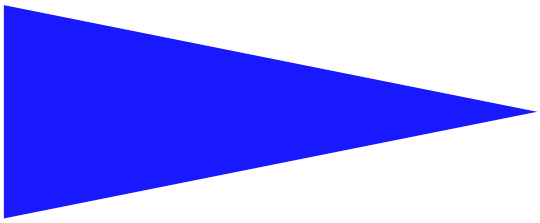
About the efficiency of partial replication to implement Distributed Shared Memory

Jean-Michel Hélary, Alessia Milani

I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

P U B L I C A T I O N
I N T E R N E
N° 1727



ABOUT THE EFFICIENCY OF PARTIAL REPLICATION TO IMPLEMENT DISTRIBUTED SHARED MEMORY

JEAN-MICHEL HÉLARY , ALESSIA MILANI

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

About the efficiency of partial replication to implement Distributed Shared Memory

Jean-Michel Hélary* , Alessia Milani**

Systèmes communicants

Publication interne n ° 1727 — juin 2005 — 20 pages

Abstract: Distributed Shared Memory abstraction (DSM) is traditionally realized through a distributed *memory consistency system* (MCS) on top of a message passing system. In this paper we analyze the impossibility of efficient partial replication implementation of causally consistent DSM. Efficiency is discussed in terms of control information that processes have to propagate to maintain consistency. We introduce the notions of share graph and hoop to model variable distribution and the concept of dependency chain to characterize processes that have to manage information about a variable even though they do not read or write that variable. Then, we weaken causal consistency to try to define new consistency criteria weaker enough to allow efficient partial replication implementations and strong enough to solve interesting problems. Finally, we prove that PRAM is such a criterion, and illustrate its power with the Bellman-Ford shortest path algorithm.

Key-words: Distributed shared memory, partial replication, consistency criterion, causal consistency, PRAM consistency, shortest path algorithm.

(Résumé : tsvp)

We like to thank Michel Raynal for suggesting this subject of research and for insightful discussions on this work.

* IRISA, Campus de Beaulieu, 35042 Rennes-cedex, France.helary@irisa.fr
** DIS, Universitá di Roma *La Sapienza*, Via Salaria 113, Roma, Italia.Alessia.Milani@dis.uniroma1.it

Centre National de la Recherche Scientifique
(UMR 6074) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Sur l'efficacité de la duplication partielle pour l'implémentation des mémoires partagées réparties

Résumé : Les mémoires partagées réparties constituent une abstraction qui est traditionellement concrétisée par un *système réparti de mémoire cohérente*, au-dessus d'un système de communication par messages. Dans ce rapport, on analyse l'impossibilité d'avoir une implémentation efficace de mémoire partagée répartie à cohérence causale, basée sur la duplication partielle des variables. L'efficacité est envisagée en terme d'information contrôle qui doit être propagée pour assurer la cohérence. On introduit les notions de *graphe de partage* et d'*arceau*, qui modélisent la répartition des variables et la notion de *chaîne de dépendance* pour caractériser les processus qui doivent gérer des informations relatives à une variable dont ils ne possèdent pas de copie locale. Ensuite, on affaiblit le critère de cohérence causale, dans le but de déterminer un nouveau critère de cohérence qui soit suffisament faible pour permettre un implémentation efficace basée sur la duplication partielle, mais suffisament forte pour pouvoir résoudre des problèmes intéressants. Finalement, on prouve que le critère appelé PRAM satisfait ces exigences, et illustrons sa pertinence en montrant une implémentation de l'algorithme de plus court chemin de Bellman-Ford.

Mots clés : Mémoire partagée répartie, duplication partielle, critères de cohérence, cohérence causale, cohérence PRAM, algorithme de plus court chemin.

1 Introduction

Distributed Shared Memory (DSM) is one of the most interesting abstraction providing data-centric communication among a set of application processes which are decoupled in time, space and flow. This abstraction allows programmers to design solutions by considering the well-known shared variables programming paradigm, independently of the system (centralized or distributed) that will run his program. Moreover, there are a lot of problems (in numerical analysis, image or signal processing, to cite just a few) that are easier to solve by using the shared variables paradigm rather than using the message passing one.

Distributed shared memory abstraction is traditionally realized through a distributed *memory consistency system* (MCS) on top of a message passing system providing a communication primitive with a certain quality of service in terms of ordering and reliability [5]. Such a system consists of a collection of nodes. On each node there is an application process and a MCS process. An application process invokes an operation through its local MCS process which is in charge of the actual execution of the operation. To improve performance, the implementation of MCS is based on replication of variables at MCS processes and propagation of the variable updates [9]. As variables can be concurrently accessed (by read and write operations), users must be provided with a consistency criterion that precisely defines the semantics of the shared memory. Such a criterion defines the values returned by each read operation executed on the shared memory.

Many consistency criteria have been considered, e.g., from more to less constraining ones: Atomic [12], Sequential [11], Causal [3] and PRAM [13]. Less constraining MCS are easier to implement, but, conversely, they offer a more restricted programming model. The Causal consistency model has gained interest because it offers a good tradeoff between memory access order constraints and the complexity of the programming model as well as of the complexity of the memory model itself.

To improve performance, MCS enforcing Causal (or stronger) consistency have been usually implemented by protocols based on complete replication of memory locations [10, 4, 8], i.e. each MCS process manages a copy of each shared variable. It is easy to notice that in the case of complete replication, dealing with a large number of shared variables avoids scalability. Thus, in large scale systems, implementations based on partial replication, i.e. each process manages only a subset of shared variables, seems to be more reasonable. Since each process in the system could be justifiably interested only in a subset of shared variables, partial replication is intended to avoid a process to manage information it is not interested in. In this sense, partial replication loses its meaning if to provide consistent values to the corresponding application process, each MCS process has to consider information about variables that the corresponding application process will never read or write. Some implementations are based on partial replication [7, 14], but they suffer this drawback.

In this paper we study the problem of maintaining consistency in a partial replicated environment. More precisely, according to the variables distribution and to the consistency criterion chosen, we discuss the possibility of an *efficient partial replication implementation*, i.e., for each shared variable, only MCS processes owning a local copy have to manage information concerning this variable. Our study shows that MCS enforcing Causal consistency

criterion (or stronger consistency criteria) have no efficient partial replication implementation. Then, several weaker criteria are considered, but all suffer the same drawback. Finally, it is shown that the PRAM consistency criterion is weak enough to allow efficient partial replication implementation. To motivate the interest of this result, the Bellman-Ford algorithm to find the shortest paths issued from a node is realized in an MCS enforcing PRAM consistency criterion, and partial replication of variables.

The rest of the paper is organized as follows. In Section 2 we present the shared memory model. In Section 3, we discuss partial replication issues and we present our main result, namely a characterization for the possibility of efficient partial replication implementation. Section 4 shows some usual consistency criteria for which no efficient partial replication implementation is possible. Finally, Section 5 is devoted to the PRAM consistency criterion and Section 6 to the solution of Bellman-Ford algorithm in such a MCS.

2 The Shared Memory Model

We consider a finite set of sequential *application* processes $\Pi=\{ap_1, ap_2, \ldots ap_n\}$ interacting via a finite set of shared variables, $\mathcal{X}=\{x_1, x_2, ...x_m\}$. Each variable x_h can be accessed through *read* and *write* operations. A write operation invoked by an application process ap_i, denoted $w_i(x_h)v$, stores a new value v in variable x_h. A read operation invoked by an application process ap_i, denoted $r_i(x_h)v$, returns to ap_i the value v stored in variable x_h[1]. Each variable has an initial value $\perp$.

A *local history* of an application process ap_i, denoted h_i, is a sequence of read and write operations performed by ap_i. If an operation o_1 precedes an operation o_2 in h_i, we say that o_1 precedes o_2 in *program order*. This precedence relation, denoted by $o_1 \mapsto_i o_2$, is a total order. A *history* $H=\langle h_1, h_2, \ldots h_n\rangle$ is the collection of local histories, one for each application process. The set of operations in a history H is denoted O_H.

Operations done by distinct application processes can be related by the *read-from order* relation. Given two operations o_1 and o_2 in O_H, the read-from order relation, $\mapsto_{ro}$, on some history H is any relation with the following properties [3][2]:

- if $o_1 \mapsto_{ro} o_2$, then there are x and v such that $o_1 = w(x)v$ and $o_2 = r(x)v$;
- for any operation o_2, there is at most one operation o_1 such that $o_1 \mapsto_{ro} o_2$;
- if $o_2 = r(x)v$ for some x and there is no operation o_1 such that $o_1 \mapsto_{ro} o_2$, then $v = \perp$; that is, a read with no write must read the initial value.

Finally, given a history H, the causality order $\mapsto_{co}$, [3], is a partial order that is the transitive closure of the union of the history's program order and the read-from order.

[1] Whenever we are not interested in pointing out the value or the variable or the process identifier, we omit it in the notation of the operation. For example w represents a generic write operation while w_i represents a write operation invoked by the application process ap_i, etc.

[2] It must be noted that the read-from order relation just introduced is the same as the writes-into relation defined in [3].

Formally, given two operations o_1 and o_2 in O_H, $o_1 \mapsto_{co} o_2$ if and only if one of the following cases holds:

- $\exists\ ap_i$ s.t. $o_1 \mapsto_i o_2$ (program order),
- $\exists\ ap_i, ap_j$ s.t. o_1 is invoked by ap_i, o_2 is invoked by ap_j and $o_1 \mapsto_{ro} o_2$ (read-from order),
- $\exists\ o_3 \in O_H$ s.t. $o_1 \mapsto_{co} o_3$ and $o_3 \mapsto_{co} o_2$ (transitive closure).

If o_1 and o_2 are two operations belonging to O_H, we say that o_1 and o_2 are *concurrent* w.r.t. $\mapsto_{co}$, denoted $o_1 \ ||_{co}\ o_2$, if and only if $\neg(o_1 \mapsto_{co} o_2)$ and $\neg(o_2 \mapsto_{co} o_1)$.

Properties of a history

Definition 1 (Serialization). *Given a history H, S is a serialization of H if S is a sequence containing exactly the operations of H such that each read operation of a variable x returns the value written by the most recent precedent write on x in S.*

A serialization S respects a given order if, for any two operations o_1 and o_2 in S, o_1 precedes o_2 in that order implies that o_1 precedes o_2 in S.

Let H_{i+w} be the history containing all operation in h_i and all write operations of H.

Definition 2 (Causally Consistent History [3]). *A* history *H is causal consistent if for each application process ap_i there is a serialization S_i of H_{i+w} that respects $\mapsto_{co}$.*

A memory is causal if it admits only causally consistent histories.

3 The problem of efficient partial replication implementation of causal memories

In this section we analyze the efficiency of implementing causal memories when each application process ap_i accesses only a subset of the shared variables $\mathcal{X}$, denoted $\mathcal{X}_i$. Assuming a partial replicated environment means that each MCS process p_i manages a replica of a variable x iff $x \in \mathcal{X}_i$. Our aim is to determine which MCS processes are concerned by information on the occurrence of operations performed on the variable x in the system. More precisely, given a variable x, we will say that a MCS process p_i is *x-relevant* if, in at least one history, it has to transmit some information on the occurrence of operations performed on variable x in this history, to ensure a causally consistent shared memory. Of course, each process managing a replica of x is x-relevant. Ideally, we would like that only those processes are x-relevant. But unfortunately, as will be proved in this section, if the variable distribution is not known a priori, it is not possible for the MCS to ensure a causally consistent shared memory, if each MCS process p_i only manages information about $\mathcal{X}_i$. The main result of the section is a characterization of x-relevant processes.

To this aim, we first introduce the notion of *share graph*, denoted SG, to characterize variable distribution and then we define the concepts of *hoop* and of *dependency chain* to highlight how particular variables distribution can impose global information propagation.

3.1 The share graph, hoops and dependency chains

The share graph is an undirected (symmetric) graph whose vertices are *processes*, and an edge (i, j) exists between p_i and p_j iff there exists a variable x replicated both on p_i and p_j (i.e. $x \in \mathcal{X}_i \cap \mathcal{X}_j$). Possibly, each edge (i, j) is labelled with the set of variables replicated both on p_i and p_j.

Figure 1 depicts an example of share graph representing a system of three processes p_i, p_j and p_k interacting through the following set of shared variables $\mathcal{X} = \{x_1, x_2\}$. In particular, $\mathcal{X}_i = \{x_1,\ x_2\}$, $\mathcal{X}_k = \{x_2\}$ and $\mathcal{X}_j = \{x_1\}$.

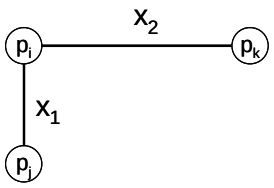


Figure 1: A share graph

It is simple to notice that each variable x defines a sub-graph $C(x)$ of SG spanned by the processes on which x is replicated (and the edges having x on their label). This subgraph $C(x)$ is a clique, i.e. there is an edge between every pair of vertices. The "share graph" is the union of all cliques $C(x)$. Formally, $SG = \bigcup_{x \in \mathcal{X}} C(x)$.

In the example depicted in Figure 1, we have the following cliques:
i) $C(x_1) = (V_{x_1}, E_{x_1})$ where $V_{x_1} = \{p_i,\ p_j\}$ and $E_{x_1} = \{(i, j)\}$,
ii) $C(x_2) = (V_{x_2}, E_{x_2})$ where $V_{x_2} = \{p_i,\ p_k\}$ and $E_{x_2} = \{(i, k)\}$.

Given a variable x, we call *x-hoop*, any path of SG, between two distinct processes in $C(x)$, whose intermediate vertices do not belong to $C(x)$ (figure 2). Formally:

Definition 3 (Hoop). *Given a variable x and two processes p_a and p_b in $C(x)$, we say that there is a x-hoop between p_a and p_b (or simply a* hoop*, if no confusion arises), if there exists a path $[p_a = p_0, p_1, \ldots, p_k = p_b]$ in SG such that:*
i) $p_h \notin C(x)$ $(1 \leq h \leq k \perp 1)$ and
ii) each consecutive pair (p_{h-1}, p_h) shares a variable x_h such that $x_h \neq x$ $(1 \leq h \leq k)$

Let us remark that the notion of hoop depends only on the distribution of variables on the processes, i.e. on the topology of the corresponding share graph. In particular, it is independent of any particular history.

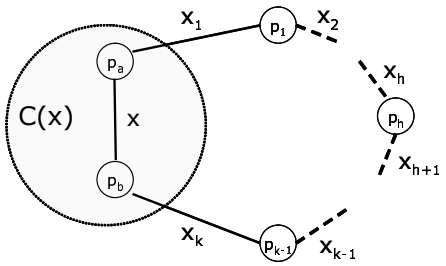


Figure 2: An x-hoop

The following concept of *dependency chain along an hoop* captures the dependencies that can be created between operations occurring in a history, when these operations are performed by processes belonging to a hoop.

Definition 4 (Dependency chain). *Let $[p_a, \ldots, p_b]$ be a x-hoop in a share graph SG. Let H be a history. We say that H includes a x-dependency chain[3] along this hoop if the following three conditions are verified:*

1. *O_H includes $w_a(x)v$, and*
2. *O_H includes $o_b(x)$, where o_b can be a read or a write on x, and*
3. *O_H includes a pattern of operations, at least one for each process belonging to the hoop, that implies $w_a(x)v \mapsto_{co} o_b(x)$.*

More precisely, we also say that $w_a(x)v$ and $o_b(x)$ are the *initial* and the *final* operations of the x-dependency chain from $w_a(x)v$ to $o_b(x)$. Figure 3 depicts such a dependency chain.

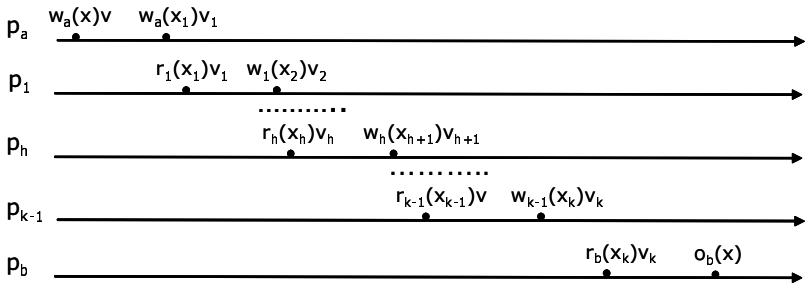


Figure 3: An x-dependency chain from $w_a(x)v$ to $o_b(x)$

[3] simply dependency chain when confusion cannot arise

3.2 A characterization of x-relevant processes

In this section, a precise characterization of x-relevant processes, where x is a variable, is given.

Theorem 1. *Given a variable x, a process p_i is x-relevant if, and only if, it belongs to $C(x)$ or it belongs to an x-hoop.*

Proof.
Necessity. If $p_i \in C(x)$, then it is obviously x-relevant. Consider now a process $p_i \notin C(x)$, but belonging to an x-hoop between two processes in $C(x)$, namely p_a and p_b. The history H, depicted Figure 3, includes an x-dependency chain along this hoop, from $w_a(x)v$ to $o_b(x)v$. In fact, we have $w_a(x)v \rightarrow_a w_a(x_1)v_1$, and, for each h, $1 \leq h \leq k \perp 1$, $r_h(x_h)v_h \rightarrow_h w_h(x_{h+1})v_{h+1}$, and $r_b(x_k)v_k \rightarrow_b o_b(x)$. If $o_b(x)$ is a read operation, the value that can be returned is constrained by the operation $w_a(x)v$, i.e., to ensure causal consistency, it cannot return neither $\perp$ nor any value written by a write operation belonging to the causal past of $w_a(x)v$. Similarly, if $o_b(x)$ is a write operation, namely $o_b(x) = w_b(x)v'$, the dependency $w_a(x)v \mapsto_{co} w_b(x)v'$ implies that, to ensure causal consistency, if a process $p_c \in C(x)$ reads both values v and v' then it reads them in such a order.

In both cases, information regarding the dependency $w_a(x)v \mapsto_{co} o_b(x)$ has to be propagated through intermediary processes $p_1, \ldots, p_{k-1}$ belonging to the x-hoop, in particular by p_i.

Sufficiency. The analysis above shows that the purpose of transmitting control information concerning the variable x is to ensure causal consistency. In particular, if an operation $o_1{=}w_a(x)v$ is performed by a process $p_a \in C(x)$, then any operation $o_2 = o_b(x)$ performed by another process $p_b \in C(x)$ is constrained by o_1 only if $o_1 \mapsto_{co} o_2$.

We have that $o_1 \mapsto_{co} o_2$ only if one of the two following cases holds:

1. A "direct" relation: $o_1 \mapsto_{ro} o_2$. In this case, no third part process is involved in the transmission of information concerning the occurrence of the operation o_1.

2. An "indirect" relation: there exists at least one o_h such that $o_1 \mapsto_{co} o_h$ and $o_h \mapsto_{co} o_2$. Such an indirect relation involve a sequence σ of processes $p_0 = p_a, \ldots, p_h, \ldots, p_k = p_b$ ($k \geq 1$) such that two consecutive processes p_{h-1} and p_h ($1 \leq h \leq k$) respectively perform operations o_{h-1} and o_h with $o_{h-1} \mapsto_{ro} o_h$. This implies that there exists a variable x_h such that $o_{h-1}{=}w_{h-1}(x_h)v_h$ and $o_h{=}r_h(x_h)v_h$. Consequently, x_h is shared by p_{h-1} and p_h, i.e., p_{h-1} and p_h are linked by an edge in the graph SG, meaning that the sequence σ is a path between p_a and p_b in the share graph SG. Such a path is either completely included in $C(x)$, or is a succession of x-hoops, and along each of them there is a x-dependency chain. Thus, a process $p_i \notin C(x)$ and not belonging to any x-hoop cannot be involved in these dependency chains. The result follows from the fact that this reasoning can be applied to any variable x, then to any pair of processes p_a and p_b in $C(x)$, and finally to any x-dependency chain along any x-hoop between p_a and p_b.

□

3.3 Impossibility of efficient partial replication

Shared memory is a powerful abstraction in large-scale systems spanning geographically distant sites; these environments are naturally appropriate for distributed applications supporting collaboration. Two fundamental requirements of large-scale systems are scalability and low-latency accesses:

i) to be scalable a system should accommodate large number of processes and should allow applications to manage a great deal of data;

ii) in order to ensure low latency in accessing shared data, copy of interested data are replicated at each site.

According to this, causal consistency criterion has been introduced by Ahamad et al. [2], [3] in order to avoid large latencies and high communication costs that arise in implementing traditional stronger consistency criteria, e.g., atomic [12] and sequential consistency [11]. "Many applications can easily be programmed with shared data that is causally consistent, and there are efficient and scalable implementations of causal memory" [2]. In particular, low latency is guaranteed by allowing processes to access local copy of shared data through wait-free operations. It means that causal consistency reduces the global synchronization between processes which is necessary to return consistent values.

This criterion is meaningful in systems in which complete replication is requested, i.e., when each process accesses all data in the system. On the other hand, considering large scale system with a huge and probably increasing number of processes and data, partial replication seems to be more reasonable: each process can directly access data it is interested in without introducing a heavy information flow in the network. From the results obtained in Section 3.2, several observations can be made, depending which is the *a priori* knowledge on variable distribution.

If a particular distribution of variables is assumed, it could be possible to build the share graph and analyze it off-line in order to enumerate, for each variable x not totally replicated, all the x-hoops. It results from Theorem 1 that only processes belonging to one of these x-hoops will be concerned by the variable x. Thus, an ad-hoc implementation of causal DSM can be optimally designed. However, even under this assumption on variable distribution, enumerating all the hoops can be very long because it amounts to enumerate a set of paths in a graph that can be very big if there are many processes.

In a more general setting, implementations of DSM cannot rely on a particular and static variable distribution, and, in that case, any process is likely to belong to any hoop. It results from Theorem 1 that each process in the system has to transmit control information regarding all the shared data, contradicting scalability.

Thus, causal consistency does not appear as the most appropriate consistency criterion for large-scale systems. For this reason, in the next sections we try to weaken the causal consistency in order to find a consistency criterion that allows efficient partial replication implementations of the shared memory, while being strong enough to solve interesting problems.

4 Weakening the causal consistency criterion

In the proof of Theorem 1, we point out that implementation constraints and information propagation in order to maintain consistency are due to dependency chains that can be created along the hoops.

In the next sections we investigate new order relations obtained by weakening the causality order relation such that, for any variable x, x-hoops cannot lead to the creation of x-dependency chains. The notion of dependency chain has been defined with respect to the particular order relation introduced. This definition holds for any relation defined on the sets O_H, just by replacing the $\mapsto_{co}$ relation by the appropriate relation, and Theorem 1 still holds in this new setting.

In the following, we respectively denote the *initial* and the *final* operation of a dependency chain as o_1 and o_2.

4.1 Lazy Causal Consistency

In this section we consider a weakened version of the traditional program order relation, based on the observation that some operations performed by a process could be permuted without effect on the output of the program (e.g., two successive read operations on two different variables). This partial order, named *Lazy Program Order* and denoted $\rightarrow_{li}$, is defined for each $ap_i \in \Pi$ as follows:

Definition 5 (Lazy program order). *Given two operations o_1 and o_2 in h_i, $o_1 \rightarrow_{li} o_2$ iff o_1 is invoked before o_2 and one of the following condition holds:*

- *o_1 is a read operation and o_2 is a read operation on the* same *variable or a write on* any *variable.*
- *o_1 is a write and o_2 is an operation on the* same *variable;*
- *$\exists\ o_3$ such that $o_1 \rightarrow_{li} o_3$ and $o_3 \rightarrow_{li} o_2$*

Given a history H, the lazy causality order $\mapsto_{lco}$, is a partial order that is the transitive closure of the union of the history's lazy program order and the read-from order. Formally:

Definition 6 (Lazy causal order). *Given two operations o_1 and o_2 in O_H, $o_1 \mapsto_{lco} o_2$ if and only if one of the following cases holds:*

- *$\exists\ ap_i$ s.t. $o_1 \mapsto_{li} o_2$ (lazy program order),*
- *$\exists\ ap_i, ap_j$ s.t. o_1 is invoked by ap_i, o_2 is invoked by ap_j and $o_1 \mapsto_{ro} o_2$ (read-from order),*
- *$\exists\ o_3 \in O_H$ s.t. $o_1 \mapsto_{lco} o_3$ and $o_3 \mapsto_{lco} o_2$ (transitive closure).*

If o_1 and o_2 are two operations belonging to O_H, we say that o_1 and o_2 are *concurrent* w.r.t. $\mapsto_{lco}$, denoted $o_1 \,||_{lco}\, o_2$, if and only if $\neg(o_1 \mapsto_{lco} o_2)$ and $\neg(o_2 \mapsto_{lco} o_1)$.

Definition 7 (Lazy Causally Consistent History). *A* history *H is lazy causal consistent if for each application process ap_i there is a serialization S_i of H_{i+w} that respects $\mapsto_{lco}$.*

A memory is lazy causal if it admits only lazy causally consistent histories.
Figure 4 depicts an history which is lazy causal but not causal.

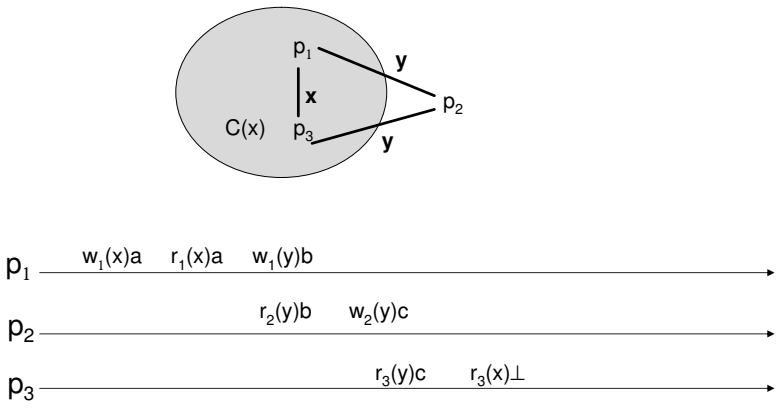


Figure 4: A lazy causal but not causal history

The corresponding serializations for the lazy causal are:
$S_1 = w_1(x)a,\ r_1(x)a,\ w_1(y)b,\ w_2(y)c$
$S_2 = w_1(x)a,\ w_1(y)b,\ r_2(y)b,\ w_2(y)c$
$S_3 = r_3(x)\perp,\ w_1(x)a,\ w_1(y)b,\ w_2(y)c,\ r_3(y)c$

In this history, no x-dependency chain is created along the x-hoop $[p_1, p_2, p_3]$. In fact, even though $w_1(x)a \mapsto_{lco} r_3(y)c$, we have, according to definition 7, $r_3(y)c||_{lco} r_3(x)\perp$ and thus $w_1(x)a \not\mapsto_{lco} r_3(x)\perp$. In particular, the value returned by the last read operation is consistent.

However, the situation is different if we consider the history depicted in Figure 5. In that case, an x-dependency chain along the x-hoop $[p_1, p_2, p_3]$ is created since, according to definition 7, $r_3(y)c \rightarrow_{l3} w_3(x)d$ and thus $w_1(x)a \mapsto_{lco} w_3(x)d$.

In particular, if process p_4 reads both values a and d, it has to read them in this order (it is not the case in the history depicted Figure 5, which is *not* lazy causal consistent). In

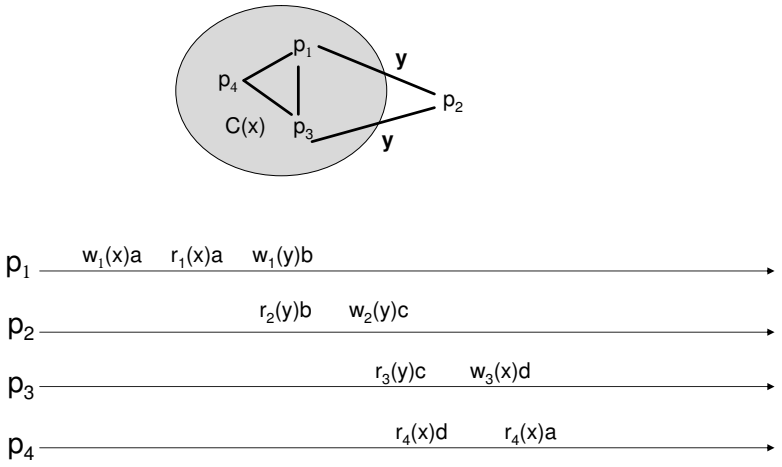


Figure 5: A not lazy causal history

particular, p_2 is x-relevant, although $p_2 \notin C(x)$. In this sense, the new order relation is still too strong to allow efficient partial replication.

Weakening further on the lazy program order, such that only operations on the same variable will be related, is not reasonable. In fact, even though this new relation would avoid the creation of dependency chains along hoops (and then would allow efficient implementation of DSM exploiting partial replication), it is too weak to solve interesting problems.

According to this, in the next section we consider the weakening of the traditional *read-from* relation that can exist between operations made by *different* processes.

4.2 Lazy Semi-Causal Consistency

Ahamad et al. [1] have introduced a weakened form of read-from relation, called *weak writes-before*. Their definition is based on a weakened program order, called *weak program order*, that is stronger than the *lazy program order* introduced in the previous section. In this section, we introduce the *lazy writes-before* relation, obtained from the weak writes-before by substituting lazy program order to weak program order. This relation, denoted $\rightarrow_{lwb}$, is formally defined as follows.

Definition 8 (Lazy write-before order). *Given two operations o_1 and o_2 in O_H, the* lazy writes-before *order relation, $\rightarrow_{lwb}$, on some history H is any relation with the following properties:*

- $o_1 = w_i(x)v$
- $o_2 = r_j(y)u$
- *exits an operation $o' = w_i(y)u$ such that $o_1 \rightarrow_{li} o'$*

Given a history H, we define the lazy semi-causality order relation, denoted $\mapsto_{lco}$, as the transitive closure of the union of the history's lazy program order and the lazy writes-before order relation.

Formally:

Definition 9 (Lazy semi-causal order). *Given two operations o_1 and o_2 in O_h, $o_1 \mapsto_{lsc} o_2$ if and only if one of the following cases holds:*

- *$o_1 \rightarrow_{li} o_2$ for some p_i;*
- *$o_1 \rightarrow_{lwb} o_2$*
- *$\exists\ o_3$ such that $o_1 \mapsto_{lsc} o_3$ and $o_3 \mapsto_{lsc} o_2$*

If o_1 and o_2 are two operations belonging to O_H, we say that o_1 and o_2 are *concurrent* w.r.t. $\mapsto_{lsc}$, denoted $o_1 \ ||_{lcc}\ o_2$, if and only if $\neg(o_1 \mapsto_{lsc} o_2)$ and $\neg(o_2 \mapsto_{lsc} o_1)$.

Definition 10 (Lazy Semi-Causally Consistent History). *A* history *H is lazy semi-causally consistent if for each application process ap_i there is a serialization S_i of H_{i+w} that respects $\mapsto_{lsc}$.*

A memory is Lazy Semi-Causal (LSC) iff it allows only lazy semi-causally consistent histories.

We show that this consistency criterion is still too strong for an efficient partial replication. In particular, we point out an history in which an x-dependency chain is created along an x-hoop. This dependency chain arises because of $\mapsto_{lwb}$ relation.

More precisely, in Figure 6 we have $w_1(x)a \mapsto_{lsc} w_3(x)d$. In fact, $w_1(x)a \mapsto_{lwb} r_2(y)b$ (because of $w_1(y)b$) and $w_2(y)e \mapsto_{lwb} r_2(z)c$ (because of $w_2(z)c$). Then, since $r_2(y)b \mapsto_{li} w_2(y)e$ and $r_3(z)c \mapsto_{li} w_3(x)d$, due to transitivity we have $w_1(x)a \mapsto_{lsc} w_3(x)d$.

In particular, if process p_4 reads both values a and d, it has to read them in this order (it is not the case in the history depicted Figure 6, which is *not* lazy semi-causal consistent). In particular, p_2 is x-relevant, although $p_2 \notin C(x)$. In this sense, the new order relation is still too strong to allow efficient partial replication.

It must be noticed that, since the semi-causality order relation, introduced by Ahamad et al. in [1], is stronger than the lazy-semi-causality introduced here, the semi-causality order relation does not allow efficient partial replication either.

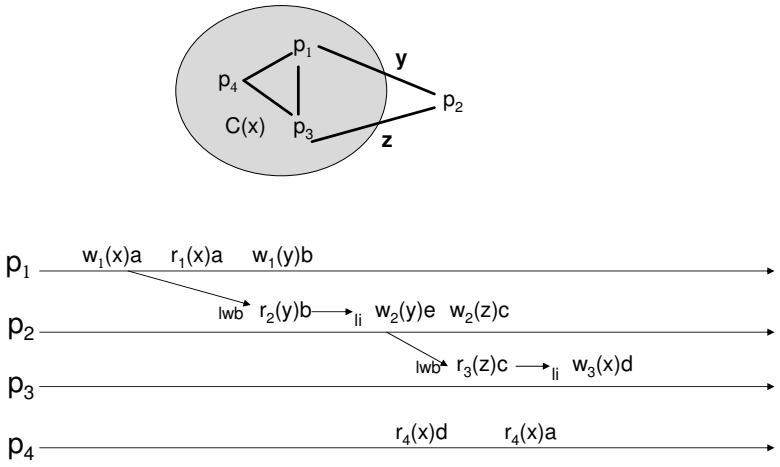


Figure 6: A not lazy semi-causally consistent history

Finally, the last possibility is to weaken the transitivity property such that two operations executed by different processes can be related only by the direct read-from relation. For what said, in the next section we consider a well-known consistency criterion, PRAM (PipelinedRAM)[13] and we prove that a PRAM memory can be efficiently implemented in a partial replicated environment.

5 PRAM

The PRAM consistency criterion [13] is weaker than the causal consistency criterion in the sense that it relaxes the transitivity due to intermediary processes [15]. In other words, it only requires that all processes observe the writes performed by a same process p_i in the same order (e.g., p_i program order), while they may disagree on the order of writes by different processes. The PRAM consistency is based on a relation, denoted $\mapsto_{pram}$, weaker than $\mapsto_{co}$. Formally [15][4]:

Definition 11 (PRAM relation). *Given two operations o_1 and o_2 in O_H, $o_1 \mapsto_{pram} o_2$ if, and only if, one of the following conditions holds:*

1. $\exists p_i : o_1 \mapsto_i o_2$ (program order), or

[4] in [15] this relation is denoted $\mapsto_{H'}$.

2. $\exists p_i\ \exists p_j\ i \neq j : o_1 = w_i(x)v$ *and* $o_2 = r_j(x)v$, *i.e.* $o_1 \mapsto_{ro} o_2$ *(read-from relation).*

Note that $\mapsto_{pram}$ is an acyclic relation, but is not a partial order due to the lack of transitivity.

Definition 12 (PRAM consistent history). *A history H is PRAM consistent if, for each application process ap_i, there exists a serialization H_{i+w} that respects $\mapsto_{pram}$.*

A memory is PRAM iff it allows only PRAM consistent histories.

The following result shows that PRAM memories allow efficient partial replication implementations.

Theorem 2. *In a PRAM consistent history, no dependency chain can be created along hoops.*

Proof. Let x be a variable and $[p_a, \ldots, p_b]$ be a x-hoop. A x-dependency chain along this hoop is created if H includes $w_a(x)v$, $o_b(x)$ and a pattern of operations, at least one for each process of the x-hoop, implying $w_a(x)v \mapsto_{pram} o_b(x)$. But the latter dependency can occur only if point 1 or point 2 of Definition 11 holds. Point 1 is excluded because $a \neq b$. Point 2 is possible only if $o_b(x) = r_b(x)v$ and the dependency $w_a(x)v \mapsto_{pram} r_b(x)v$ is $w_a(x)v \mapsto_{ro} r_b(x)v$, i.e., does not result from the operations performed by the intermediary processes of the hoop. □

As a consequence of this result, for each variable x, there is no x-relevant process out of $C(x)$, and thus, PRAM memories allow efficient partial replication implementations.

Although being weaker than causal memories, Lipton and Sandberg show in [13] that PRAM memories are strong enough to solve a large number of applications like FFT, matrix product, dynamic programming and more generally the class of oblivious computations[5]. In his PhD, Sinha [16] shows that totally asynchronous iterative methods to find fixed points can converge in Slow memories, which are still weaker than PRAM. In the next section we illustrate this power, together with the usefulness of partial replication, by showing how the Bellman-Ford shortest path algorithm can be solved by using PRAM memory.

6 Case study: Bellmann-Ford algorithm

A packet-switching network can be seen as a directed graph, $G(V, \Gamma)$, where each packet-switching node is a vertex in V and each communication link between node is a pair of parallel edges in Γ, each carrying data in one direction. In such a network, a routing decision is necessary to transmit a packet from a source node to a destination node traversing several links and packet switches. This can be modelled as the problem of finding a path through the graph. Analogously for an Internet or an intranet network. In general, all packet-switching networks and all internets base their routing decision on some form of least-cost criterion,

[5] "A computation is oblivious if its data motion and the operations it executes at a given step are independent of the actual values of data."[13]

i.e minimize the number of hope that correspond in graph theory to finding the minimum path distance. Most least-cost routing algorithms widespread are a variations of one of the two common algorithms, Dijkstra's algorithm and the Bellman-Ford algorithm[6].

6.1 A distributed implementation of the Bellman-Ford algorithm exploiting partial replication

In the following we propose a distributed implementation of the Bellman-Ford algorithm to compute the minimum path from a source node to every other nodes in a system, pointing out the usefulness of partial replication to efficiently distribute the computation. In the following we refer to nodes as processes.

The system (network) is composed by N processes $ap_1, \ldots, ap_N$ and it is modelled with a graph $G = (V, \Gamma)$, where V is the set of vertex, one for each process in the system and Γ is the set of edges (i, j) such that ap_i, ap_j belong to V and there exists a link between i and j.

Let us use the following notation:

- $\Gamma^{-1}(i) = \{j \in V | (i, j) \in \Gamma\}$ is the set of predecessors of process ap_i,
- s=source process,
- $w(i, j)$=link cost from process ap_i to process ap_j. In particular:
  i) $w(i, i) = 0$,
  ii) $w(i, j) = \infty$ if the two processes are not directly connected,
  iii) $w(i, j) \geq 0$ if the two processes are directly connected;
- x_i^k= cost of the least-cost path from source process s to process ap_i under the constraint of no more than k links traversed.

The centralized algorithm proceeds in steps. For each successive $k \geq 0$:

1. [**Initialization**]
   $x_i^0 = \infty$, $\forall\ i \neq s$
   $x_s^k = 0$, for all k
2. [**Update**] for each successive $k \geq 0$:
   $\forall\ i \neq s$, compute $x_i^{k+1} = \min_{j \in \Gamma^{-1}(i)} [x_j^k + w(j, i)]$

It is well-known that, if there are no negative cost cycles, the algorithm converge in at most N steps.

The algorithm is distributively implemented as follows. Without loss of generality, we assume that process ap_1 is the source node. We denote as x_i the current minimum value from node 1 to node i. Then, to compute all the minimum path from process ap_1 to every

other process in the system, processes cooperate reading and writing the following set of shared variables $\mathcal{X} = \{x_1,\ x_2, \ldots,\ x_N\}$. Moreover, since the algorithm is iterative, in order to ensure liveness we need to introduce synchronization points in the computation. In particular, we want to ensure that at the beginning of each iteration each process ap_i reads the new values written by his predecessors $\Gamma^{-1}(i)$. Thus each process knows that at most after N iterations, it has computed the shortest path. With this aim, we introduce the following set of shared variables $\mathcal{S} = \{k_1,\ k_2,\ \ldots,\ k_N\}$.

Each application process ap_i only access a subset of shared variables. More precisely, ap_i accesses $x_h \in \mathcal{X}$ and $k_h \in \mathcal{S}$, such that $h = i$ or ap_h is a predecessor of ap_i.

```
MINIMUM PATH
1  k_i := 0;
2  if (i == 1)
3     x_i := 0;
4     else   x_i := ∞;
5  while(k_i < N) do
6           while(⋀_{h∈Γ^{-1}(i)} (k_h < k_i)) do;
7           x_i := min([x_j + w(j,i)] ∀j ∈ Γ^{-1}(i));
8           k_i := k_i + 1
```

Figure 7: pseudocode executed by process ap_i

Since each variable x_i and k_i is written only by one process, namely ap_i, it is simple to notice that the algorithm in Figure 7, correctly runs on a PRAM shared memory. Moreover, since each process has to access only a subset of the shared memory, we can assume a partial replication implementation of such memory. In particular, at each node where the process ap_i is running to compute the shortest path, there is also a MCS process that ensure Pram consistency in the access to the shared variables.

The algorithm proposed is deadlock-free. In fact, given two processes ap_i and ap_j such that ap_i is a predecessor of ap_j and viceversa, the corresponding barrier conditions (line 6 of Figure 7) cannot be satisfied at the same time: $k_i < k_j$ and $k_j < k_i$.

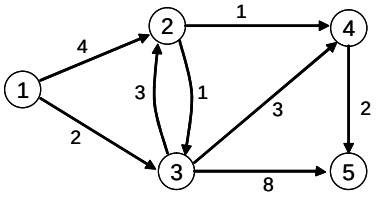


Figure 8: An example

As an example, let us consider the network depicted in Figure 8. We have the following set of processes $\Pi = \{ap_1,\ ap_2,\ ap_3,\ ap_4,\ ap_5\}$ and the corresponding variable distribution:

$\mathcal{X}_1 = \{x_1,\ k_1\}$,
$\mathcal{X}_2 = \{x_1,\ x_2,\ x_3,\ k_1,\ k_2,\ k_3\}$,
$\mathcal{X}_3 = \{x_1,\ x_2,\ x_3,\ k_1,\ k_2,\ k_3\}$,
$\mathcal{X}_4 = \{x_2,\ x_3,\ x_4,\ k_2,\ k_3,\ k_4\}$,
$\mathcal{X}_5 = \{x_3,\ x_4,\ x_5,\ k_3,\ k_4,\ k_5\}$.

In Figure 9 we show the pattern of operations generated by each process at the k-th step of iteration, we only explicit value returned by operations of interest. In reality, in order to point out the sufficiency of PRAM shared memory to ensure the safety and the liveness of the algorithm, we start the scenario showing the two last write operations made by each process at $(k \perp 1)$-th step. In this sense, it must be noticed that the protocol correctly runs if each process reads the values written by each of its neighbors according to their program order.

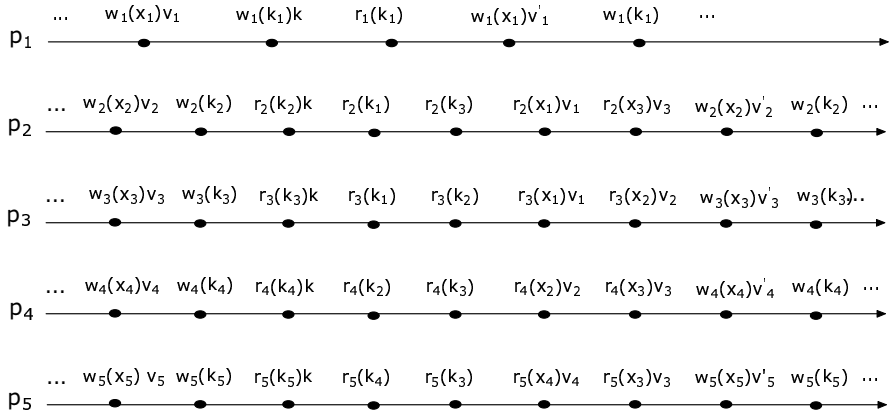


Figure 9: A step of the protocol in Figure 7 for the network in Figure 8

7 Conclusion

This paper has focused on the pertinence of implementing distributed shared memories by using partial replication of variables. It has introduced the notions of share graph and hoops to model the distribution of variables on the processes, and the notion of dependency chain along hoops to characterize processes that have to transmit information on variables that they don't manage. As a consequence, it has been shown that, in general, distributed shared memories enforcing consistency criteria stronger than lazy-semi-causality do not allow

efficient implementation based on partial replication. Among these consistency criteria are semi-causality, causality, sequentiality and atomicity, all previously known, and lazy-semi-causality, lazy-causality, introduced here. It has also shown that distributed shared memories enforcing consistency criteria weaker than PRAM are prone to efficient implementation based on partial replication. The power of PRAM memories has been illustrated with the particular example of Bellman-Ford shortest path algorithm.

This paper opens the way for future work. First, the design of an efficient implementation of PRAM memories based on partial replication. Second, on a more theoretical side, the "optimality" of the PRAM consistency criterion, with respect to efficient implementation based on partial replication. In other words, the existence of a consistency criterion stronger than PRAM, and allowing efficient partial replication implementation, remains open.

Acknowledgements

We like to thank Michel Raynal for suggesting this subject of research and for insightful discussions on this work.

References

[1] M. Ahamad, R.A. Bazzy, P. Kohli and G. Neiger. The Power of Processor Consistency. *ACM*, 1993.

[2] M.Ahamad, R. John, P. Kohli and G. Neiger. Causal Memory Meets the Consistency and Performance Needs of Distributed Application!.*EW 6:Proceedings of the 6th workshop on ACM SIGOPS European workshop*, 45-50, 1994.

[3] M. Ahamad, G. Neiger, J.E. Burns, P. Kohli and P.W. Hutto. Causal Memory: Definitions, Implementation and Programming. *Distributed Computing* 9(1): 37-49, 1995.

[4] M. Ahamad, M. Raynal and G. Thia-Kime. An adaptive architecture for causally consistent distributed services. *Distributed System Engineering*, 6: 63-70, 1999.

[5] H. Attiya and J. Welch. *Distributed Computing* (second edition), Wiley, 2004.

[6] R. E. Bellman. On a routing problem. *Quarterly Applied Mathematics*, XVI(1): 87-90, 1958.

[7] R. Baldoni, C. Spaziani, S. Tucci-Pergiovanni and D. Tulone. An implementationof causal memories using the writing semantics, *Proc. 6th Int. Conf. on Principles of Distributed Systems*, Hermes Press, 43-52, 2002.

[8] R. Baldoni, A. Milani, S. Tucci-Pergiovanni. Optimal propagation-based protocols implementing causal memories. *Distributed Computing*, to appear, 2005.

[9] E. Jimenez, A. Fernández, V. Cholvi: On the Interconnection of Causal Memory Systems. In: press in Journal of Parallel and Distributed Computing(2004).

[10] A.D. Kshemkalyani, M. Singhal. Necessary and sufficient conditions on the information for causal message ordering and thier optimal implementation. *Distributed Computing*, 11:91-111, 1988

[11] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. IEEE Transactions on Computers **28(9)**, 690–691(1979).

[12] L. Lamport. On Interprocess communication; part I: Basic formalism. *Distributed Computing*, 1(2):77-85, 1986.

[13] R. Lipton, J. Sandberg. PRAM: a Scalable Shared Memory. Technical Report **CS-TR-180-88**, Princeton University(1988).

[14] M. Raynal, M. Ahamad. Exploiting write semantics in implementing partially replicated causal objects, *Proc. 6th Euromicro Conference on Parallel and Distributed Systems*, 164-175, 1998.

[15] M. Raynal, A. Schiper. A suite of formal definitions for consistency criteria in distributed shared memories.*Proc. 9-th Int. IEEE Conference on Parallel and Distributed Computing Systems (PDCS96), Dijon, France*, pp. 125-131, 1996.

[16] H. S. Sinha. Mermera: non-coherent distributed shared memory for parallel computing. Technical Report **BU-CS-93-005**, Boston University, 1993.