How to find his way in the jungle of consistency criteria for distributed shared memories (or how to escape from Minos' labyrinth)*†

Michel RAYNAL **Masaaki MIZUNO**‡
IRISA Projet ADP, Campus universitaire de Beaulieu, 35042 Rennes Cedex, France
email: raynal@irisa.fr

This paper surveys consistency criteria that have been proposed, and sometimes implemented, for distributed shared objects and memories. Linearizability, sequential consistency, hybrid consistency and causal consistency are particularly emphasized. These criteria are precisely analyzed and protocols that implement them are described. It is suggested that the hybrid consistency, introduced by Attiya and Friedman, constitutes the Ariadne's clew to understand this jungle of consistency criteria.

1 Introduction

A clear view of the future trends of distributed computing systems cannot be obtained only by proposing new ideas, concepts, methods, and technologies. Such new proposals are necessary but in any case not sufficient. To attain this goal, we also have to clarify our ideas about previous proposals (some of which may be contradictory to each other). In this paper, we focus on various consistency criteria for distributed shared memory systems and give an Ariadne's clew to understand them. This will give us clear directions when we are to design and build a distributed system based on a shared memory abstraction.

The paper consists of three major sections. Section 2 presents a distributed system model we consider: the system consists of multiple nodes interconnected by a communication network and provides shared memory abstraction to application programs. The following two sections present various consistency criteria for distributed shared memory systems which have been previously proposed and/or implemented. First in section 3, we address two classes of most stringent (strongest) correctness criteria: One class is characterized by a property called *linearizability*. Another is characterized by a property called *sequential consistency*. Among the two properties, linearizability is more stringent but allows less concurrency in its implementation than sequential consistency. Both classes of distributed memory systems are precisely analyzed, and protocols that implement them are presented. Section 4 addresses a

* This work has been partly supported by the Commission of European Communities under Esprit Basic Research project Number 6360 (BROADCAST) and by the National Science Foundation under Grant CCR-9201645.

† This paper will appear in the proceedings of the IEEE International Conference on Future Trends of Distributed Computing Systems, Lisboa, Sept. 1993

‡ Kansas State University - Manhattan, KS 66506, USA - masaaki@cis.ksu.edu

general framework, called *hybrid consistency*, which was introduced by Attiya and Friedman. Hybrid consistency defines two types of operations, strong and weak operations. By using the operations, it generalizes several classes of consistency criteria, including linearizability, sequential consistency, and other less stringent (weaker) classes of consistency criteria. This framework is particularly helpful to understand subtle differences among many proposed criteria. In Section 4, we also describe causal consistency.

There exists a trade-off between the degree of stringency in consistency criteria and the degree of concurrent executions allowed in their implementations. Thus, clear clarification of consistency criteria for distributed shared memory abstraction is important for a system designer to choose the right semantics of the abstraction for his requirements. This paper contributes a step toward such a clarification.

2 Distributed model

2.1 Application programs

We assume that application programs consist of a set of n processes $P_1, \ldots, P_n$. Processes run concurrently and communicate with one another by accessing a shared memory.

The shared memory maintains a collection of objects. Each object can be accessed by some predefined operations. For our purpose, we consider that each of the operations can be reduced to either a read or a write operation. A read operation by process P_i on object x which returns value v is denoted by $r_i(x)v$. Similarly, a write operation on y with value v by P_i is denoted by $w_i(y)v$. At the application level, read and write operations are indivisible.

2.2 Distributed execution

During the execution of a distributed program, each process issues a sequence of read and write operations on a set of shared memory objects (objects private to each process are not considered). Such a sequence is called a *process history*. A distributed execution is represented by a partial order on the set of operations issued by all the processes. Such a partially ordered set is called a *history*. Consistency criteria on shared memory systems are defined based on particular partially ordered sets to be allowed by the system and the views of processes on the partially ordered sets.

2.3 Underlying hardware support

The underlying support that executes programs consists of a collection of nodes interconnected by a communication network. Nodes communicate with each other by exchanging messages through FIFO channels. There is neither a central memory nor a global clock. The underlying system is reliable and asynchronous; that is, nodes execute processes at their own speed and message transfer delay is finite but unpredictable. Each node has a local cache memory. Since we are not interested in load balancing or migration, in order to facilitate the presentation, we assume that there are N nodes, each of which executes exactly one process. Thus, in the following sections, we use *node* and *process* interchangeably.

2.4 Problem description

We are interested in designing a software layer which implements a shared memory abstraction. The software layer uses local memories of nodes and the

communication network. It is defined by a protocol executed by each node. A protocol at one node interprets all the read and write operations issued by a process running on the node. In order to implement a particular consistency criterion defined on the set of shared objects, a protocol also communicates with protocols on other nodes.

3 Linearizability and sequential consistency

3.1 Basic idea

A centralized memory system (multiprocessor with a central memory) implements a classical *atomic consistency* criterion, which is also called *linearizability*: each object has a unique copy, all the write operations are totally ordered, and a read operation on an object always returns the last value written into the object. Furthermore, all the non-overlapping operations are performed in the order issued.

With the advent of distributed memory parallel machines, object management protocols have been developed. They allow several copies of each objects to exist concurrently (in order to improve the efficiency of read operations), and still ensure linearizability.

As a less restrictive form of consistency, as compared to linearizability, sequential consistency was introduced by Lamport in [9]. The basic idea of the *sequential consistency* criterion is to provide each execution with a total order on all the invoked operations [5,7,9]:

- that is an interleaving of the process histories of all the processes, and
- that could have occurred in a real execution (this means the semantics of objects is not violated by the total order; this property is called legality[1]).

The idea behind sequential consistency is that there exists a correct sequential execution of the distributed program that produces the same result as the distributed execution under consideration.

In this section, adopting the terminology of [7] and borrowing notations from [5], we formally define linearizability and sequential consistency, and present protocols that implement them.

3.2 Linearizability

3.2.1 Consistency criterion

In order to formally define linearizability, the following notation is used to denote real-time order in execution. A distributed execution σ is associated with a partial order $\stackrel{\sigma}{\rightarrow}$ on operations in execution defined by: $op_1 \stackrel{\sigma}{\rightarrow} op_2$ if operation op_1 completes before op_2 begins in real time.

An execution σ is linearizable if there is a legal sequential execution τ that produces the same behavior as σ for all the processes and for all the objects, and all the operations that are ordered in $\stackrel{\sigma}{\rightarrow}$ are ordered in the same way in τ. This definition, introduced by Herlihy and Wing [7], states formally what is sometimes called atomicity by several authors.

Figure 1 displays an execution $\sigma 1$. Let $\sigma 2$ be an execution similar to $\sigma 1$ except for $r_2(x)c$ in $\sigma 1$ being replaced by $r_2(x)b$. Executions $\sigma 1$ and $\sigma 2$ involve three processes and one object. The segments indicate real-time durations of operations with the real time progressing from left to right. Execution $\sigma 1$ is linearizable, while $\sigma 2$ is not. The legal sequential execution $\tau 1$ for $\sigma 1$ is:

$$\tau 1 \; = \; w_1(x)a \; w_2(x)b \; r_1(x)b \; w_3(x)c \; r_2(x)c$$

[1] More formally, a sequence of operations τ is legal if, for every object x, the restriction of τ to operations on x belongs to the serial specification of x [5,7]

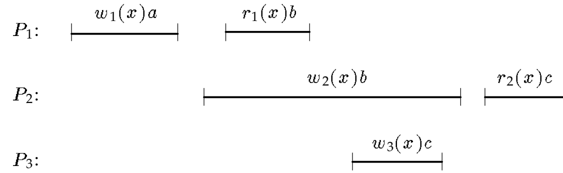

Figure 1. A linearizable execution $\sigma 1$

If $w_3(x)c$ overlapped with $r_1(x)b$ in $\sigma 2$, then $\sigma 2$ would be linearizable as we would have the following legal sequential execution $\tau 2$:

$$\tau 2 \; = \; w_1(x)a \; w_3(x)c \; w_2(x)b \; r_1(x)b \; r_2(x)b$$

3.2.2 Protocols

To improve the efficiency of read operations, many protocols maintain local copies of objects at each node. In order to ensure the linearizability property. they either invalidate (write-once) or update (write-through) local copies when a write operation is performed.

In the *write-once* approach, when an object is written, all of its copies are atomically invalidated, except for the copy in the local memory of the writing node. When a node reads an object, it checks its local copy. If the copy is invalidated, the node must obtain a valid copy from other nodes. One of the most well-known write-once protocols is proposed by Li and Hudak's in [10]. The protocol borrows its principle from the Berkeley cache consistency protocol, where an object is a page. It uses the notion of object owner, which is the last node that wrote it, and object manager, which is a node that knowns the current owner of the object and can thus tell others where to obtain a current copy. The readers are referred to [10] which gives a description and an evaluation of the protocol and to [3] which analyzes cache consistency protocols that ensure linearizability.

In the *write-through* approach, each write operation updates all the copies. The updates must be done atomically [6].

3.2.3 An important practical property

Herlihy and Wing proved the following important result regarding linearizability [7] : (using their terminology) *linearizability is a local property*. It means that a shared memory system that provides the linearizability property can be implemented by a collection of separate protocols, each of which supports linearizability on an individual object or a set of objects independently ([5] calls this property *compositionality*). This property is important from a practical point of view. It implies that linearizability naturally copes with the scaling problem when the number of objects increases.

3.3 Sequential consistency

3.3.1 Consistency criterion

An execution σ is sequentially consistent if there exists a legal sequence τ on all operations of σ such that for each process P_i, the restriction of τ to operations of P_i is the process history of P_i. In others words, "sequential consistency" is "linearizability" minus "real-time order on non overlapping operations."

>From a practical point of view, if an execution is sequentially consistent, all processes and all objects agree on a legal interleaving of operations. However, this sequence can differ from what really occurred. What is required is only

that the commonly agreed sequence τ could have occurred. Consider Figure 2. Execution $\sigma 3$ is not linearizable but is sequentially consistent since every process can agree on $\tau 3$:

$$\tau 3 \; = \; w_3(x)a \; r_3(x)a \; w_2(y)c \; w_2(x)b \; r_1(x)b$$

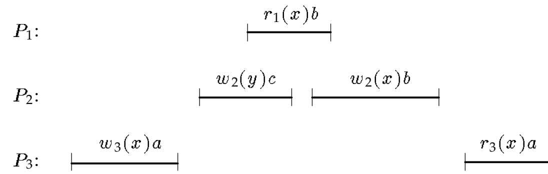

Figure 2. A sequentially consistent execution $\sigma 3$

In concurrency control, term "*serializability*" is used to represent a similar concept. If all transactions consisted of only one operation (ether read or write) on one object, then serializability and sequential consistency merge.

3.3.2 Protocols implementing sequential consistency

Several protocols have been proposed to implement sequential consistency as the correctness criterion. We briefly present two of these protocols. Both protocols assume that each local memory contains copies of all the objects (this assumption is not mandatory, but makes protocols simpler). The first protocol is presented by Attiya and Welch [5]. It is based on an atomic broadcast facility at the system level. The second protocol, by Mizuno et al. [11], is based on a central object manager.

3.3.3 Protocol based on atomic broadcast

An atomic broadcast facility is a primitive, denoted *ab*, that guarantees the following properties:

- every message sent is delivered to every process (including the sender),
- all messages sent are delivered in the same order, and
- two messages sent by the same process are delivered in their sending order.

The protocol executed by process P_i is defined as follows [5]:

$r_i(x)v \; = \; return\ for\ v\ the\ value\ of\ the\ local\ copy\ of\ x$
$w_i(x)v \; =$ **begin** $ab(x, v, i);\ done := false;$
$\quad wait(done);$
end

when (x, v, j) **is delivered to node** i
begin $value\ of\ the\ local\ copy\ of\ x := v;$
if $j = i$ **then** $done := true$ **fi**;
end

It is easy to see that this protocol ensures sequential consistency. All the write operations are executed in the same order at all the nodes due to the *ab* primitive. Furthermore, this protocol implements a fast read operation; that is, no message passing is required to implement a read operation. The duration of a write operation depends on the implementation of the *ab* primitive.

3.3.4 Protocol based on a central manager

This protocol is a special case of a general protocol described in [11]. It assumes the existence of a special node which acts like a central manager for the whole

set of objects; the central manager owns the most up-to-date copy of each object. To simplify the presentation, this node is assumed to be a special node which is not associated with any application process. Let *object_range* be the set of all objects. The central manager maintains the following control information:

- an array *current[object_range]*.
  current[x] is the version number of current x.

- a matrix *known_by[1..N, object_range]*.
  known_by[i,x] is the version number of the copy of x that P_i has in its local memory.

When processor P_i issues a write operation on object x, it sends a message to the central manager. The central manager updates its copy of x and checks, by looking up its tables, whether some copies of objects that P_i has in its local memory are too old (i.e., inconsistent with respect to the current values of the objects). If so, new values are sent to P_i, and P_i updates its local memory.

Process P_i executes the following:

```
r_i(x)v = return for v the value of the local copy of x
w_i(x)v =
begin value of the local copy of x := v
      send (x, v) to the central manager;
      receive (new) from the central manager;
      for all (y, v_y) ∈ new :
                do value of the local copy of y := v_y od;
end
```

When the central manager receives message *(x,v)* from process P_i, it executes the following:

```
begin value of the copy of x := v;
      current[x] := current[x] + 1;
      known_by[i, x] := current[x];
      new := φ;
      forall y ∈ object_range :
         do if known_by[i, y] < current[y]
               then new := new ∪ (y, value of y);
                      known_by[i, y] := current[y]
            fi
         od;
      send (new) to P_i
end
```

It is easy to see that the above protocol totally orders all the write operations. In the protocol by Attiya and Welch, the total order on write operations is enforced by the atomic broadcast primitive, while in this protocol it is achieved by sequentiality of the central manager that processes one message at a time.

This protocol also implements a fast read operation. It can be modified to allow a fast write operation, by giving up a fast read operation, in the following way. After P_i sends a message to the central manager, it does not wait for a response. Instead, it continues its execution. However, when P_i invokes a read operation, it has to wait for responses for all of its previous write operations before executing the read operation.

Finally, the protocol can be generalized to the case where local memories maintain copies of only a dynamically varying subset of the objects (see [11] that proves this general protocol ensures sequential consistency).

4 Hybrid consistency

4.1 Weak consistency criteria

Herlihy and Wing precisely formulated and analyzed linearizability [7], which was the classical criterion for memory consistency and was, and sometimes still is, called atomic consistency. Sequential consistency, introduced informally by Lamport [9], has received attention in past several years. Linearizability and sequential consistency are referred to as *strong consistency*. These consistency criteria ensure that for a given distributed execution σ, there is always a legal sequential execution τ that produces the same behavior as σ for each process and for each object. However, a price must be paid to ensure such consistency criteria; namely, the price to build a total order.

In order to obtain more efficient implementations, many weaker consistency criteria have been proposed: weak ordering, type-specific memory coherence, causal memory, slow memory, release consistency, lazy consistency, processor consistency, weaker memory-access order, etc [1]. All these criteria express some forms of *weak consistency*. Some of these definitions have only subtle differences, and it can be difficult to see their common and different points. Furthermore, they are generally defined by protocols that implement them, instead of formal descriptions of how read and write operations appear to users.

In order to clarify these criteria, we adopt a general framework, called *hybrid consistency*, proposed by Attiya and Friedman [4]. In our point of view, this framework is an Ariadne's clew to escape Minos'labyrinth composed of all these consistency criteria.

In hybrid consistency, read and write operations on the shared objects are classified as either strong or weak. Hybrid consistency guarantees properties on the order in which these operations appear to be executed at the program level. By defining which operations are strong and which are weak, a user can tune the consistency criterion to his own need.

4.2 Strong and weak operations

Informally, hybrid consistency guarantees the following two properties:

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

All processes agree on the total order of any two strong operations or any pair of strong and weak operations issued by one process, but can disagree on the relative order of any two weak operations that appear between two strong operations.

If all operations are weak, we obtain the weakest possible consistency. In the other extreme, in which all operations are strong, we obtain strong consistency. In the latter case, if we additionally impose the real-time ordering of operations on the sequential order, we obtain linearizability. If the order of operations in the sequential order can differ from the real-time order, we obtain sequential consistency. The interested readers will find more formal definitions in [4].

4.3 A protocol implementing hybrid consistency

Implementation of strong operations needs global synchronization. On the other hand, a weak operation can be executed immediately at the invoking node and disseminated to other nodes without special care. As mentioned in Section 2.3, we assume that the underlying network is fully connected with FIFO channels. This property is used to ensure that messages carrying strong

and weak operations issued by a process are received in the same order at every destination node. A message broadcast by a node is received by all other nodes.

First, consider the implementation of a weak operation *p* on object *x* invoked by P_i, denoted $p_i(x, \ldots)$. This case is easy: the operation is executed locally and broadcast to other nodes that execute it as soon as they receive it (weak read operations need not be broadcast since they do not modify values of objects):

```
weak operation p_i(x,...) =
        begin p(x,...) is executed locally;
              broadcast (weak, p(x,...));
        end
receipt by node i of (weak, p(x,...))
              execute p(x,...) on the local copy of x;
```

Next, consider the implementation of strong operations proposed in [4]. This implementation ensures linearizability for strong operations. The global ordering on these operations is enforced by the use of timestamps. At each node P_i, $expected_i[i]$ stores a logical clock and $expected_i[j]$ is the lower bound on all the logical clock values that node P_i will receive from node P_j. A timestamp of a strong operation issued by P_i is represented by pair $(expected_i[i], i)$, and the comparison of two timestamps is defined as follows:

$$(h, i) < (k, j) \Leftrightarrow (h < k \textbf{ or } (h = k \textbf{ and } i < j))$$

In order to keep messages that have arrived but cannot yet be interpreted, each node P_i maintains a set $pending_i$. To ensure liveness, some *ack* messages carrying clock values are used as in [8]. Finally a control message *done* is sent by node P_j to node P_i to inform that the strong operation issued by P_i has been executed at P_j. As soon as P_i knows that the strong operation $p(x, \ldots)$ issued by P_i has been executed on all the nodes, the invocation of $p_i(x, \ldots)$ is considered to have completed and P_i can proceed. Note that if sequential consistency is to be implemented rather than linearizability, for a strong operation, P_i needs to receive message *done* only from itself.

In a more formal way, the implementation of a strong operation $p_i(x, \ldots)$ invoked by P_i can be described as follows (see [4] for a proof of an improved version of this protocol):

```
strong operation p_i(x,...) =
  begin broadcast(strong, p(x,...), (expected_i[i], i));
        put(p(x,...), (expected_i[i], i)) in pending_i;
        expected_i[i] := expected_i[i] + 1;
        progress;
        wait until N messages done have arrived;
  end
receipt by node P_i of (strong, p(x,...), (k, j))
  begin put(p(x,...), (k, j)) in pending_i;
     expected_i[j] := k + 1;
     if expected_i[i] < k + 1 then  expected_i[i] := k + 1;
                                    broadcast ack(k + 1, i)
     fi;
     progress;
  end

receipt by node i of ack(t, j)
  begin expected_i[j] := t;
     progress;
  end
procedure progress;
  begin continue := true;
    while pending_i ≠ φ and continue
     do let y = (p(x,...), (k, j)) the element of pending_i
```

```
                    with the smallest timestamp;
      if k < min_j(expected_i[j])
       then execute locally p(x, ...);
        delete y from pending_i;
        send done to node j
       else continue := false
      fi
    enddo
  end
```

4.4 Remarks

In an actual use of the protocol, read operations are generally weak and write operations are strong. The protocol allows the attribute of an operation (strong or weak) to be defined at run-time. In a more general setting, weak and strong attributes are a way to declare which operations are commutative and which are not.

4.5 Integrating causal consistency

Causal consistency was first introduced by Ahamad et al. [2]. It is stronger than weak consistency but weaker than strong consistency. The consistency is based on Lamport's causality rules [8], adapted to read and write operations. Read operations have to return values consistent with causally related read and write operations. However, write operations on an object can be concurrent and perceived differently by different processes. The causality relation, denoted $\xrightarrow{c}$, resulting from an execution is defined as follows:

- if op_1 and op_2 are invoked by P_i and op_1 occurred first, then $op_1 \xrightarrow{c} op_2$.
- if $op_1 = w_j(x)v$ and $op_2 = r_i(x)v$, then $op_1 \xrightarrow{c} op_2$.
  (we assume that all values written to an object are distinct; furthermore, for $\xrightarrow{c}$ to be legal, there is no intervening $w_k(x)u$ such that $op_1 \rightarrow w_k(x)u \rightarrow op_2$ [11].)
- if $op_1 \xrightarrow{c} op_2$ and $op_2 \xrightarrow{c} op_3$, then $op_1 \xrightarrow{c} op_3$.

To illustrate causal consistency, consider the following distributed execution σ:

$$P_i : \ w_i(x)a; \ r_i(x)b$$
$$P_j : \ w_j(x)b; \ r_j(x)a$$

Execution σ is causally consistent but not sequentially consistent. The write operations on *x* are concurrent and P_i and P_j perceive the following sequential executions τ_i and τ_j that are consistent with σ:

$$\tau_i = w_i(x)a \ w_j(x)b \ r_i(x)b$$
$$\tau_j = w_j(x)b \ w_i(x)a \ r_j(x)a$$

In other words, all processes could agree on the same partial order. However, each process P_i has its own sequential perception on the set of operations that P_i has invoked, say O_i, and other operations causally preceding the operations in O_i. The sequential perceptions by different processes may by different.

Several protocols have been proposed to implement causal consistency. We show the one presented in [2]. Each node receives all the write operations and perceives them sequentially in an order consistent with $\xrightarrow{c}$. The partial order $\xrightarrow{c}$ is captured by vector clocks *vc[1..n]* associated with each write operation. If *vc(op)* is the timestamp (vector clock) value associated with *op*, we have: $op_1 \rightarrow op_2 \Leftrightarrow vc(op_1) < vc(op_2)$, where

$$vc(op_1) < vc(op_2) \Leftrightarrow \forall k : vc(op_1)[k] \leq vc(op_2)[k] \ \textbf{and} \ vc(op_1) \neq vc(op_2)$$

Each node P_i maintains the following: (1) a set $pending_i$ to store all write operations received but not yet processed by node P_i, (2) a vector vc_i, and (3) copies of all shared objects.

$r_i(x)v =$ *returns for v the value of the local copy of x*

$w_i(x)v =$

begin $vc_i[i] := vc_i[i] + 1;$
value of the local copy of $x := v;$
$broadcast(x, v, vc_i, i);$ *progress*
end

receipt by node i **of** (x, v, vc, j)
begin $vc_i[j] := vc[j];$
$put(x, v, vc)$ **in** $pending_i;$
progress
end

procedure *progress*;
begin $continue := true;$
while $pending_i \neq \phi$ **and** *continue*
do let $y = (x, v, vc)$ *one of the elements of* $pending_i$
such that *all other elements of* $pending_i$ *have a timestamp* vc' *with* $\neg(vc' < vc);$
if $vc \leq vc_i$ **then** *localcopy of* $x := v;$
delete y **from** $pending_i$
else $continue := false$
fi
enddo
end

As we can see, causal consistency is "weak operations" plus "causality". Furthermore, if all the processes perceive write operations sequentially in the same order, it becomes strong consistency. Thus, we obtain the following relation : "causal consistency" plus "total order on all writes on all objects" $\Rightarrow$ "strong consistency".

5 Conclusion

This paper surveyed and classified various consistency criteria for distributed shared memories. We showed that hybrid consistency is a good framework to understand these criteria. Linearizability is well understood as it corresponds to the classic common memory multiprocessor run-time model. Some work remains to be done on other consistency criteria. Among such work, classification of applications based on consistency criteria that they fit in is certainly very important from a theoretical as well as a practical point of view.

References

[1] ADVE S.V., HILL M.D. *Weak ordering: a new definition.* Proc. 17th IEEE Symposium on Comp. Arch., (1990), pp. 2-14.

[2] AHAMAD M., BURNS J.E., HUTTO P.W., NEIGER G. *Causal memory.* Proc. 5th Int. Workshop on Dist. Alg., Delphi Greece, Springer-Verlag LNCS 579, (1991), pp. 9-30.

[3] ARCHIBALD J., BAER J.L. *Cache coherence protocols: evaluation using a multiprocessor simulation model.* ACM Trans. on Comp. Systems, Vol.4,4, (1986), pp. 273-298.

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

[5] ATTIYA H., WELCH J. *Sequential consistency versus linearizability.* Proc. 3rd ACM Symposium on Par. Algo. and Architectures, (july 1991), pp. 304-315.

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

[7] HERLIHY M., WING J. *Linearizability : a correctness conditions for concurrent objects.* ACM Trans. on Prog. Languages and Systems, Vol.12,3, (1990), pp. 463-492.

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

[9] LAMPORT L. *How to make a multiprocessor computer that correctly executes multiprocess programs.* IEEE Trans. on Computers, Vol.C-28,9, (1979), pp. 690-691.

[10] LI K., HUDAK P. *Memory coherence in shared virtual memory systems.* ACM Trans. on Comp. Systems, Vol.7,4, (1989), pp. 321-359.

[11] MIZUNO M., RAYNAL M., SINGH G., NEILSEN M. *Communication efficient distributed shared memories.* Research Report 691, IRISA, Rennes, (déc. 1992), 21 p.