A Parametrized Algorithm that Implements Sequential, Causal, and Cache Memory Consistencies

Ernesto Jiménez [b], Antonio Fernández [a], Vicent Cholvi [c,*]

[a] *Universidad Rey Juan Carlos, 28933 Móstoles, Spain*

[b] *Universidad Politécnica de Madrid, 28031 Madrid, Spain*

[c] *Universitat Jaume I, 12071 Castellón, Spain*

Abstract

In this paper we present an algorithm that can be used to implement sequential, causal, or cache consistency in distributed shared memory (DSM) systems. For this purpose it has a parameter that allows to choose the consistency model to be implemented. We can also use our algorithm such that not all processes have the same value in this parameter (we have shown the resulting consistency). This characteristic allows to choose a concrete consistency model but implementing it with the algorithm more efficient in each case (in function of the requirements of the applications). As far as we know, this is the first algorithm proposed that implements cache coherence.

In our algorithm, when implementing causal and cache consistency all read and write operations are executed locally (i.e., are *fast*). It is known that no sequential algorithm has only fast memory operations. However, in our algorithm, when implementing sequential consistency all write operations and some read operations are fast. The algorithm uses propagation and full replication, where values written by a process are propagated to the rest of processes. It works in a cyclic turn fashion, with each process of the DSM system broadcasting one message in its turn. The values written by the process are sent in the message (instead of sending one message for each write operation), but unnecessary values are excluded. All this allows to control the amount of message traffic due to the algorithm.

Key words: Distributed shared memory, causal consistency, sequential consistency, distributed systems, coherency models.

* Corresponding author. Address: Departamento de Lenguajes y Sistemas Informáticos, Universitat Jaume I, Campus de Riu Sec, 12071 Castellón (Spain). Email: vcholvi@uji.es

1 Introduction

Distributed shared memory (DSM) is a well-known mechanism for inter-process communication in a distributed environment [1,2]. Roughly speaking, it consists of using read and write operations for inter-process communication thus hiding from the programmer the particular communication technique employed so that they do not need to be involved in the management of messages.

In general, in order to increase concurrency, most DSM protocols support *replication* of data. With replication, there are copies (replicas) of the same variables in the local memories of several processes of the system, which allows these processes to use the variables simultaneously. However, in order to guarantee the *consistency* of the shared memory, the system must control the replicas when the variables are updated. That control can be done by either *invalidating* outdated replicas or by *propagating* the new variable values to update the replicas. When propagation is used, a replica of the whole shared memory is usually kept in each process.

An interesting property of any algorithm implementing a consistency model is how long can a memory operation take. If a memory operation does not need to wait for any communication to finish, and can be completed based only on the local state of the process that issued it, it is said that the operation is *fast*, which is a very desirable feature. An algorithm is fast if all its operations are fast. For instance, one of the most widely known memory models, the causal consistency model [3], has been implemented in a fast way a number of times [3–5]. On the contrary, another of the widely known memory models, the sequential [6], has been shown that cannot be implemented in a fast manner [7]. This impossibility result restricts the efficiency of any implementation of sequential consistency[8,9].

Our Work

In this paper, we introduce a parametrized algorithm that implements sequential, causal, and cache consistency [10] and allows us to change the model it implements on-line. The main reasons to choose these three models of consistency are the following. It has been shown that many practical distributed applications require competing operations (i.e., operations that need synchronization among them) ([11]). We have chosen to implement the sequential consistency model because it is the most popular proposed model that provides competing operations (beside atomic consistency model ([12]), which is more restrictive). However, it has also been shown that there are several classes of applications which when executed with algorithms that implement

causal consistency behave as sequentially consistent ([3,13]). Hence, we have also chosen to implement the causal consistency model with an algorithm where all memory operations are fast, and in this manner to avoid the efficiency problems of sequential consistency algorithms ([7]). The cache model is also included, even though it is not so popular, because it is extremely simply integrated in our algorithm and it has an interest (at least theoretical) for applications that require competing operations but only on the same variable. Furthermore, as far as we know, this is the first algorithm proposed to implement cache consistency.

In order to increase concurrency, most DSM algorithms support *replication* of data. With replication, there are copies (replicas) of the same variables in the local memories of several processes of the system, which allows these processes to use the variables simultaneously. However, in order to guarantee the consistency of the shared memory, the system must control the replicas when the variables are updated. That control can be done by either *invalidating* outdated replicas or by *propagating* the new variable values to update the replicas. When propagation is used, a replica of the whole shared memory is usually kept in each process. Our algorithm uses full propagation and uses broadcasts to perform such a task. It works as follows: a write operations is propagated from the process that issues it to the rest of processes so that they can apply it locally. However, write operations are not propagated immediately. The algorithm works on a cyclic turn fashion, with each process broadcasting one message in its turn. This scheme allows a very simple control of the load of messages in the network, since only one message is sent by each process at its turn. That makes several write operations to be grouped in a single propagation message, thus reducing the network load.

When implementing causal and cache consistency, all the operations in our algorithm are fast. Obviously, this is not the case for the sequential model (remember that, from the results in [7], it is derived the impossibility of having all the memory operations fast). However, even in the case of the sequential model, all write operations are always fast. In turn, this does not happen for read operations, but there is only one situation where read operations must be non–fast: when the process that issues that read operation has not issued, since its last turn, any write operations on the variable being read and has issued some write operation on another variable. In this case, the process must wait until reaching its turn.

Comparison with Previous Work

From the set of algorithms that implement DSM, two of them have features similar to those presented in this paper. The first one, which has been pro-

posed by Afek et al. [14] (for sequential memory), they also ensure that write operations will be fast. Also, read operations are fast except for some situations. But those situations are more general than that in our algorithm, which makes our algorithm more "fast". Furthermore, we do not send each variable update in a single message as it is done in [14] and we can also bound the number of messages sent. Finally, in [14] it is assumed that there is a communication medium among all processes (and with the shared memory) that guarantees total order among concurrent write operations. In our case, we do not have such a restriction and enforce the order of the operations by using a cyclic turn technique.

On the other hand, in [13], the authors propose an algorithm that implements three consistency models (sequential, causal and a hybrid between both models). Such an algorithm can dynamically switch among these three consistency models. However, there are a number of differences with the algorithm we propose. First, their algorithm is designed separating the propagation mechanism from the consistency policy. On the contrary, in our algorithm the propagation mechanism is enough to maintain the consistency model. Furthermore, in their implementation they use an adaptation of vector clocks [15] (called *version vectors*) which results in a big waste of memory in each node, and bigger messages to send through the network. Finally, their implementation also forces several restrictions to achieve a total order: a) two update transactions can not be executed concurrently, and b) no update transaction is allowed whenever query transactions are ongoing.

The rest of the paper is organized as follows. In Section 2 we introduce basic definitions. In Section 3 we introduce the algorithm we propose. In Sections 4, 5, and 6 we prove the correctness of our algorithm. In Section 7 we provide an analysis of the complexity of our algorithm. In Section 8 we show the consistency when not all processes are executing our algorithm with the same parameter, and in Section 9 we present our concluding remarks.

2 Definitions

In this paper we assume a distributed system that consists of a set of n processes (each uniquely identified by a value in the range $0...n-1$). Processes do not fail and are connected by a reliable message passing subsystem. These processes use their local memory and the message passing system to implement a shared memory abstraction. This abstraction is accessed through read and write operations on variables of the memory. The execution of these memory operations must be consistent with the particular *memory consistency model*.

Each memory operation acts on a named variable and has an associated value.

A write operation by process p, denoted $w_p(x)v$, stores the value v in the variable x. Similarly, a read operation, denoted $r_p(x)v$, reports to the process p that issued it the value v stored in the variable x by write operation. When it doesn't matter the process that performs the operation, we simply will denote them as $w(x)v$ and $r(x)v$. To simplify the analysis, we assume that a given value is written at most once in any given variable and that the initial values of the variables are set by using write operations.

In this paper we present an algorithm that uses replication and propagation. We assume each process holds a copy of the whole set of variables in the shared memory. When needed, we use x_p to denote the local copy of variable x in process p. Different copies of the same variable can hold different values at the same time.

We call α to the set of read and write operations obtained in an execution of the memory algorithm.

Now we define an order among the operations observed by the processes in an execution of the memory algorithm.

Definition 1 (Execution Order) *Let* op *and* $\mathsf{op}' \in \alpha$. *Then* op *precedes* op' *in the execution order, denoted* $\mathsf{op} \prec \mathsf{op}'$, *if:*
1. op *and* op' *are operations from the same process and* op *is issued before* op',
2. $\mathsf{op} = w(x)v$ *and* $\mathsf{op}' = r(x)v$, *or*
3. $\exists \mathsf{op}'' \in \alpha : \mathsf{op} \prec \mathsf{op}'' \prec \mathsf{op}'$

From this last definition, we also derive the non-transitive execution order (denoted as $\prec_{nt}$) as a restriction of the execution order if the transitive closure (i.e., the third condition) is not applied. Note that if $\mathsf{op} \prec_{nt} \mathsf{op}'$, then op has been executed before op'. Hence, if $\mathsf{op} \prec \mathsf{op}'$, then op has also been executed before op'. If $\mathsf{op} \prec \mathsf{op}'$, we define by *related sequence* between op and op' a sequence of operations $\mathsf{op}^1, \mathsf{op}^2, \ldots, \mathsf{op}^m$ such that $\mathsf{op}^1 = \mathsf{op}$, $\mathsf{op}^m = \mathsf{op}'$, and $\mathsf{op}^j \prec_{nt} \mathsf{op}^{j+1}$ for $1 \leq j < m$.

We say that α_p is the set of operations obtained by removing from α all read operations issued by processes other than p. We also say that $\alpha(x)$ is the set of operations obtained by removing from α all the operations on variables other than x.

Definition 2 (View) *We denote by* system view β, process view β_p *or* variable view $\beta(x)$ *a sequence formed with all operations of* α, α_p *or* $\alpha(x)$, *respectively, such that this sequence preserves the execution order* $\prec$.

Note that, due to the existence of operations that are not affected by the execution order, there can be a lot of sequences on α, α_p or $\alpha(x)$, not only β,

β_p or $\beta(x)$, that preserve $\prec$.

We use $\mathsf{op} \rightarrow \mathsf{op}'$ to denote that op precedes op' in a sequence of operations. Abusing the notation, we will also use $\mathsf{set}_1 \rightarrow \mathsf{set}_2$, where set_1 and set_2 are set of operations, to denote that all the operations in set_1 precede all the operations in set_2.

Definition 3 (Legal View) *A view β on α is legal if* $\forall r(x)v \in \alpha$, $\nexists w(x)u \in \alpha :$
$w(x)v \rightarrow w(x)u \rightarrow r(x)v$ *in* β.

Definition 4 (Sequential, Causal or Cache Algorithm)

- *An algorithm implements* sequential consistency *if for each execution* α *there exists a legal view of it.*
- *An algorithm implements* causal consistency *if for each execution* α *there exists a legal view* β_p *of each* α_p, $\forall p$
- *An algorithm implements* cache consistency *if for each execution* α *there exists a legal view* $\beta(x)$ *of each* $\alpha(x)$, $\forall x$.

3 The Algorithm

In this section we present the parametrized algorithm A that implements causal, cache and sequential consistency. Figure 1 presents the algorithm in detail. As it can be noted, it is run with a parameter model, which defines the consistency model that the algorithm must implement. Hence, the parameter must take one of the values causal, $\mathsf{sequential}$, or cache.

In Figure 1 it can be seen that all write operations are fast. When a process p issues a write operation $w_p(x)v$, the algorithm changes the local copy of variable x (which we denote by x_p) to the value v, includes the pair (x, v) in a local set of variable updates (which we call $\mathsf{updates}_p$), and returns control. This set $\mathsf{updates}_p$ will later be asynchronously propagated to the rest of processes. Note that, if a pair with the variable x was already in $\mathsf{updates}_p$, it is removed before inserting the new pair, since it does not need to be propagated anymore.

Processes propagate their respective sets $\mathsf{updates}_p$ in a cyclic turn fashion, following the order of their identifiers. To maintain the turn, each process p uses a variable turn_p which contains the identifier of the process whose set must be propagated next (from p's view). When $\mathsf{turn}_p = p$, process p itself uses the communication channels among processes to send to the rest of processes its local set of updates $\mathsf{updates}_p$. This is done in the algorithm

```
Initialization ::
begin
  turn_p ← 0
  updates_p ← ∅
end

w_p(x)v :: atomic function
begin
  x_p ← v
  if ((x, ·) ∈ updates_p) then
     remove (x, ·) from updates_p
  include (x, v) in updates_p
end

r_p(x) :: atomic function
begin
  if (model = sequential) and (updates_p ≠ ∅) and ((x, ·) ∉ updates_p)
then
     wait until turn_p = p
  return(x_p)
end

send_updates() :: atomic task activated whenever turn_p = p
begin
  /* send to all processes, except itself */
  broadcast(updates_p)
  updates_p ← ∅
  turn_p ← (turn_p + 1) mod n
end

apply_updates() :: atomic task activated whenever turn_p = q, p ≠ q, and
the set updates_q from process q is in the receiving buffer of process p
begin
  take updates_q from the receiving buffer
  while updates_q ≠ ∅ do
     extract (x, v) from updates_q
     if (model = causal) or ((x, ·) ∉ updates_p) then
        x_p ← v
  turn_p ← (turn_p + 1) mod n
end
```

Fig. 1. The algorithm $A(\mathsf{model})$ for process p. It is invoked with the parameter model, which defines the consistency model that it must implement.

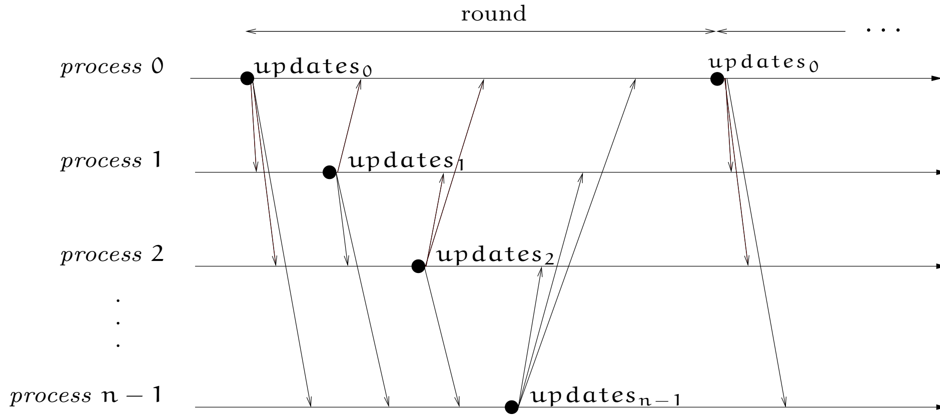

Fig. 2. Cyclic turn fashion.

with a generic broadcast call, which could be simply implemented by sending $n-1$ point-to-point messages if the underlying message passing subsystem does not provide a more appropiate communication primitive. All this is done by the atomic task $\mathsf{send_updates}()$, which also empties the set $\mathsf{updates}_p$. The message sent implicitly passes the turn to the next process in order $(\mathsf{turn}_p + 1) \bmod n$ (see Figure 2).

The atomic task $\mathsf{apply_updates}()$ is the one in charge of applying the updates received from another process q in $\mathsf{updates}_q$. This task is activated whenever $\mathsf{turn}_p = q$ and the set $\mathsf{updates}_q$ is in the receiving buffer of process p. Note that, when implementing sequential and cache consistency, after a local write operation has been performed in some variable, this task will stop applying the write operations on the same variable from other processes. That allows the system to "view" those writes as if they were overwritten with the write value issued by the local process.

Read operations are always fast with causal and cache consistencies. When implementing sequential consistency, a read operation $r_p(x)u$ is fast unless $\mathsf{updates}_p$ contains a pair with a variable different from x. That is, the read operation is not fast only if, since the latest time it held the turn, process p has not issued write operations on x and has issued write operations on other variables. In this case, and only in this case, it is necessary to delay such a read operation until $\mathsf{turn}_p = p$ for the next time (see Fig. 3). Note that this condition is the same as the condition to execute the task $\mathsf{send_updates}()$. We enforce a blocked read operation to have priority over the task $\mathsf{send_updates}()$. Hence, when $\mathsf{turn}_p = p$, a blocked read operation finished before $\mathsf{send_updates}()$ is executed.

We have labeled the code of the read operation as atomic because we do not want it to be executed while the variable $\mathsf{updates}_p$ is manipulated by some other task. However, if the read operation blocks, other tasks are free to access the algorithm variables. In particular, it is necessary that $\mathsf{apply_updates}()$

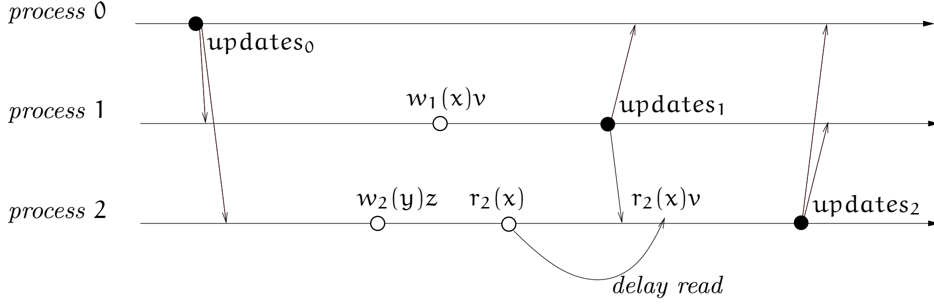

Fig. 3. An example of "non fast" read operation.

updates the variable turn_p for the operation to finish eventually.

4 $A(causal)$ Implements Causal Consistency

In this section, we show that the algorithm A, executed with the parameter $causal$, implements causal consistency. In the rest of this section we assume that α is the set of operations obtained in the execution of the algorithm $A(causal)$, and α_p is the set of operations obtained by removing from α all read operations issued by processes other than p.

Definition 5 *The* i^{th} writes *of process* q*, denoted* $writes_q^i$, $i > 0$*, is the sequence of all write operations of process* q *in* α_p*, in the order they are issued, after* $send_updates()$ *is executed for the* i^{th} *time in this process* q*, and before it is executed for the* $i+1^{st}$ *time.*

For simplicity, we assume that no write operation is issued by any process before it executes $send_updates()$ for the first time. This allows us to consider $writes_p^0$ as the empty sequence. Observe in $A(causal)$ that the $i+1^{st}$ set $updates_q$ broadcasted by process q contains, for each variable, the last (if any) write operation in $writes_q^i$ on that variable.

Then, we construct a sequence β_p that we will show in the following lemmas that preserves $\prec$ and is legal.

Definition 6 *We denote by* β_p *the sequence formed with all operations of* α_p *as follows. Given the sequence of operations issued by* p*, in the order they are issued, we insert the sequence* $writes_q^i$ *in the point of the sequence in which* $apply_updates()$ *is executed with the set* $updates_q$ *for the* $i+1^{st}$ *time, for all* $q \neq p$ *and* $i \geq 0$.

Since the execution of $apply_updates()$ is atomic, it does not overlap any of the operations issued by p, and the placement of every sequence $writes_q^i$ can be easily found.

Lemma 1 *Let* op *and* op' *be two write operations in* α_p *issued by different processes. If* $\mathsf{op} \prec \mathsf{op}'$*, then* $\mathsf{op} \rightarrow \mathsf{op}'$ *in* β_p.

PROOF. From Definition 1, we know there is a related sequence of operations $op^1, op^2, \ldots, op^m$ such that $op^1 = op$, $op^m = op'$, and $op^j \prec_{nt} op^{j+1}$ for $1 \leq j < m$. This sequence can be divided in r subsequences of consecutive operations , s_1, ..., s_r, such that the operations in each subsequence s_i are issued by the same process, the first operation of s_1 is op, the last operation of s_r is op', and for two consecutive subsequences s_i and s_{i+1}, the last operation of s_i writes the value read by the first operation of s_{i+1}. Then, from the algorithm $A(causal)$ it can be seen that the last operations of two consecutive subsequences s_i and s_{i+1}, $i \leq r-1$, belong, from Definition 6, to sequences $write^i_q$ and $write^j_s$ such that either $j > i$, or $j = i$ and $s > q$. Then, $op \rightarrow op'$ in β_p. $\Box$

Lemma 2 β_p *preserves the order* $\prec$

PROOF. Let op and op' be two operations of β_p such that $op \prec op'$. Let us assume by way of contradiction that $op' \rightarrow op$.
Case 1. op and op' are issued by the same process. Let us suppose that they are issued by process q. Recall that α_p only contains write operations of a process different of p. From Definition 5, if $op' \rightarrow op$ is because op' is executed before op, but as we know, from Definition 1, if $op \prec op'$, op must be executed before op'. Hence, we reach a contradiction. Similarly, now let us suppose that op and op' are issued by p. From Definition 6, if $op' \rightarrow op$ is because op' is executed before op, but, from Definition 1, if $op \prec op'$, then op must be executed before op'. Hence, we also reach a contradiction.
Case 2. op and op' are issued by different processes. First, let us suppose that op and op' are operations issued by processes other than p, and differents between them. Then, as α_p only contains read operations of process p, op and op' must be write operations. Therefore, from Lemma 1, $op \rightarrow op'$, and we reach a contradiction.

Now, let us suppose that op is a read operation issued by p, and, as as α_p only contains write operations of processes other than p, op' must be a write operation of a process different of p. We know that if $op \prec op'$, we have a related sequence $op = op^1 \prec_{nt} op^2 \prec_{nt} \ldots \prec_{nt} op^n = op' = w(y)v$. If op^i is the first write operation after op, from Definition 1, op^i has to be issued by p, and op has to be issued before op^i. Therefore, from Definition 6, $op \rightarrow op^i$, and, from Lemma 1, $op^i \rightarrow op'$. Hence, $op \rightarrow op'$, and we reach a contradiction.

Finally, let us suppose that op' is a read operation issued by p, and, as as α_p only contains write operations of processes other than p, op must be a write operation of a process different of p. We know that if $\mathsf{op} \prec \mathsf{op}'$, we have a related sequence $\mathsf{op} = \mathsf{op}^1 = w(x)v \prec_{nt} \ldots \prec_{nt} \mathsf{op}^{n-1} \prec_{nt} \mathsf{op}'$. If $\mathsf{op}^j = w_q(y)v$ is the last write operation before op', from Definition 1, op^j has to be executed before op'. This implies, with our algorithm $A(\mathsf{causal})$, that op^j is propagated to process p before op' is issued. Then, this is because $\mathsf{apply_updates}()$ is executed in process p with the set $\mathsf{updates}_q$ containing op^j before op' is issued. Therefore, from Definition 6, $\mathsf{op}^j \to \mathsf{op}'$, and, from Lemma 1, $\mathsf{op} \to \mathsf{op}^j$. Hence, $\mathsf{op} \to \mathsf{op}'$, and we reach a contradiction. □

Lemma 3 β_p *is legal.*

PROOF. Let us assume, by way of contradiction, that β_p is ilegal because there exists $\mathsf{op}' = w_q(x)u \to \mathsf{op}'' = w_s(x)v \to \mathsf{op} = r_p(x)u$ in β_p. From Definition 6, if op' precedes op'' and op'' precedes op, then we have in process p that: first, op' is issued (or applied if $q \neq p$), next, op'' is issued (or applied if $s \neq p$), and finally, op is issued. From our algorithm $A(\mathsf{causal})$ we can see that, due to these write operations op' and op'', the local copy x_p of x will have the value u and later the value v. We can also see that in $A(\mathsf{causal})$ a read operation always returns the value of the local copy of a variable. Then, it is not possible to have op in β_p after op'', since it would mean that op would have found the value v in x_p, instead of the value u. Hence, we reach a contradiction, and β_p is legal. □

Theorem 1 *The algorithm* $A(\mathsf{causal})$ *implements causal consistency.*

PROOF. From Lemma 2 and Lemma 3, every execution of the algorithm $A(\mathsf{causal})$ has a view β_p of α_p, $\forall p$, that preserves $\prec$ and legal. Hence, from Definition 4, the algorithm $A(\mathsf{causal})$ is causal. □

5 $A(sequential)$ Implements Sequential Consistency

In this section, we show that the algorithm A, executed with the parameter $\mathsf{sequential}$, implements sequential consistency. In the rest of this section we assume that α is the set of operations obtained in the execution of the algorithm $A(\mathsf{sequential})$. Any time reference in this section has to do with the time in which the operations of α are executed. Now we first introduce some definitions of subsets of α.

Definition 7 *The* i^{th} iteration *of process* p*, denoted* it_p^i*,* $i > 0$*, is the subset of* α *that contains all the operations issued by process* p *after* $\mathsf{send_updates}()$ *is executed for the* i^{th} *time, and before it is executed for the* $i + 1^{st}$ *time.*

Observe that any operation in it_p^i finishes before $\mathsf{send_updates}()$ is executed for the $i + 1^{st}$ time, since all write and most read operations are fast, and we assume that blocked read operations have priority over the execution of $\mathsf{send_updates}()$.

Definition 8 *The* i^{th} iteration tail *of process* p*, denoted* $tail_p^i$*, is the subset of* it_p^i *that includes all the operations from the first write operation (included) until the end of* it_p^i*. If* it_p^i *does not contain any write operation,* $tail_p^i$ *is the empty sequence.*

Observe that all write operations in it_p^i are in $tail_p^i$. Furthermore, it is easy to check in $A(sequential)$ that the $i+1^{st}$ set $updates_p$ broadcasted by process p contains, for each variable, the last (if any) write operation in $tail_p^i$.

Definition 9 *The* i^{th} iteration header *of process* p*, denoted* $head_p^i$*, is the subset of* it_p^i *that contains all the operations in* it_p^i *that are not in* $tail_p^i$*.*

It should be clear that all the operations in $head_p^i$ precede all the operations in $tail_p^i$ in the execution of A. We use now the time instants sets received from other processes are applied to partition the sequence $head_p^i$. Note that between the ith and the $i + 1^{st}$ execution of $\mathsf{send_updates}()$ by p (which defines the operations that are in it_p^i, and hence in $head_p^i$) the task $\mathsf{apply_updates}()$ is executed $n - 1$ times, with sets from processes $(p + 1) \bmod n, ..., n - 1, 0, ..., (p - 1) \bmod n$ (in this order).

Definition 10 *The* iteration subheader q *of* $head_p^i$*, denoted* $subhead_{p,q}^i$*, is the subset of* $head_p^i$ *that contains the following operations.*

- *If* $q = p$*, then* $subhead_{p,p}^i$ *contains all the operations issued before* $\mathsf{apply_updates}()$ *is executed with the set* $updates_{(p+1) \bmod n}$*.*
- *If* $q = (p - 1) \bmod n$*, then* $subhead_{p,q}^i$ *contains all the operations issued after* $\mathsf{apply_updates}()$ *is executed with the set* $updates_q$*.*
- *Otherwise,* $subhead_{p,q}^i$ *contains all the operations issued after* $\mathsf{apply_updates}(mess_q)$ *is executed with the set* $updates_q$ *and before it is executed with the set* $updates_{(q+1) \bmod n}$*.*

Clearly, if the first write operation in it_p^i *is issued before* $\mathsf{apply_updates}()$ *is executed with the set* $updates_q$*, then* $subhead_{p,q}^i$ *is the empty sequence (see* it_2^{i-1} *in Fig. 4).*

To simplify the notation and the analysis, we assume that no operation is issued by any process before it executes $\mathsf{send_updates}()$ for the first time.

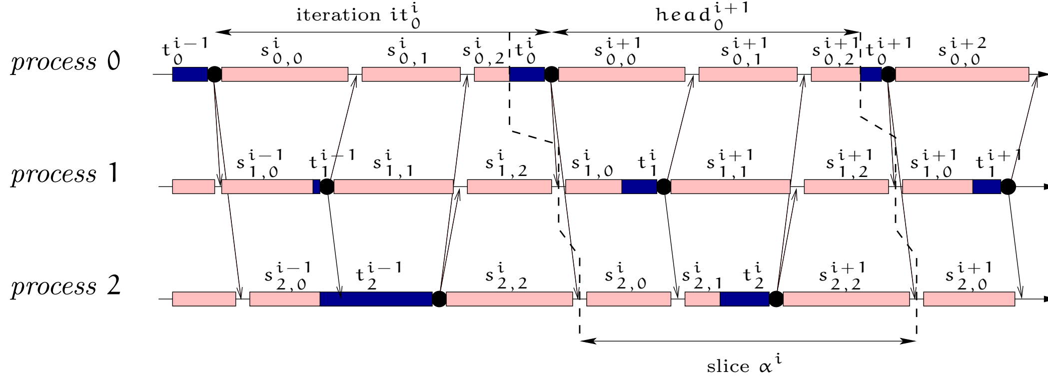

Fig. 4. Iterations and slices. We have abbreviated $\mathtt{tail}$ with $\mathtt{t}$ and $\mathtt{subhead}$ with $\mathtt{s}$.

This allows us to define, for any p and q, the sequences $\mathtt{it}_p^0$, $\mathtt{tail}_p^0$, $\mathtt{head}_p^0$, and $\mathtt{subhead}_{p,q}^0$ as empty sets of operations.

With these definitions, we divide now the set of operations α in in slices. This division is done in such a way that preserves the order in the execution of α (see Fig. 4).

Definition 11 *The* i^{th} slice *of* α*, denoted* α^i*,* $i \geq 0$*, is the subset of* α *formed by the sets of operations* $\mathtt{tail}_p^i, \forall p$*,* $\mathtt{subhead}_{p,q}^i, \forall p, q : p > q$*, and* $\mathtt{subhead}_{p,q}^{i+1}, \forall p, q : p \leq q$*.*

Note that, if we consider α^0 the first slice, every operations in α is in one and only one slice. There are subheaders of iteration 0 that are not assigned to any slice, but since by definition they are empty, they do not need further consideration.

The slice is the basic unit that we will use to define the sequential order that our algorithm enforces. We present now the sequential order for each slice separately. The order for the whole execution is obtained by simply concatenating the slices in their numerical order. However, to complete this sequential order, we yet need to define an order into each subset of operations in tails and subheads that constitute the slice α^i. Then, from now to the rest of this section, we suppose that the operations into any $\mathtt{tail}_p^i$ and $\mathtt{subhead}_{q,p}^i$ are ordered among them as they were issued by process p. Hence, we define now, for each slice α^i, the sequence β^i which contains all the operations of the slice in the sequential order.

Definition 12 *The sequence* β^i *is obtained by ordering the operations into each* $\mathtt{tail}_p^i$ *and* $\mathtt{subhead}_p^j$ *of* α^i *in the order as they were issued by process* p*,*

and by concatenating the set of tails and subheads of α^i as follows.

$$\mathtt{tail}^i_0 \to \mathtt{subhead}^{i+1}_{0,0} \to \mathtt{subhead}^i_{1,0} \to \mathtt{subhead}^i_{2,0} \to ... \to \mathtt{subhead}^i_{n-1,0} \to$$
$$\mathtt{tail}^i_1 \to \mathtt{subhead}^{i+1}_{0,1} \to \mathtt{subhead}^{i+1}_{1,1} \to \mathtt{subhead}^i_{2,1} \to ... \to \mathtt{subhead}^i_{n-1,1} \to$$
$$\dots$$
$$\mathtt{tail}^i_p \to \mathtt{subhead}^{i+1}_{0,p} \to ... \to \mathtt{subhead}^{i+1}_{p,p} \to \mathtt{subhead}^i_{p+1,p} \to ... \to \mathtt{subhead}^i_{n-1,p} \to$$
$$\dots$$
$$\mathtt{tail}^i_{n-1} \to \mathtt{subhead}^{i+1}_{0,n-1} \to \mathtt{subhead}^{i+1}_{1,n-1} \to \mathtt{subhead}^{i+1}_{2,n-1} \to ... \to \mathtt{subhead}^{i+1}_{n-1,n-1}$$

In fact, this is only one of many ways to order the sequences of the slice to obtain a sequential order. All the subheaders that appear above in the same line could be permuted in any possible way, since they only contain read operations and each contains operations from a different process. We choose the above order for simplicity.

We define now the sequence β.

Definition 13 β *is the sequence of* α *obtained by the concatenation of all sequences* β^i *in order (i.e.,* $\beta^i \to \beta^{i+1}, \forall i \geq 0$*).*

From the above definitions, in β, we have that $\mathtt{tail}^i_p \to \mathtt{tail}^j_q$ if and only if either $i < j$ or $i = j$ and $p < q$. This is exactly the order in which the sets associated with each tail are processed and applied in the algorithm.

We show in the following lemmas that β is in fact a view that preserves the order $\prec$ and is legal.

Lemma 4 β *preserves the order* $\prec$*.*

PROOF. Let op and op' be two operations of β such that $op \prec op'$. From Definition 1, we know that there is a related sequence of operations $op^1, op^2, \dots, op^m$ such that $op^1 = op$, $op^m = op'$, and $op^j \prec_{nt} op^{j+1}$ for $1 \leq j < m$. If β preserves $\prec$, then $op^j \to op^{j+1}$, $\forall j$, and, hence, $op \to op'$. We consider several cases.
Case 1. op^j and op^{j+1} are operations issued by the same process. If $op^j \prec op^{j+1}$, from Definition 1, op^j must be issued before op^{j+1}. Then, it is easy to check from the above definitions of β and β^i that operations from the same process appear in the same order in β as they were issued. Then, $op^j \to op^{j+1}$, $\forall j$. Hence, $op \to op'$.

Case 2. op^j and op^{j+1} are a write operation and a read operation, respectively, issued by different processes. Let us suppose that, from Definition 1, $op^j = w_q(x)u$ and $op^{j+1} = r_s(x)u$. We know that if $op^j \prec op^{j+1}$, then op^j must be executed before op^{j+1}. From our algorithm $A(sequential)$ and from the above definitions of β and β^i, we can see that op^j always belongs to $tail^i_q$, and in the case of op^{j+1} we have two possibilities: a) op^{j+1} belongs to $subhead^j_{s,l}$, $i < j$, or if $i = j$ $q \leq l$; or b) op^{j+1} belongs to $tail^j_p$, $i < j$. In both cases, $op^j \rightarrow op^{j+1}$, $\forall j$. Hence, $op \rightarrow op'$. $\square$

Lemma 5 β *is legal.*

PROOF. Let us suppose that there exists $op' = w(x)v \rightarrow op = r(x)v$ in β. From definition Definition 3, β is legal if $\nexists op'' = w(x)u \in \alpha$ such that $op' \rightarrow op'' \rightarrow op$ in β. Then, this is equivalent to say that β is legal if for every read operation $op = r(x)v$ in β, the nearest previous write operation in β on the variable x is $op' = w(x)v$.

Let us assume op is issued by process p. Note first that the order in which the iteration tails appear in β is exactly the order imposed by the token passing procedure. Then, in p, the order in β reflects exactly the order in which the sets $updates_q$ are applied in the local memory of p. The only exceptions are the sets $updates_p$, since the write operations of p itself, are applied in its local memory immediately, and do not wait until p holds the token. However, note that any update from other processes on a variable written locally is not applied (see $apply_updates()$). This gives the illusion that the local write operations have in fact been applied at the time of p's token possession. Then we consider several cases.

Case 1. Both op and op' belong to the same iteration tail $tail^i_p$. When issued by p, op' sets the value of the local local copy x_p of x. After op' is executed, $(x, \cdot) \in updates_p$, and no update applied from other process changes this value (see $apply_updates()$). Hence, if op returns the value v is because op' wrote the value v in x.

Case 2. op belongs to an iteration subheader $subheader^i_{p,q}$. The value v returned by op is the value of x_p after locally applying the write operations in the following tails.

- If $p > q$, $tail^j_r$ for each $j < i$, and for each $r \leq q$ when $j = i$.
- If $p \leq q$, $tail^j_r$ for each $j < i - 1$, and for each $r \leq q$ when $j = i - 1$.

These are the tails that precede $subheader^i_{p,q}$ in β As we said above, these tails are applied in the order they appear in β. Then, v has to be the value written by the nearest write operation on x that precedes op in β, which by definition is op'.

Case 3. op belongs to an iteration tail $tail^i_p$, while op belongs to a different

iteration tail. Then the read operation op was issued when p had already issued a write operation (since it belong to a tail) on a variable different from x (by definition of op'). Then, op was blocked until the token was assigned to p. The value v returned by op is the value of x_p after locally applying the write operations in the tails tail_q^j for each $j < i$ and for each $q < p$ when $j = i$, which are the tails that precede tail_p^i in β. As we said above, these tails are applied in the order they appear in β. Then, v has to be the value written by the nearest write operation on x that precedes op in β, which by definition is op'.

Thus, in the above three cases we have shown that $\mathsf{op}' = w(x)v$ is the nearest write operation on variable x previous to $\mathsf{op} = r(x)v$ in β. Hence, β is legal. □

Theorem 2 *The algorithm* $A(\mathsf{sequential})$ *implements sequential consistency.*

PROOF. From Lemma 4 and From Lemma 5, every execution of the algorithm $A(\mathsf{sequential})$ has a view β of α that preserves the order $\prec$ and legal. Hence, from Definition 4, the algorithm $A(\mathsf{sequential})$ is sequential. □

6 $A(\mathsf{cache})$ Implements Cache Consistency

In this section, we show that the algorithm A, executed with the parameter cache in each process, implements cache consistency. In the rest of this section we assume that α is a set of operations produced in the execution of the algorithm $A(\mathsf{cache})$, and $\alpha(x)$ is a set of operations formed by all operations in α on the variable x.

The proof of correcteness follows the same lines as the proof of correctness for $A(\mathsf{sequential})$, but on $\alpha(x)$ instead of α. First we define the sequences $\mathsf{it}(x)_p^i$, $\mathsf{tail}(x)_p^i$, $\mathsf{head}(x)_p^i$, $\mathsf{subhead}(x)_{p,q}^i$, and the slice $\alpha(x)^i$ of $\alpha(x)$. Then we construct the sequence $\beta(x)$ from these sequences in a similar way as the sequence β was defined in Section 5. A version for $\beta(x)$ of Lemma 4 is directly derived. In a version for $\beta(x)$ of Lemma 5 with the above sequences the case 3 dissapears. Hence we have that $\beta(x)$ is a view of $\alpha(x)$ that preserves the order α and legal. Since this is true for any variable x, we have the following theorem.

Theorem 3 *The algorithm* $A(\mathsf{cache})$ *implements cache consistency.*

PROOF. From Lemma 4 (but only with operations of $\alpha(x)$) and From Lemma 5 (only with operations of $\alpha(x)$), every execution of the algorithm

$A(cache)$ has a view $\beta(x)$ of $\alpha(x)$, $\forall x$, that preserves the order $\prec$ and legal. Hence, from Definition 4, the algorithm $A(cache)$ is cache. $\square$

7 Complexity Measures

7.1 Worst-Case Response Time

In this section we consider that local operations are executed instantaneously (i.e., in 0 time units) and that any communication takes d time units. In the algorithm A executed with parameter $causal$ or $cache$ all operations are executed locally, while when executed with parameter $sequential$ all write and some read operations are also executed locally. Therefore, the response time for them is always 0.

Let us now consider a read operation that is blocked in algorithm $A(sequential)$. To obtain the maximum response time for such a read operation, we will consider the worst case. This can happen if the operation blocks (almost) immediately after the process that issued it sent a message. Then, the read operation will be blocked until the turn of this process again, which can take up to n message transmissions. Therefore, in the worst case, a process will have to wait nd time units.

The previous analysis assumes that the messages are never delayed at the processes. However, the protocol allows the processes to control when to send the messages. For instance, it is possible for a process p, when $turn_p = p$, to wait a time T before executing its task $send_updates()$ (see Fig. 1). Thus, we can reduce the number of messages sent by this process per unit of time. Obviously, this can increase the response time, since in this case the delay time of a message sent by p, in the worst case, will be $T + d$.

7.2 Message Size

It is easy to check in Fig. 1 that the size of the list $updates_p$ of process p depends on the number of write operations performed by p during each round, which can be very high. However, the number of pairs (x, v) in $updates_p$ will be, at most, the same as the number of shared variables, since we only hold at most one pair for each variable.

The bound obtained may seem extremely bad. However, note that the real number of pairs in a set $updates_p$ really depends on the frequency f of write operations and the rotation time nd. Hence if we have a write operation on a

variable every milisecond, in a system with 100 processes and 1 milisecond of delay, we will have at most 100 pairs in the set $\mathtt{updates}_p$ broadcasted, which is a reasonable number.

Furthermore, note that most algorithms that implement propagation and full replication send a message for *every* write operation that is performed. This would mean that 100 messages would have to be sent. With our algorithm, only one pair per variable is sent, and all of them are grouped into one single message. With the overhead per message in current networks, this implies a significant saving in bandwidth.

7.3 Memory Space

Finally, note that we do not require the communication channels among processes to deliver messages in order. Hence, a process could have received messages that are held until the message from the appropriate process arrives. It is easy to check that the maximum number of messages that will ever be held is $n - 2$.

8 Consistency in A with different parameters

In this section we show that the algorithm A, executed in some processes with the parameter $\mathtt{sequential}$ and with $\mathtt{causal}$ in the rest of processes, implements causal consistency. Also we prove in this section that if there are processes executig $A(\mathtt{sequential})$ and others $A(\mathtt{cache})$, the algorithn A implements cache consistency.

In this part of this section we assume that α is the set of operations obtained in the execution of the algorithm $A(\mathtt{sequential})$ and $A(\mathtt{causal})$, and α_p is the set of operations obtained by removing from α all read operations issued by processes other than p.

Theorem 4 *The algorithm A implements causal consistency when there exist processes executing $A(\mathtt{sequential})$ and others $A(\mathtt{causal})$.*

PROOF. We can see in the Figure 1 that $A(\mathtt{causal})$ and $A(\mathtt{sequential})$ have the same mechanism to propagate the write operations. Besides, in both algorithms read operations are always done locally (without generating messages) in the process where they were issued.

Recall that a sequence β_p is formed with every write operation issued by any process and with every read operation issued only by process p. Hence, from the point of view of each process executing $A(\mathsf{causal})$, the existence of processes executing $A(\mathsf{sequential})$ neither include nor modify the set of operations α_p to order in β_p different from other process executing $A(\mathsf{causal})$. Therefore, if we now construct β_p for each process p executing $A(\mathsf{causal})$ as we did in Definition 6, β_p will remain legal and preserving the order $\prec$.

Now, in the following two definitions, we redefine β_p for each process p executing $A(\mathsf{sequential})$. After that, we will show that this new sequence β_p preserves $\prec$ and is legal.

For construct β_p we use, from Section 5, the notation of slice, but now on α_p instead of α, to see how the set of operations of α_p is divided. We also use writes_p^i and $\mathsf{subhead}_{p,q}^i$ as we defined in Section 4 and Section 5, respectively, to see how the set of operations of α_p^i is ordered.

Definition 14 *The* i^{th} slice *of* α_p*, denoted* α_p^i*,* $i \geq 0$*, is the subset of* α_p *formed by the sets of operations* $\mathsf{writes}_q^i, \forall q$*,* $\mathsf{subhead}_{q,p}^i, \forall q : q > p$*, and* $\mathsf{subhead}_{q,p}^{i+1}, \forall q : q \leq p$*.*

We denote by β_p^i the sequence of all operations of the slice α_p^i, and by β_p the sequence of α_p formed by the concatenation of β_p^i in increasing numerical order

Definition 15 *The sequence* β_p^i*,* $\forall p$ *executing* $A(\mathsf{sequential})$*, is obtained by ordering the operations into each* writes_p^i *and* $\mathsf{subhead}_p^j$ *of* α_p^i *in the order as they were issued by process* p*, and by concatenating the set of writes and subheads of* α_p^i *as follows.*
$\mathsf{writes}_0^i \rightarrow \mathsf{subhead}_{p,0}^i \rightarrow$
$\mathsf{writes}_1^i \rightarrow \mathsf{subhead}_{p,1}^i \rightarrow$
$\dots$
$\mathsf{tail}_p^i \rightarrow \mathsf{subhead}_{p,p}^{i+1} \rightarrow$
$\dots$
$\mathsf{writes}_{n-1}^i \rightarrow \mathsf{subhead}_{p,n-1}^{i+1}$

Definition 16 *The sequence* β_p*,* $\forall p$ *executing* $A(\mathsf{sequential})$*, is the sequence of* α *obtained by the concatenation of all sequences* β_p^i *in order (i.e.,* $\beta_p^i \rightarrow \beta_p^{i+1}, \forall i \geq 0$*).*

As we can see comparing Definition 12 and Definition 15, β_p^i is the same sequence that β^i (and, therefore, β_p and β) but with the following two differences:

- writes_q^i, $q \neq p$, is the same sequence that it_q^i where we have removed every

read operation issued by q.

- We have removed every subhead with operations other than process p. That is to say, every $\mathsf{subhead}^j_{q,n}$ such that $q \neq p$, $j = i$ or $j = i + 1$, and $\forall n$.

As we can see, we have constructed β_p in a similar way as the sequence β was defined in Section 5. Then, all operations belonging to β_p are in in the same order as in the definition of β, and a version for β_p of Lemma 4 is directly derived. Hence, β_p preserves the order $\prec$.

Similarly, we know that all write operations are in β_p in the same order as in the definition of β. Then, a version for β_p of Lemma 5 is also directly derived. Hence, β_p is legal.

Thus, we have for each process p executing $A(\mathsf{causal})$ a β_p formed as we described in Section 4. We also have for each process q executing $A(\mathsf{sequential})$ a β_q formed as Definition 16. As in both cases β_p is legal and preserves $\prec$, we can affirm that A implements causal consistency when there are processes executing $A(\mathsf{sequential})$ and others $A(\mathsf{causal})$. □

In this part of this section we assume that α is the set of operations obtained in the execution of the algorithm $A(\mathsf{sequential})$ and $A(\mathsf{cache})$, and $\alpha(x)$ is the set of operations of α on variable x.

Theorem 5 *The algorithm A implements cache consistency when there exist processes executing $A(\mathsf{sequential})$ and others $A(\mathsf{cache})$.*

PROOF. We can see in the Figure 1 that $A(\mathsf{cache})$ and $A(\mathsf{sequential})$ are the same algorithm except in one case: when a read operation is not fast. Therefore, they use the same mechanism to propagate the write operations, and read operations do not add new messages because they are done locally in the process where it is invoked.

Hence, from the point of view of each process executing $A(\mathsf{cache})$, the existence of processes executing $A(\mathsf{sequential})$ neither include nor modify the set of operations to order in $\beta(x)$ different from other process executing $A(\mathsf{cache})$. Therefore, if we now construct $\beta(x)$ for each process p executing $A(\mathsf{cache})$ as we did in Section 6, $\beta(x)$ will remain legal and preserving $\prec$.

Now, we are going to use the same definition and way of construction of $\beta(x)$ from Section 6 for all processes that executes $A(\mathsf{sequential})$. Then, the difference between the sequence $\beta(x)$ for processes executing $A(\mathsf{sequential})$ with respect to processes executing $A(\mathsf{cache})$ is what happen when a non–fast read operation occurs. To analyse this case, we use the same notation from Section 6 to define a slice $\alpha^i(x)$, a tail $\mathsf{tail}^i_p(x)$, and a subhead $\mathsf{subhead}^j_{q,p}(x)$.

Then, let us suppose that a non–fast read operation happens in a process p executing $A(sequential)$ in the slice $\alpha^i(x)$. We know, by definition of $\beta(x)$, that this read operation belongs to $tail_p^i(x)$. We are also going to suppose, that the first operation of $tail_p^i(x)$ is done just after $subhead_{q,p}^j(x)$, $i = j$ if $q > p$, or $j = i + 1$ if $q \leq p$. Hence, in this case, the unique difference with respect to an execution in a process $A(cache)$ is that each subhead subsequent to $subhead_{q,p}^j(x)$ in $\alpha^i(x)$ is always empty. Therefore, as the sequence $\beta(x)$ of a process executing $A(cache)$ from Section 6, A version of Lemma 4 and Lema 5 for $\beta(x)$ of a process executing $A(sequential)$ are directly derived. Hence, $\beta(x)$ of a process executing $A(sequential)$ is legal and preserves the order $\prec$.

Thus, we have shown for each process p executing $A(cache)$ or $A(sequential)$ that there is a $\beta(x)$, $\forall x$, formed as we described in Section 6, such that preserves the order $\prec$ and legal. Therefore, A implements cache consistency when there exist processes executing $A(sequential)$ and others $A(cache)$. $\Box$

9 Conclusions and Future Work

In this paper, we have presented a parametrized algorithm that implements sequential, causal, and cache consistency on a distributed system. To our knowledge, this is the first algorithm that implements cache consistency.

The algorithm presented in this paper guarantees fast operations in its causal and cache executions. It is proven in [7] that it is imposible to have a sequential algorithm with all operations fast. The algorithm presented in this paper guarantees in its sequential execution fast writes and reduces to only one case the reads that can not be executed locally.

Considering possible extensions of this work for the sequential version, we would like to know how many read operations are fast in real applications with several system parameters. Our belief is that most read operations will be fast. A second line of work has to do with the scalability of the protocol. The worst case response time is linear on the number of processes. Hence, it will not scale well, since it may become high when the system has a large number of processes. It would be nice to remove this dependency. Finally, the protocol works in a token passing fashion, which can be very risky in an environment with failures, since a single failure can block the whole system. It would be interesting to extend the protocol with fault tolerance features.

References

[1] R. C. Steinke, G. J. Nutt, A unified theory of shared memory consistency, J. ACM 51 (5) (2004) 800–849.

[2] C. Manovit, S. Hangal, Efficient algorithms for verifying memory consistency, in: SPAA'05: Proceedings of the 17th annual ACM symposium on Parallelism in algorithms and architectures, ACM Press, New York, NY, USA, 2005, pp. 245–252.

[3] M. Ahamad, G. Neiger, J. Burns, P. Kohli, P. Hutto, Causal memory: Definitions, implementation and programming, Distributed Computing 9 (1) (1995) 37–49.

[4] R. Prakash, M. Raynal, M. Singhal, An adaptive causal ordering algorithm suited to mobile computing environments, Journal of Parallel and Distributed Computing 41 (1997) 190–204.

[5] M. Raynal, M. Ahamad, Exploiting write semantics in implementing partially replicated causal objects, in: Proceedings of the 6th EUROMICRO Conference on Parallel and Distributed Computing, 1998, pp. 157–163.

[6] L. Lamport, How to make a multiprocessor computer that correctly executes multiprocess programs, IEEE Transactions on Computers 28 (9) (1979) 690–691.

[7] H. Attiya, J. Welch, Sequential consistency versus linearizability, ACM Transactions on Computer Systems 12 (2) (1994) 91–122.

[8] M. Raynal, Token-based sequential consistency., Comput. Syst. Sci. Eng. 17 (6) (2002) 359–365.

[9] M. Raynal, K. Vidyasankar, A distributed implementation of sequential consistency with multi-object operations., in: ICDCS, 2004, pp. 544–551.

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

[11] H. Attiya, R. Friedman, A correctness condition for high-performance multiprocessors, in: Proceedings of the 24th Annual ACM Symposium on the Theory of Computing, 1992, pp. 679–690.

[12] J. Misra, Axioms for memory access in asynchronous hardware systems, ACM Transactions on Programming Languages and Systems 8 (1) (1986) 142–153.

[13] M. Raynal, A. Schiper, From causal consistency to sequential consistency in shared memory systems, Tech. Rep. 926, IRISA (May 1996).

[14] G. B. Yehuda Afek, M. Merritt, Lazy caching, ACM Transactions on Programming Languages and Systems 15 (1) (1993) 182–205.

[15] A. Singh, Bounded timestamps in process networks, Parallel Processing Letters 6 (2) (1996) 259–264.