An Implementation of Causal Memories using the Writing Semantic

R. Baldoni *, C. Spaziani and S. Tucci-Piergiovanni
Dipartimento di Informatica e Sistemistica
Universita' di Roma "La Sapienza"
Via Salaria 113, I-00198 Roma, Italy
baldoni@dis.uniroma1.it

D. Tulone
Bell-Laboratories
700 Mountain Avenue
Murray Hill, NJ 07974-0636, USA
tulone@research.bell-labs.com

Abstract

Causal consistency has been introduced to increase wait-free concurrent accesses to a shared memory. Contrarily to sequential consistency, causal consistency allows independent write operations, with respect to the causality relation, to be executed in different orders at each distinct process. This paper introduces a protocol for fully replicated settings which exploits the writing semantic and piggybacks on each update message related to a write operation an amount of control information which is $O(n)$ where n is the number of processes. The protocol tracks causality relation by taking transitive dependencies on write operations into account.

Keywords: Causal Memories, Writing Semantic, Replication, Distributed Coordination, Distributed Systems.

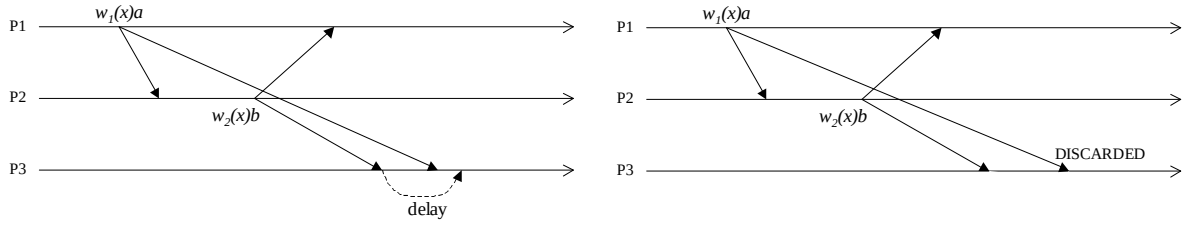
1 Introduction

Implementing a shared memory abstraction which preserves a given consistency criterion in an asynchronous distributed system is a real challenge due to unpredictable latencies and failures. Programmers are more comfortable to work with very strict consistency criteria like sequential consistency and atomic consistency. These criteria impose all processes implementing the shared memory abstraction agree on logical (sequential consistency [8]) or physical (atomic consistency [12]) execution order of a set of operations on the memory. Such consistency criteria are then costly to implement if processes are deployed in a partially synchronous distributed systems [5], or, even impossible (in deterministic way) in fully asynchronous distributed systems in the presence of failures due to FLP impossibility result [6].

*Corresponding Author: Fax n. + 39 06 85300849. Email: baldoni@dis.uniroma1.it.

This is the reason why weaker consistency critera such as PRAM (also called FIFO consistency) [10] and causal consistency [1] have been introduced in the past. These consistency criteria allow to get better scalability and performance with respect to more restrictive consistency criteria. In particular causal consistency ensures that values returned by a read operation are consistent with the Lamport causality relation [9]. In a setting where each process has a local copy of the shared memory locations, causal ordering ensures that readers can access their local copy concurrently (wait-free) without the risk of an invalidation of those reads due to a write operation. This enlarges the set of possible executions with respect to atomic and sequential consistency criteria.

To our knowledge, two protocols have been presented in the literature in the context of fully ([1]) and partially replicated memory locations ([14]). To ensure causal ordering of readings each message that notifies a write operation (update message) piggybacks a control information (i.e., actually a vector clock [11]) which is at least $O(n)$ where n is the number of processes (See section 3.5 for a deeper discussion). More specifically, [1] works as follows: when an update message of a write operation arrives at a process out of causal order (as $w_2(x)b$ at process p_3 depicted in Figure 1(a)), the execution of that operation is delayed till all update messages of causally preceding operations have been delivered ($w_1(x)a$ update message in figure 1(a)). To get more efficiency causal consistency protocols, Raynal and Ahamad [14] introduce the notion of writing semantic. This notion informally states that a process p_i can write a new value b in the memory location x, (e.g. $w(x)b$) as soon as it receives the $w(x)b$'s update message even though p_i detects, looking into the control information of $w(x)b$'s update message, there exists a causally preceding write operation on x, (e.g. $w(x)a$) not yet arrived at p_i. When the $w(x)a$'s update message arrives at p_i, it is discarded as it contains an obsolete value. This scenario is depicted in Figure 1(b). From an abstract point of view, *the causality relation is actually preserved* as it is like $w(x)a$ has arrived and executed first at p_i and then it has been immediately overwritten by $w(x)b$.



(a) (b)

Figure 1: Example of writing semantic

The protocol of Raynal and Ahamad [14], adapted to a fully replicated setting, piggybacks on each update message a control information which is the maximum between $O(n)$ and $O(m)$, where m is the number of replicated objects. In this paper we provide a protocol exploiting writing semantic which guarantees causal consistency of a shared memory system by piggybacking on each update message a control information which is $O(n)$. More specifically, the protocol piggybacks into an update message of a write operation $w(x)$ issued by p_i: (i) a vector of size n which represents to the p_i's knowledge the sequence numbers of the last write operation on

x executed by each process (used for detecting obsolete values) (ii) a set of write operations which represents the causal barrier of $w(x)$. i.e., $w(x)$ cannot be executed at a process till all the elements in its causal barrier have been executed . The idea of causal barrier was presented for the first time in [13] in the context of guaranteeing causal message deliveries within multimedia applications.

Compared to the protocol of Raynal and Ahamad [14], the one presented in this paper is well-suited to settings where the number of shared memory locations is more than n. Moreover, from a pedagogical point of view it clearly separates data structures for managing writing semantic and the ones for ensuring causal consistency.

The paper is structured into 4 sections. Section 2 presents the model. Section 3 presents the protocol and the relative data structures and Section 4 introduces some discussion points.

2 Shared Memory Model

We consider a finite set of sequential processes $\Pi \equiv \{p_1, p_2, \ldots p_n\}$ interacting via a shared memory $\mathcal{M}$ composed by m locations $x_1, x_2, \ldots x_m$. The locations can be accessed through *read* and *write* operations. A write operation executed by a process p_i, denoted $w_i(x)v$, stores a new value v in location x. A read operation executed by a process p_i, denoted $r_i(x)v$, returns to p_i the value v stored in location x.[1]

A *local history* of a process p_i, denoted h_i, is a set of read ad write operations. If p_i issues two operations o_1 and o_2 and o_1 is issued first, then o_1 *precedes* o_2 in *process order of* p_i. This precedence relation is denoted by $o_1 \mapsto_{po_i} o_2$.

Definition 1. *A* history*, denoted H, is a collection of local histories $\langle h_1, h_2, \ldots h_n \rangle$ one for each process.*

Operations issued by distinct processes in H are related by the *write-into* relation. Formally write-into relation, denoted $\mapsto_{wo}$, is defined as follow:

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

We denote as causal order relation ($o_1 \mapsto_{co} o_2$) the transitive closure of the union of the local history's relations $\mapsto_{po_i}$ and the relation $\mapsto_{wo}$. In other words $o_1 \mapsto_{co} o_2$ if and only if:

- $o_1 \mapsto_{po_i} o_2$ for some p_i;
- $o_1 \mapsto_{wo} o_2$;
- $\exists o_3 : o_1 \mapsto_{co} o_3$ and $o_3 \mapsto_{co} o_2$.

[1] Whenever not necessary we omit the value v and the identifier of the process that issued the operation

If o_1 and o_2 are two operations belonging to H, we said that o_1 and o_2 are *concurrent* w.r.t. $\mapsto_{co}$, denoted by $o_1||o_2$, if and only if $\neg(o_1 \mapsto_{co} o_2)$ and $\neg(o_2 \mapsto_{co} o_1)$.
Let as note that $\widehat{H} = (H, \mapsto_{co})$, is a partial order. We call such a partial order *execution history* of H.

Definition 2. *A read event belonging to H, denoted $r(x)v$, is legal if*

$$\exists w(x)v : w(x)v \mapsto_{co} r(x)v \wedge \nexists w(x)v' : w(x)v \mapsto_{co} w(x)v' \mapsto_{co} r(x)v$$

Definition 3. *An* execution history*, denoted $\widehat{H}$, is legal if all read operation in H are legal.*

Definition 4. *An* execution history *$\widehat{H} = (H, \rightarrow)$ is causally consistent iff*

1. *$\widehat{H}$ is legal;*
2. *$\mapsto_{co} \Rightarrow \rightarrow$;*

As processes are sequential the definition of causal memory allows to each process to see a specific linear extension of the partial order $\widehat{H}$. More specifically, this allows concurrent writes to be viewed in different orders by different processes.

Let us finally remark that if we replace point 1 of Definition 4 with:

1. there exists a linear extension of $\widehat{H}$ where all read events in H are legal;

we get another way to define sequential consistency [8, 15].

3 An Implementation of Causal Memory using Message Passing

This section presents an implementation of causal memory using message passing. It requires correct processes and reliable channels (a sent message is eventually received exactly once, no spurious messages can be ever delivered).

Each process p_i maintains a local copy M_i of the abstract shared memory $\mathcal{M}$ and issues read and write operations. Each time p_i issues a read operation, it actually reads immediately (i.e., without delay) the value accessing its own copy M_i. When p_i issues a write operation, it sends an update message to all the processes which are responsible to add the necessary synchronizations between update messages to respect causal consistency.

3.1 Causality Graph

The causality graph of a distributed execution $\widehat{H} = (H, \rightarrow)$ is a directed acyclic graph whose vertices are all *write* operation in H, such that there is a direct edge from $w(x)$ to $w(y)$ (denoted $w(x) \rightsquigarrow w(y)$) if

1. $w(x) \mapsto_{co} w(y)$ and
2. $\nexists w(z)$: $w(x) \mapsto_{co} w(z)$ and $w(z) \mapsto_{co} w(y)$

We say that $w(x)$ is an *immediate predecessor* of $w(y)$. It should be noted that as each process can issue a write operation at a time, a write operation v can have at most n immediate predecessors, one from each process.

In Figure 2 is shown an example of computation in which update messages, establishing causality relation among write operations, are depicted by thick line. Figure 3 shows the causality graph associated with the computation of Figure 2 .

Let $\overset{+}{\rightsquigarrow}$ denote the transitive closure of $\rightsquigarrow$. Two write operations $w(x)$ and $w(y)$ are said to be *concurrent* if neither of them is a predecessor of the other in the causality graph, *i.e.*, $\neg(w(x) \overset{+}{\rightsquigarrow} w(y))$ and $\neg(w(y) \overset{+}{\rightsquigarrow} w(x))$.

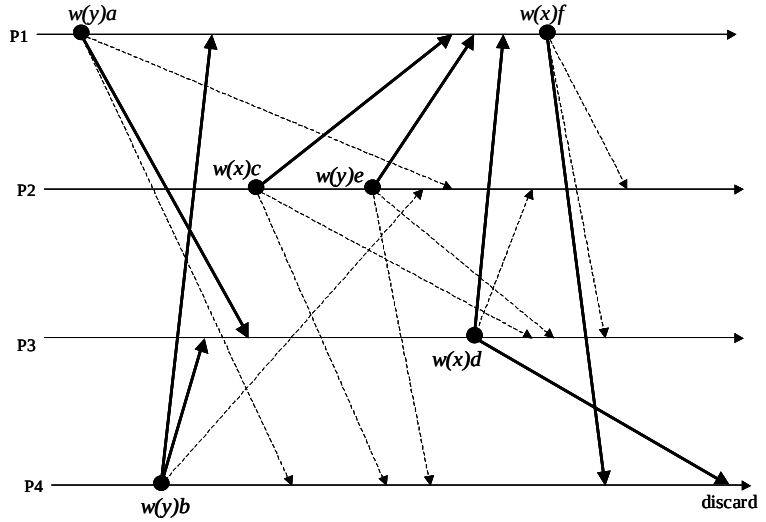


Figure 2: Example of message pattern

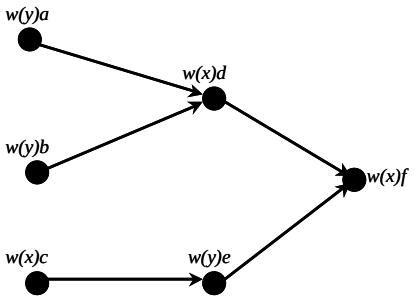


Figure 3: Causality graph of the message pattern of Figure 2

3.2 Writing semantic

To reduce the number of synchronizations among update messages of write operations, improving thus performance, Raynal and Ahamad observed in [14] that in a specific condition, based on writing semantic of a shared memory, a message of a write operation w can be early delivered at a process p_i even though a message of a write operation w' (such that $w' \mapsto_{co} w$) has not arrived at p_i yet (i.e., the execution of operation w is intentionally actually done out-of-causal order) provided that w' is not executed by p_i (i.e., the update message of w' is discarded by p_i). In this way, from the p_i's viewpoint it is like the operation w' has been virtually executed immediately (and atomically) before the execution of w (i.e., no read between the execution of o' and o). This guarantees causally consistent executions.

From a formal point of view the writing semantic (WS) condition, introduced by Raynal-Ahamad, can be expressed as follows using the relation $\rightsquigarrow$:

(WS)
if $(w(x)a \overset{+}{\rightsquigarrow} w(x)b)$ and $(\nexists w(z) \ : \ w(x)a \overset{+}{\rightsquigarrow} w(z)$ and $w(z) \overset{+}{\rightsquigarrow} w(x)\mathrm{b})$
then the value b can be applied at the location x of process p_i as soon as the relative update message is received (no need to wait for the message related to $w(x)a$).

In the following we refer $w(x)a$ as the *obsolete value* and $w(x)b$ as the *overwrite value*.

Let us consider the causality graph depicted in Figure 3: the write operation $w(x)f$ is an overwrite of the write operation $w(x)d$. This means that if a process p_4 (see Figure 2) receives $w(x)f$, it applies the overwrite value f immediately and then it will discard the $w(x)d$'s update message that carries the obsolete value d.

3.3 Data Structure

Each process p_i manages a local variable $\#seq$ (initially set to zero) in which it stores the sequence number of its last write operation. A write operation $w(x)a$ is uniquely identified by an *operation timestamp* t which is a triple $(sender\_identity, sequence\_number, variable)$. As an example $w(x)f$ issued by p_1 in Figure 2 is identified by the timestamp $(1, 2, x)$.

In the next subsections we present the data structures needed (i) to track causal dependencies among operations on the causality graph and (ii) to take into account the writing semantic.

3.3.1 Handling the Causality Graph: the IP set

Figure 3 shows that the execution of $w(x)f$ cannot occur before the execution of all the other write operations. But, due to transitive dependencies, the execution of $w(x)f$ can depend only on its direct predecessors $w(x)d$ and $w(y)e$ as $w(x)d$ can be only executed after $w(y)a$ and $w(y)b$, and, $w(y)e$ can be executed only after the execution of $w(x)c$. As remarked in [4], this simple observation allows us to reduce the control information

piggybacked on update messages with respect to solutions (e.g. [1]) that always piggyback a vector of integers of size n.

We denote as *immediate predecessor set* (IP_i) at process p_i the set of operation timestamps whose corresponding write operations would immediately precede a write operation issued at that time by p_i.

Each time a write operation w is *executed* at p_i, w is added into IP_i. When a process p_i *issues* a write operation with a timestamp ts then IP_i is set to ts.

3.3.2 Applying writing semantic: the DEL array

If a process p_i receives an update message of a write operation $w(x)a$ issued by p_j whose timestamp is say $(j, 25, x)$, then assume p_i executes $w(x)a$, all update messages related to write operations on variable x issued by p_j with sequence number less than 25 are actually overwritten. This information has to be transferred to the receiving processes when p_i issues a write operation in order that receivers can discard obsolete values (if any).

At this aim, each process manages an array DEL_i : *array* $[1 \dots n; 1, \dots m]$ *of Integers*. The component $DEL_i[j, x] = \#seq$ expresses the knowledge of p_i on the sequence number of the last write operation on variable x executed by p_j.

Each element of the array is initially set to zero, each time a write operation $w(x)a$ is issued by p_i, the column of DEL_i related x, denoted $DEL_i[*, x]$, is piggybacked to the update message.

When a process p_i receives an update message with a timestamp $(j, \#seq, x)$ piggybacking $DEL[*, x]$, after executing the corresponding write operation, it updates $DEL_i[*, x]$ by doing a component wise maximum between $DEL_i[*, x]$ and $DEL[*, x]$ and finally sets $DEL_i[j, x]$ to $\#seq$.

3.3.3 Guaranteeing Causal Consistency: the CB and the LCB set

A write operation $w(x)f$ should be executed at each process once all the immediate predecessors of $w(x)f$ have been executed but the ones that are made obsolete by $w(x)f$ *due to writing semantic* (e.g. $w(x)d$ in Figure 3). On the other hand, if we do not constraint the execution of $w(x)f$ to the one of $w(x)d$, $w(x)f$ has to be executed anyway after all the immediate predecessors of $w(x)d$ in order the execution be causally consistent (see Figure 3). All write operations that constraint the execution of a given write operation w form the *causal barrier* of w. To manage the causal barrier of a write operation, we introduce two data structures: CB and LCB.

Intuitively, the *causal barrier* of a write operation $w(x)$ is formed by a set of operation timestamps. This set includes (i) the timestamps of each operation $w(y)$ that immediately precedes $w(x)$ with $x \neq y$ and (ii) for each immediately predecessor $w'(x)$ of $w(x)$, all timestamps of the operations belonging to the causal barrier of $w'(x)$.

Let us note that from previous intuitive definition, the size of a causal barrier would have, as complexity, the number of write operations executed during the computation. However a simple observation on the program

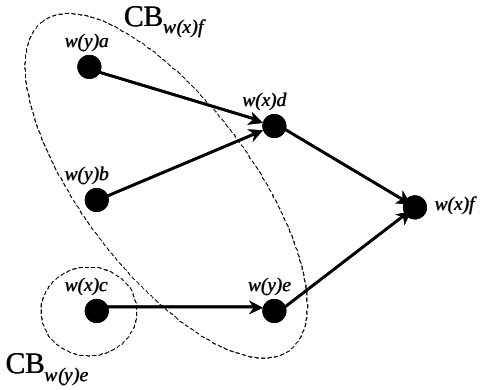


Figure 4: Example of causal barriers of the message pattern depicted in Figure 2

order allows to reduce the size of the causal barrier to a data structure of $O(n)$ through a filtering statement. Let us suppose two write operations $w(y)$ and $w(z)$ issued by the same process p_i whose timestamps, denoted $(i, \#seq_y, y)$ and $(i, \#seq_z, z)$ with $\#seq_y < \#seq_z$, belong to the causal barrier of $w(x)$. There is no point in keeping the timestamp of $w(y)$ in the causal barrier of $w(x)$ as the constraint on the the execution of $w(x)$ imposed by $w(z)$ transitively implies the one of $w(y)$ due to the program order on p_i. Following this observation, the causal barrier can be formally defined as follows:

Definition 5. *The* causal barrier *($CB_{w(x)}$) of a write operation $w(x)$ is a set of operation timestamps at most one for each process. This set includes the timestamp with the greatest sequence number one for each process taken from the set of timestamps formed by (i) the timestamps of each operation $w(y)$ that immediately precedes $w(x)$ with $x \neq y$ and (ii) for each immediately predecessor $w'(x)$ of $w(x)$, all timestamps of the operations belonging to $CB_{w'(x)}$.*

As an example Figure 4 shows the causal barrier of $w(x)f$. This data structure has therefore a complexity that only in the worst case is $O(n)$.

From an operational point of view, in order to guarantee causal consistency, each $w(x)v$'s update message sent by p_i has to piggyback as control information $CB_{w(x)v}$. In this way, when a destination process p_i receives an update message related to $w(x)v$, p_i will execute $w(x)v$ (i.e., the value v will be stored in $M_i[x]$) as soon as the following predicate (P) becomes true:

$$\forall (k, d, \ell) \in CB_{w(x)v} \ : \ (d \leq DEL_j[k, \ell]) \quad (P)$$

When a process p_i issues a $w(x)v$, to build $CB_{w(x)v}$ needs (i) IP_i and (ii) the causal barriers of all the

immediate predecessors in IP_i. To realize the second point p_i needs a data structure, denoted LCB_i, which memorizes such causal barriers. LCB is therefore a set of causal barriers. It is initialized to $\emptyset$, each time a write operation is executed at p_i the corresponding causal barrier is added to LCB_i and after p_i issues a write operation $w(x)v$, LCB_i is set to $CB_{w(x)v}$.

3.4 Protocol Behavior

When a process p_i wants to read a location x, p_i returns immediately, by using a single assignment statement, the value stored in the location x of local copy of $\mathcal{M}$. The read is legal due to the fact that the execution of a single statement is atomic and then the procedure will return the last value written into the location.

```
1  WRITE(x, v)
2    begin
3      #seq_i := #seq_i + 1; CB_(i,#seq,x) := ∅;
4      for each (k, d, ℓ) ∈ IP_i :: if (ℓ = x)
5                                    then CB_(i,#seq,x) := CB_(i,#seq,x) ⋃ CB_(k,d,ℓ) ∈ LCB_i;
6                                    else  CB_(i,#seq,x) := CB_(i,#seq,x) ⋃ (k, d, ℓ);
7      ∀p_k CB_(i,#seq,x) := ⋃_(k=1,...n)(max_#seq(CB_(i,#seq,x)|_pk));
8      send ["Write", x, v, #seq_i, DEL_i[∗, x], CB_(i,#seq,x)] to Π;
9      IP_i := {(i, #seq_i, x)}; LCB_i := {CB_(i,#seq,x)};
10   end
```

Figure 5: Write operation executed by a process p_i

When a process p_i wants to a write operation $w_i(x)v$, then it executes the procedure WRITE(x, v) depicted in Figure 5. In details, it first increments its sequence number and initializes the causal barrier of $w(x)v$ to $\emptyset$ (line 3). Then p_i builds $CB_{w(x)v}$ (lines 4-6) based on IP_i and LCB_i as defined in Section 3.3.3. Line 7 actually compresses $CB_{w(x)v}$ obtained exiting from the **for** statement by leaving, according to Definition 5, into $CB_{w(x)v}$ at most one timestamp for each distinct process. Then the update message of $w(x)v$ is broadcast to other processes piggybacking $CB_{w(x)v}$ (necessary for the causally ordered execution of $w(x)v$ at receiver side), $DEL[*, x]$ (to propagate knowledge on obsolete write operations on variable x) as well as x, $\#seq$ and the value v. Finally IP_i and LCB_i are updated according to their definitions (line 9).

Each time a "$Write$" message is received by a process p_i a new thread is spawned and the code of this thread is depicted in Figure 6. First at line 2 it is checked if the operation is obsolete (i.e., to the knowledge of p_i the operation $w(x)v$ has been overwritten). In the affirmative, the update message is discarded and the operation is not executed. Otherwise, the write operation wait until all operations in $CB_{(j,\#seq,x)}$ have been executed. Then the value v is stored into the location x (line 6) and then all data structures are updated according to their definition. Lines from 6 to 10 are executed atomically.

Figure 7 shows a message pattern with (i) the data structures piggybacked by update messages of $w(x)f$ and $w(x)d$ and (ii) the evolution of the DEL array at process p_4.

```
1  when (deliver ["Write", x, v, #seq, DEL[*, x], CB_(j,#seq,x)] from p_j) do
2    if ((#seq ≤ DEL_i[j, x])
3      then discard the "Write" message;
4           exit;
5    wait until (∀(k, d, ℓ) ∈ CB_(j,#seq,x) : d ≤ DEL_i[k, ℓ]);   % Synchronization to respect ↦_co %
6    M_i[x] := v;
7    LCB_i := LCB_i ⋃{CB_(j,#seq,x)};
8    IP_i = IP_i − CB_(j,#seq,x) ⋃{(j, #seq, x)};
9    DEL_i[j, x] := #seq;
10   ∀k ≠ j DEL_i[k, x] := max(DEL_i[k, x], DEL[k, x]);
```

Figure 6: Processing of an update message by a process p_i

3.5 Discussion

Correctness Proof. The proof showing safety and liveness requirement is omitted due to lack of space as it uses the typical induction techniques employed in [1, 2, 4]. However, it can be easily devised that:

- Liveness (i.e., no write operation remains blocked forever at a process) is ensured by the fact that either a write operation w is overwritten by another one, say w', at process p_i. In this case w is discarded at p_i; or the write operation w waits all the operations in its causal barrier. Due to the fact that messages cannot be lost and to the update rule of the data structure internal at each process, eventually w will be executed at p_i.
- Safety (i.e., the execution respects Definition 4) Point 2 of Definition 4 (i.e., write operations are causally ordered) is ensured by the wait condition in Figure 6 that guarantees write operations are causally ordered at each process (due to properties of transitive dependencies) and by the use of DEL array to ensure a safe discard of obsolete values. Finally Point 1 of Definition 4 is trivially satisfied (in a wait-free way) as each process reads a value from its local copy of the memory location. Therefore, each read operation returns the last value written onto that location.

Comparison to prior solutions. The only solution that exploits the writing semantic ([14]) was designed for a partially replicated setting. The simple adaptation of that protocol to a fully replicated setting would imply a control information, piggybacked to each update, composed by two vectors of size $O(n)$ and $O(m)$ respectively. On the other hand, the solution presented in this work piggybacks on each update message a control information composed by two vectors of size $O(n)$. More specifically, managing the writing semantic

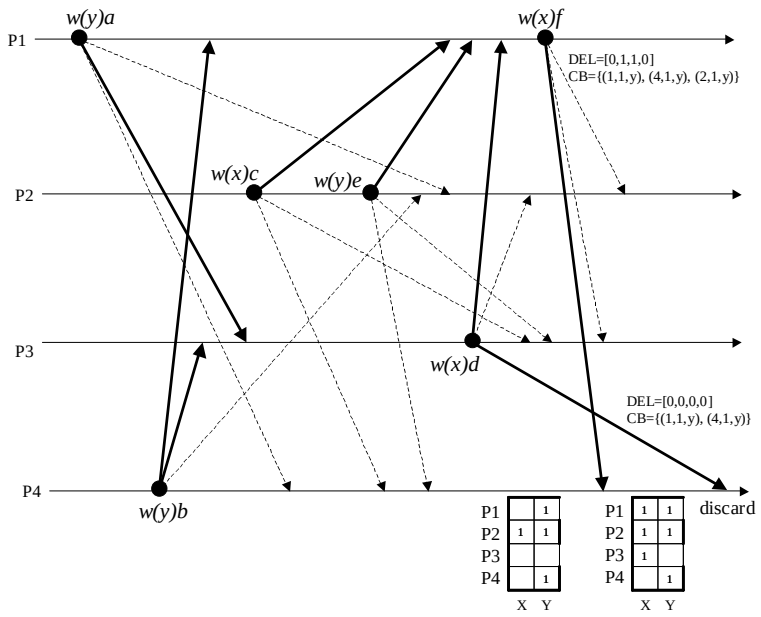


Figure 7: Example of the message pattern of Figure 2 with some data structures

requires locally each process to be equipped with an array $n \times m$ (i.e., $DEL[]$) and to attach to each message a column of this array. Guaranteeing that the execution of a write operation is causally consistent requires to attach a set of timestamps, the causal barrier, whose cardinality is at most n and each process has to be equipped locally with a set of causal barriers that actually stores the casual barriers of the immediate predecessors of a write operation issued by that process.

Then our solution is well-suited for environments in which the number of replicated objects (or memory locations) m is greater than the number of processes n.

Let us finally point out that if we do not consider the writing semantic, $CB_{w(x)v} = IP_i$. Therefore the predicate P of Section 3.3.3 becomes (i) the same to the one defined in [4] in the context of causal delivery of messages; and (ii) equivalent to the predicate defined by Ahamad et al. in [1] based on vector clocks. This equivalence can be easily proved as from the point of view of ensuring causality, a system of vector clock is equivalent to a causal barrier which tracks immediate predecessor of an operation [4].

4 Conclusion

In this paper we proposed a protocol to guarantee causal consistency in a shared memory implemented in a fully replicated setting. This protocol combines two interesting notions to get good performance (in terms for example of local buffer reduction): the writing semantic (defined in [14]) and the transitive dependency tracking

to guarantee causally consistent executions (defined in [13]). The protocol piggybacks on each update message a control information which is $O(n)$. As it provides a clear separation of concerns among the two basic notions, it can be used as a starting point for the development of other protocols in more complex settings such as client/server and overlapping groups.

References

[1] M. Ahamad, G. Neiger, J.E. Burns, P. Kohli, P.W. Hutto. Causal Memory: Definitions, Implementation and Programming, *Distributed Computing*, Vol. 9, No. 1, 1995, pp.37-49.

[2] R. Baldoni, A. Mostefaoui, and M.Raynal, Causal Delivery of Messages with Real-Time Data in Unreliable Networks, *Journal of Real-time Systems*, Vol. 10, No. 3, 1996.

[3] R. Baldoni, M. Raynal. Fundamentals of Distributed Computing: A Practical Tour of Vector Clocks, *IEEE Distributed Systems on-line*, Vol. 3, No. 2, 2002.

[4] R. Baldoni, R. Prakash, M. Raynal, and M. Singhal. Efficient Delta-Causal Broadcasting. *International Journal of Computer Systems Science and Engineering*, Pages 263-271, September 1998.

[5] C. Dwork and N.A. Lynch L. Stockmeyer, Consensus in the Presence of Partial Synchrony, *Journal of the ACM*, 1988, 35(2), 288–323.

[6] M. Fischer and N. Lynch and M. Patterson, Impossibility of Distributed Consensus with One Faulty Process, *Journal of the ACM*, 1985, 32(2), 374-382.

[7] M. Herlihy and J. Wing. Linearizability: a Correctness Condition for concurrent Objects, *ACM TOPLAS*, 12(3): 463-492, 1990

[8] L. Lamport, How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs, *IEEE Transactions on Computers*, Vol. C-28, No. 9, 1979, pp. 690-691.

[9] Lamport L., Time, Clocks and the Ordering of Events in a Distributed System, *Communications of the ACM*, 21(7):558-565, 1978.

[10] Lipton R. and Sandberg J, PRAM: A scalable shared memory, *Tech. Rep. CS-TR-180-88*, Princeton University, Sept. 1988.

[11] Mattern F., Virtual Time and Global States of Distributed Systems, *Proc. "Parallel and Distributed Algorithms" Conference*, (Cosnard, Quinton, Raynal, Robert Eds), North-Holland, pp. 215-226, 1988.

[12] J. Misra, Axioms for memory access in asynchronous hardware systems, *ACM Transactions on Programming Languages and Systems*, Vol.8, No. 1, 1986, pp.142-153.

[13] R. Prakash, M. Raynal, M. Singhal, An Efficient Causal Ordering Algorithm for Mobile Computing Environments, *Proc. 16th IEEE International Conference on Distributed Computing Systems* (1996) pp.744-751

[14] M. Raynal, M. Ahamad, Exploiting Write Semantics in Implementig Partially Replicated Causal Objects, *Proc. of 6th Euromicro Conference on Parallel and Distributed Systems*, pp.175-164, Madrid, January 1998.

[15] M. Raynal, A. Schiper, From Causal Consistency to Sequential Consistency in Shared Memory Systems, *Proc. 15th Int. Conf. on Foundations of Software Technology & Theoretical Computer Science*, Bangalore, India, Springer-Verlag LNCS 1026 (P.S. Thiagarajan Ed.), 1995, pp. 180-194.