{H'}$ from which all read operations not issued by P_i have been removed (Figure 4 depicts the sub-history $\widehat{H'_3}$ associated with the computation $\widehat{H_3}$ described in Figure 3). $\widehat{H}$ is PRAM consistent if, for each P_i, all read operations of $\widehat{H'_i}$ are legal. This definition of PRAM consistency shows that its difference, with respect to causal consistency, lies only in the nature of the transitivity considered (point *iii* of their definitions).

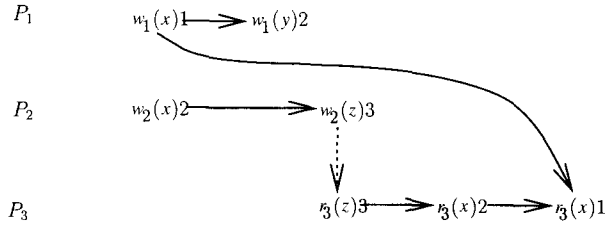

FIG. 4. — The sub-history $\widehat{H'_3}$

Une sous-histoire $\widehat{H'_3}$.

VII. OTHER CONSISTENCY CRITERIA

VII.1. Mixed consistency.

Mixed consistency has been introduced by Agrawal *et al.* in [3]. This consistency criterion on one side considers histories including memory operations (read and write) and synchronization operations (lock, barrier and await), and on the other side combines PRAM consistency with causal consistency; namely every read operation is tagged either PRAM or *causal*.

A history $\widehat{H}$ is mixed consistent if it is :

- causally consistent when considering only the legality of read operations tagged *causal*, and
- PRAM consistent when considering only the legality of read operations tagged PRAM.

The following result is shown in [3]. A mixed consistent history $\widehat{H}$ in which all read operations are tagged *causal*[12] and in which every pair of concurrent operations commute[13], is sequentially consistent.

10. Of course, only one of the updates can be known, if updates overwrite the value previously written by other processes.

11. As $(e \rightarrow_i f) \Rightarrow (e \rightarrow_H f)$, we have $\widehat{H'} \subset \widehat{H}$. When compared to $\widehat{H}$, some transitivity part of the causality relation are missing in $\widehat{H'}$.

12. Note $\widehat{H}$ is then causally consistent.

13. Two concurrent operations commute if their execution order is irrelevant (this is not the case for two concurrent writes on a same object).

VII.2. Hybrid consistency.

Hybrid consistency has been introduced by Attiya and Friedman in [8]. This consistency criterion guarantees properties on the order in which operations appear to be executed at the program level. Operations are labeled either *strong* or *weak*. By defining which operations are strong and which are weak, a user can tune the consistency criterion to his own need. Informally hybrid consistency guarantees the following two properties :

- all strong operations appear to be executed in some sequential order,
- if two operations are invoked by the same process and at least one of them is strong, then they appear to be executed in their invocation order to all processes.

Hence all processes agree on a total order for all strong operations, and on the same order for any pair of strong and weak operations issued by the same process. They can disagree on the relative order of any pair of weak operations issued by a process between two strong operations. Let us consider the two following constraints :

- constraint SW : all writes are strong and all reads are weak,
- constraint SR : all writes are weak and all reads are strong.

The following result is proved in [9] : every hybrid consistent history that satisfies either the constraint SW or the constraint SR is sequentially consistent. When considering only constraint SW, this result is similar to the one implied by the concurrent-write freeness synchronization constraint (see Section V.2) : to get sequential consistency, SW and concurrent-write freeness order all write operations.

VII.3. Non-primitive read and write operations.

Till now we have supposed that read and write operations offered to users are primitive operations. Some authors have considered to provide users with mechanisms allowing them to define non-primitive read and write operations on a set of shared data objects (notation : READ, WRITE). Each such READ or WRITE operation is actually a procedure bracketed by two synchronization operations (*release* and *acquire*). A non-primitive READ is composed of non synchronized primitive read operations while a non-primitive WRITE can include read and write primitive operations. Both *release* consistency [17] and *entry* consistency [13] address such non-primitive READ and WRITE operations, and provide sequential consistency when acquire and release operations guarantee the readers-writers discipline. Concerning protocols implementing these consistency criteria, eager *vs* lazy [21] is an implementation issue whose aim is to reduce the number of messages and the amount of data exchanged; invalidation *vs* update [11] is another implementation issue addressing the management of multiple copies of objects when a cached-based approach is used.

VIII. SHARED MEMORY MODEL VS MESSAGE PASSING MODEL

VIII.1. Message passing computation.

The definition of an history in the shared memory model, and its definition in the message passing model, have some similarities. The first definition of an history in the message passing model is due to Lamport [22]. In the message passing model, operations issued by a process are modeled as events which can be :

- the sending of a message m (event $send(m)$);
- the reception of a message m (event $receive(m)$);
- the execution of a statement involving neither the send nor the receive of a message (*internal* event).

The local history of a process P_i is the sequence $\widehat{h_i}$ of events it has produced. A distributed computation (or an history) $\widehat{H}$ of a set of processes $P_1, \cdots, P_n$ is a partial order $(H, \rightarrow_H)$ defined as in the case of the shared memory model except for point *ii)*. More precisely, the definition of the *read-from* relation is replaced by the definition of a *message* relation :

- $H = \cup_i h_i$,
- $op1 \rightarrow_H op2$ if :

i) $\exists P_i : op1 \rightarrow_i op2$ (in that case, $\rightarrow_H$ is called *process-order* relation),

or ii) $op1 = send(m)$ and $op2 = receive(m)$ (in that case $\rightarrow_H$ is called *message* relation),

or iii) $\exists op3 : op1 \rightarrow_H op3$ and $op3 \rightarrow_H op2$.

VIII.2. Logically instantaneous and causally ordered communications.

The similarity between both models is not limited to their definitions. PRAM, causal and sequential consistencies in the shared memory model correspond to Fifo, causally ordered [14, 33] and logically instantaneous [36] communications in the message passing model. Characterizations of these communication modes can be found in [12, 16, 36].

VIII.2.1. Causally ordered communications.

A distributed computation $\widehat{H}$ has causally ordered communications if :

$\forall m_1, m_2$: $(send(m_1) \rightarrow_H send(m_2)) \wedge (m_1$ *and* m_2 *are sent to the same destination process*$) \Rightarrow (receive(m_1) \rightarrow_H receive(m_2))$.

So, causal ordering imposes, for each process, receive events to be ordered as their associated send events. It

is easy to see that, if $\widehat{H}$ has causally ordered communications, it has also Fifo communications. Figure 5 illustrates causal ordering : in Figure 5a communications are causally ordered, while they are not in Figure 5b. References [14, 35, 33] describe protocols implementing causally ordered communications on top of an asynchronous system, and give examples of problems whose solutions are easier to design when the underlying network ensures communications are causally ordered at the application level.

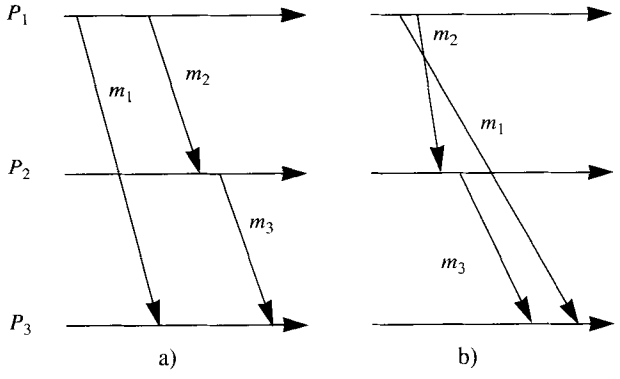

FIG. 5. — Causal ordering.
(a) causally ordered communication ;
(b) not causally ordered communication.

L'ordre causal sur les messages.

VIII.2.2. Logically instantaneous communications.

Informally, a distributed computation $\widehat{H}$ has logically instantaneous communications if the time diagram associated with $\widehat{H}$ can be drawn in such a way that, for each message, its send and receive events can be put on the same vertical line [16] (Fig. 6)[14]. Remark that if $\widehat{H}$ has logically instantaneous communications, it has also causally ordered communications.

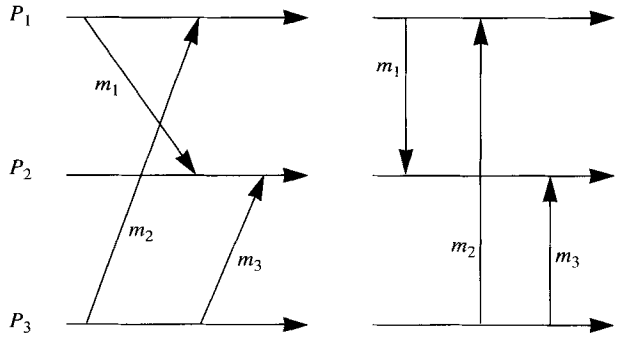

FIG. 6. — Logically instantaneous communication.

Communications logiquement instantanées.

More formally, $\widehat{H}$ has logically instantaneous communications if it is possible to assign logical timestamps to all events such that the time increases within each process and, for each message, the send and the receive events have the same timestamp [29] (T is the timestamping function which assigns a timestamp $T(x)$ to each event x) :

$$\exists T : H \mapsto N \text{ such that :}$$
$$(a \rightarrow_i b \Rightarrow T(a) < T(b)) \wedge$$
$$(\forall m : T(send(m)) = T(receive(m))).$$

VIII.2.3. Relations between both models.

The aim of this section is to exhibit a correspondence between consistency criteria for the shared memory model and communication modes for the message-passing model (see Table I).

TABLE I. — Correspondence between models.

Correspondances entre modèles.

shared memory model	message-passing model
atomic consistency	rendezvous
sequential consistency	logical instantaneity
causal consistency	causal order
PRAM consistency	Fifo order
possibly illegal read	full asynchrony

Let $\widehat{H}$ be a history in the message passing model. [16] introduces the *empty interval property*[15] and shows the following results :

- $\widehat{H}$ has causally ordered communications if it satisfies the following empty interval property :

(3) $\forall receive(m) \in \widehat{H}$:
$$\{op | send(m) \rightarrow_H op \rightarrow_H receive(m)\} = \emptyset,$$

- $\widehat{H}$ has logically instantaneous communications if it has a linear extension $\widehat{S}$ satisfying the following empty interval property : for all messages m, the receive event follows immediately the corresponding send event (*i.e.* communications events appear instantaneous in $\widehat{S}$) :

(4) $\forall receive(m) \in \widehat{S}$:
$$\{op | send(m) \rightarrow_S op \rightarrow_S receive(m)\} = \emptyset.$$

Similar characterizations can be provided for their counterpart in the shared memory model, causal consistency and sequential consistency, respectively. Let $\widehat{H}$ be a history in the shared memory model. Causal consistency and sequential consistency can be characterized in the following way :

14. Remark that if we consider a single unidirectional channel, the Fifo property confuses with logical instantaneity.

15. Two events e_1 and e_2 of a partial order $(P, \rightarrow_P)$ satisfy the *empty interval property* if $\{e | e_1 \rightarrow_P e \rightarrow_P e_2\} = \emptyset$.

• H is causally consistent if, for each process P_i, all its read operations are legal in the history $\widehat{H_i}$ it is associated with[16], *i.e.* :

$$(5)\quad \forall i : \forall r(x)v \in \widehat{H_i} : \{op(x)u | u \neq v \wedge w(x)v \rightarrow_{H_i} op(x)u \rightarrow_{H_i} r(x)v\} = \emptyset.$$

• $\widehat{H}$ is sequentially consistent if it admits a linear extension $\widehat{S}$ in which all read operations are legal, *i.e.* :

$$(6)\quad \forall r(x)v \in \widehat{S} : \{op(x)u | u \neq v \wedge w(x)v \rightarrow_S op(x)u \rightarrow_S r(x)v\} = \emptyset.$$

(3) and (5) on one side, and (4) and (6) on the other side, exhibit strong similarities. In all the four cases, the characterization is based on a similar *empty interval property*. It is applied to a linear extension of the computation in cases (4) and (6). In cases (3) and (5) it is applied, for each process, to the operations/events that may causally affect it[17]. An additional common point between causal consistency and causally ordered communications is the following one : the protocol implementing causally ordered communications described in [33] is used in [5] as basic mechanism to implement causally consistent shared memories on top of a distributed system.

Finally, let us note that the rendezvous communication mode [20, 10] is stronger than logical instantaneity; rendezvous considers *physical* time while logical instantaneity *logical* time. Rendezvous is usually defined in the following way : a system has rendezvous communication if "no message of a given type can be sent along a channel before the receiver is ready to receive it... For an external observer the transmission then looks instantaneous and atomic" [15]. So, it appears that it is possible to exhibit similarities between the rendezvous communication mode and the atomic consistency criterion. Both of them relies on realtime for defining interactions. A deep study of their connections is beyond the scope of this paper. The reader interested by such a study will consult [8].

IX. CONCLUSION

Numerous protocols implementing distributed shared memory systems have been designed. In the most of them, the semantics (consistency criterion) they offer to users is defined only by the description of the protocol and not in an abstract way. This makes it difficult the study of their properties and the appreciation of their differences. In this paper we provided a suite of formal definitions for the most encountered consistency criteria, namely atomic, sequential, causal, release and entry consistencies. These definitions are not bound to particular implementations and are based on a unique framework. This, not only eases their understanding and their comparison, but should facilitate the design of a generic protocol which could be customized to the specific need of each user.

Last but not least, strong *similarities* between the shared memory model and the message passing model have been exhibited.

ACKNOWLEDGMENT

The authors want to thank the referees whose comments helped improve the presentation.

Manuscrit reçu le 28 mars 1997,
accepté le 30 mai 1997.

16. Remember $\widehat{H_i}$ is $\widehat{H}$ from which all the read operations not from P_i have been removed. So, $\widehat{H_i}$ describes the computation that may causally affect P_i.

17. In the message passing model all send and receive events may causally affect (by creating causality chains) each process P_i, so $\widehat{H}$ is considered. In the shared memory model a process P_i cannot be affected by reads of the other processes, so for each process P_i, its associated $\widehat{H_i}$ is considered.

REFERENCES

[1] ADVE (S. V.), HILL (M. D.). Weak ordering – a new definition. *Proc. 17th Annual ISCA (Int. Symposium on Computer Architecture)* (1990), pp. 2-20.

[2] AFEK (Y.), BROWN (G.), MERRITT (M.). Lazy caching. *ACM Transactions on Programming Languages and Systems* (1993), **15,** n° 1, pp. 182-205.

[3] AGRAWAL (D.), CHOY (M.), LEONG (H. V.), SINGH (A.). Mixed consistency : a model for parallel programming. *In Proc. 13th ACM Symposium on Principles of Dist. Computing*, Los Angeles (1994), pp. 101-110.

[4] AHAMAD (M.), BURNS (J. E.), HUTTO (P. W.), NEIGER (G.). Causal memory. *In Proc. 5th Int. Workshop on Distributed Algorithms (WDAG-5)* (1991), *Springer Verlag*, LNCS 579, pp. 9-30.

[5] AHAMAD (M.), HUTTO (P. W.), NEIGER (G.), BURNS (J. E.), KOHLI (P.). Causal memory : definitions, implementations and programming. *Distributed Computing* (1995), **9,** pp. 37-49.

[6] ARCHIBALD (J. L.), BAER (J. L.). Cache coherence protocols : evaluation multiprocessor simulation model. *ACM Transactions on Computer Systems* (1986), **4,** n° 4, pp. 276-298.

[7] ATTIYA (H.), WELCH (J. L.). Sequential consistency versus linearizability. *ACM Transactions on Computer Systems* (1994), **12,** n° 2, pp. 91-122.

[8] ATTIYA (H.), FRIEDMAN (R.). A correctness condition for high performance multiprocessors. *In Proc. 24th ACM Annual Symposium on the Theory of Computing* (1992), pp. 679-690.

[9] ATTIYA (H.), CHAUDHURI (S.), FRIEDMAN (R.), WELCH (J. L.). Shared memory consistency conditions for non sequential executions : definitions and programming strategies. *In Proc. 5th ACM Symposium on Parallel Algorithms and Architectures*, Vale, Germany (July 1993).

[10] BAGRODIA (R. L.). Synchronization of asynchronous processes in CSP. *ACM Transactions on Programming Languages and Systems* (1989), **11,** n° 4, pp. 1053-1065.

[11] BAL (H. E.), KAASHOEK (F.), TANENBAUM (A. S.), JANSEN (J.). Replication techniques for speeding up parallel applications on distributed systems. *Concurrency : Practice and Experience* (1992), **4,** n° 5, pp. 337-355.

[12] BALDONI (R.), RAYNAL (M.). A graph-based characterization of communications modes in distributed executions. *Journal of Foundations of Computing and Decision Sciences* (1995), **25,** n° 1, pp. 3-20.

[13] BERSHAD (B. N.), ZEKAUSKAS (M. J.), SAWDON (W. A.). The Midway distributed shared memory system. *Proc. of the Compcon 93 Conference* (Feb. 1993), pp. 528-537.

[14] BIRMAN (K.), JOSEPH (T.). Reliable communications in the presence of failures. *ACM Transactions on Computer Systems* (1987), **5,** n° 1, pp. 47-76.

[15] BOUGÉ (L.). Repeated snapshots in distributed systems with synchronous communications and their implementation in CSP. *Theoretical Computer Science* (1987), **49,** pp. 145-169.

[16] CHARRON-BOST (B.), MATTERN (F.), TEL (G.). Synchronous and asynchronous communications in distributed systems. *Tech. Report TR91.55, University of Paris 7* (Sep. 1991).

[17] GHARACHORLOO (K.), LENOSKI (D.), LAUDON (J.), GIBBONS (P.), GUPTA (A.), HENNESSEY (J.). Memory consistency and event ordering in scalable shared memory multiprocessors. *Proc. 17th Annual ISCA (Int. Symposium on Computer Architecture)*, Seattle, WA (1990), pp. 15-26.

[18] GARCIA-MOLINA (H.), BARBARA (D.). How to assign votes in a distributed system? *Journal of the ACM* (1985), **32,** n° 4, pp. 841-850.

[19] HERLIHY (M.), WING (J.). Linearizability : a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* (1990), **12,** n° 3, pp. 463-492.

[20] HOARE (C. A. R.). Communicating sequential processes. *Communications of the ACM* (1978), **21,** n° 8, pp. 666-677.

[21] KELEHER (P.), COX (A. L.), ZWAENEPOEL (W.). Lazy release consistency for software distributed shared memory. *Computer Architecture News* (1992), **22,** n° 2, pp. 13-21.

[22] LAMPORT (L.). Time, clocks and the ordering of events in a distributed system. *Communications of the ACM* (1978), **21,** n° 7, pp. 558-565.

[23] LAMPORT (L.). How to make a multiprocessor computer that correctly executes multiprocess programs? *IEEE Transactions on Computers* (1979), **C28,** n° 9, pp. 690-691.

[24] LI (K.), HUDAK (P.). Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems* (1989), **7,** n° 4, pp. 321-359.

[25] LEVELT (W. G.), KAASHOEK (M. F.), BAL (H. E.), TANENBAUM (A. S.). A comparison of two paradigms for distributed shared memory. *Software Practice and Experience* (1992), **22,** n° 11, pp. 985-1010.

[26] LIPTON (R. J.), SANDBERG (J. S.). PRAM : a scalable shared memory. *Tech. Report CS-TR-180-88*, Princeton University (Sep. 1988).

[27] MISRA (J.). Axioms for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems* (1986), **8,** n° 1, pp. 142-153.

[28] MIZUNO (M.), RAYNAL (M.), ZHOU (J. Z.). Sequential consistency in distributed systems. *Proc. Int. Workshop Theory and Practice in Dist. Systems*, Dagstuhl, Germany, *Springer-Verlag* LNCS 938 (K. Birman, F. Mattern and A. Schiper Eds) (1994), pp. 227-241.

[29] MURTY (V. V.), GARG (V. K.). Synchronous message passing. *Technical Report ECE-PDS-93-01*, University of Texas at Austin, Dpt. of Elec. and Computer Engineering (1993).

[30] NITZBERG (B.), LO (V.). Distributed shared memory : a survey of issues and algorithms. *Computer* (1991), **24,** n° 8, pp. 52-60.

[31] PAPADIMITRIOU (C.). The theory of concurrency control. *Computer Science Press* (1986).

[32] PROTIC (J.), TOMAŠEVIĆ (M.), MILUTINOVIĆ (V.). A survey of distributed shared memory systems. *Proc. 28th Annual Hawaii Int. Conf. on System Sciences*, Vol. I (Architecture) (1995), pp. 74-84.

[33] RAYNAL (M.), SCHIPER (A.), TOUEG (S.). The causal ordering abstraction and a simple way to implement it. *Information Processing Letters* (1991), **39,** pp. 343-350.

[34] RAYNAL (M.), SCHIPER (A.). From causal consistency to sequential consistency in shared memory systems. *Proc. 15th Int. Conf. FST&TCS (Foundations of Software Technology and Theoretical Computer Science)*, Bangalore, India, Springer-Verlag LNCS Series 1026 (P. S. Thiagarajan Ed.) (Dec. 1995), pp. 180-194.

[35] SCHIPER (A.), EGGLI (J.), SANDOZ (A.). A new algorithm to implement causal ordering. *In Proc. 3rd Intl. Workshop on Distributed Algorithms (WDAG-3), Springer Verlag* LNCS 392 (J. C. Bermond and M. Raynal Eds) (1989), pp. 219-232.

[36] SONEOKA (T. S.), IBARAKI (T.). Logically instantaneous message passing in asynchronous distributed systems. *IEEE Transactions on Computers* (1994), **43,** n° 5, pp. 513-527.

[37] THOMAS (R. H.). A majority consensus approach to concurrency control for multiple copies databases. *ACM Transactions on Database Systems* (1979), **4,** n° 2, pp. 180-209.