Optimal Propagation based Protocols implementing Causal Memories

Roberto Baldoni, Alessia Milani and Sara Tucci-Piergiovanni
Dipartimento di Informatica e Sistemistica
Universita' di Roma "La Sapienza"
Via Salaria 113, I-00198 Roma, Italy
{baldoni,milani,tucci}@dis.uniroma1.it

Abstract

Ensuring causal consistency in a Distributed Shared Memory (DSM) means all operations executed at each process will be compliant to a causality order relation. This paper first introduces an optimality criterion for a protocol P, based on a complete replication of memory locations at each process and propagation of write updates, that enforces causal consistency. This criterion measures the capability of a protocol to update the local copy *as soon as* possible while respecting causal consistency. Then we present an optimal protocol built on top of a *reliable broadcast communication primitive* and we show how previous protocols based on complete replication presented in the literature are not optimal. Interestingly, we prove that the optimal protocol embeds a system of vector clocks which captures the read/write semantics of a causal memory. From an operational point of view, an optimal protocol exploiting reliable broadcast strongly reduces its message buffer overhead. Simulation studies show that the optimal protocol roughly buffers a number of messages of one order of magnitude lower than non-optimal ones based on the same communication primitive.

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. Consistency conditions define the rules of read/write accesses in the shared memory model. For instance, sequential consistency [19] requires that application processes agree on a common order for all read/write operations; atomic consistency [20] (also called linearizability [14]) requires that this order also respects the real-time. Finally, PRAM requires each application process agrees on the order of write operations issued by a process [21]. This allows two write operations invoked by two different application processes to be perceived in different order by a third one. A stronger consistency criterion than PRAM and weaker than

sequential is *causal consistency* introduced by Ahamad et al. in [1]. Causal consistency allows two or more non-causally related write operations to appear in different order to different application processes. Two operations o_1 and o_2 are causally related by the causality order relation, denoted $o_1 \mapsto_{co} o_2$, if and only if one of the following conditions is true: (i) o_1 and o_2 are issued by the same application process and o_1 precedes o_2 (*program order relation*), (ii) o_1 is a write operation $w(x)v$ and o_2 is a read operation on x which returns the value written by o_1 (*read-from order relation*) or (iii) there exists an operation o such that $o_1 \mapsto_{co} o$ and $o \mapsto_{co} o_2$. The interest of causality relation lies in the fact that it allows wait-free read/write operations [1].

The distributed shared memory abstraction has been 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 [3]. The implementation of MCS enforces a given consistency criterion. To improve performance, an MCS enforcing causal consistency has been usually implemented by protocols based on a complete replication of memory locations at each MCS process and propagation of the memory location updates [17]. In these protocols, namely *Complete Replication and Propagation* (CRP) based protocols, a read operations immediately returns (to the application process that invoked it) the value stored in the local copy. A write operation returns after (i) the updating of the local copy and (ii) an update message carrying the new value is sent to all MCS processes, exploiting communication primitives provided by the message passing system. Due to the concurrent execution of processes and to the fact that the underlying network can reorder messages, a CRP protocol is in charge to properly order incoming update messages at each process. This reorder is implemented through the suspension/reactivation of process threads which are in charge of executing the local update. If an update message m arrives at a process p and its immediate application violates causal consistency, the update thread is suspended by the CRP protocol, i.e. its application is delayed. This implies buffering of m at p. The thread is reactivated by the CRP protocol when the update can be applied without the risk of violation of causal consistency. Informally, a CRP protocol is *optimal* if each update is applied at a MCS process *as soon as* the causal consistency criterion allows it. In other words, no update thread is kept suspended for a period of time more than necessary.

In this paper, firstly, we formally define such optimality criterion for CRP protocols. This criterion actually defines a predicate, namely the activation predicate, on the distributed computation generated by the CRP protocol that becomes trues as soon as an update thread can be reactivated and thus the local update can be executed in a causally consistent way at a process. Secondly, an *optimal* CRP protocol is presented. Theoretically, an optimal CRP protocol actually exploits all the concurrency admitted by the causal consistency criterion. Third we show that when CRP protocols rely on top of a *reliable broadcast communication primitive*, an optimal protocol exhibits a strong reduction on the message buffer overhead at the MCS level with respect to a non-optimal one.

More precisely, after introducing the consistency memory model in Section 2, the paper presents, in Section 3, the definition of optimal CRP protocol. This passes through a precise definition of the implementation setting along with the relation between operations executed at application level and the corresponding distributed computation produced by the CRP protocol at MCS level. In the same section

we show that the most representative CRP protocol introduced by Ahamad et al in [1] (hereafter denoted as ANBKH) is *non-optimal*.

In Section 4, the paper presents an optimal CRP protocol $OptP$ that relies on a reliable broadcast primitive. Interestingly, $OptP$ adopts a vector as main data structure embedding actually the read/write operation semantics of a causal memory. The paper formally shows that this vector, namely $Write_{co}$, is actually a system of vector clocks characterizing $\mapsto_{co}$. This proof and the transitivity property of $\mapsto_{co}$ give us powerful optimization tools, borrowed from efficient representation of Fidge-Mattern vector clocks [11, 22], to derive an efficient implementation of $OptP$, namely $OptP_{ef}$ (Section 5). More specifically $OptP_{ef}$ embeds the Singhal-Kshemkalyani ([27]) and the direct dependency tracking ([15, 24]) techniques. Both techniques aim to reduce the number of entries of a system of vector clocks to be piggybacked onto protocol messages by following two orthogonal criterion.

In Section 5, we finally compare $OptP$ and $ANBKH$, both protocol based on a reliable broadcast primitive, in terms of message buffer overhead at MCS level. Simulation results clearly show that optimality has a strong impact on message buffer overhead. More specifically, $OptP$ outperforms $ANBKH$ by allowing one order of magnitude buffer space saving.

2 Shared Memory Model

This model is based on the one proposed by Ahamad et al. in [1]. We consider a finite set of sequential *application* processes $\{ap_1, ap_2, \ldots ap_n\}$ interacting via a shared memory $\mathcal{M}$ composed by m memory locations $x_1, x_2, ...x_m$. The memory 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 the location 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 the location x_h [1]. Each memory location 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. 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 is denoted by $o_1 \mapsto_{po_i} o_2$. The *history* $H=\langle h_1, h_2, \ldots h_n\rangle$, i.e. H is the collection of local histories (one for each application process). Operations done by distinct application processes are related by the *read-from order* relation. A read-from order relation,$\mapsto_{ro}$, on H is any relation with the following properties [1][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 o_1 such that $o_1 \mapsto_{ro} o_2$;
- if $o_2 = r(x)v$ for some x and there is no 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.

[1] Whenever not necessary we omit either the value v or the value and the variable or the value, the process identifier and the variable. 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 writes-into defined in [1]

A causality order $\mapsto_{co}$, [1], is a partial order that is the transitive closure of the union of the history's program order and the read-from order. Formally, $o_1 \mapsto_{co} o_2$ if and only if one of the following cases holds:

- $\exists\ ap_i$ s.t. $o_1 \mapsto_{po_i} o_2$ (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),
- $\exists\ o_3 \in H$ 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 H, we said 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 $\neg(o_2 \mapsto_{co} o_1)$. Let us finally define the *causal past of an operation o in a* history H *with respect to* $\mapsto_{co}$, denoted $\downarrow(o, \mapsto_{co})$, as follows:

$$\downarrow(o, \mapsto_{co}) = \{o' \in H | o' \mapsto_{co} o\}$$

Properties of a history

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

A serialization S respects a given order if, for any operation o_1 and o_2 in S, o_1 precedes o_2 in the 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 in H

Definition 2 (Causally Consistent History [1]). *An* 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.

Example 1. Let us consider a system composed by three application processes. The following history H_1 is causal:

h_1: $w_1(x_1)a; w_1(x_1)c$
h_2: $r_2(x_1)a; w_2(x_2)b$
h_3: $r_3(x_2)b; w_3(x_2)d$

Note that $w_1(x_1)a \mapsto_{co} w_2(x_2)b$, $w_1(x_1)a \mapsto_{co} w_1(x_1)c$ and $w_2(x_2)b \mapsto_{co} w_3(x_2)d$ while $w_1(x_1)c ||_{co} w_2(x_2)b$, $w_1(x_1)c ||_{co} w_3(x_2)d$.

3 Distributed Memory Consistency System

The shared memory abstraction is implemented by a *memory consistency system*(MCS) on top of a message passing system [3]. We assume a system consisting of a collection of nodes. On each node i

there is an application process ap_i and a MCS process p_i [3]. An application process ap_i invokes an operation to its local MCS process p_i which is in charge of the actual execution of the operation. The execution of an operation at a MCS process p_i generates events. Hereafter, for brevity, sometimes *we drop the acronym MCS and we refer to a MCS process as simply a process.* The distributed system is asynchronous. Message transfer delay is unpredictable but finite and there is no bound to the relative process speeds, however, the time taken by a MCS process to execute a computational step is finite.

3.1 Distributed Computation at MCS

The partial order induced on the history H corresponds to a sequence of events E_i produced at each MCS process p_i by a protocol P implementing the MCS level and ordered by the relation $<_i$. $e <_i e'$ means both e and e' occurred at p_i and e has occurred first. We also denote as $E_i|_e$ the prefix of E_i until e (not included). The set of all events produced by all MCS processes is denoted as $E = \bigcup_{i=1}^{n} E_i$.

The events of E are also ordered by Lamport's "happened before" relation [18], denoted $\rightarrow$, defined as follows: let e and e' be two events of E, $e \rightarrow e'$ iff (i) $e <_i e'$ or (ii) e is the sending of a message m and e' is the receipt of m or (iii) there exists e'' such that $e \rightarrow e''$ and $e'' \rightarrow e'$.

Let e and e' be two events belonging to E, e and e' are *concurrent* w.r.t. $\rightarrow$, denoted by $e \mid\mid e'$, if and only if $\neg(e \rightarrow e')$ and $\neg(e' \rightarrow e)$. The partial order induced by $\rightarrow$ on E is the distributed computation $\widehat{E} = \{E, \rightarrow\}$. The set of messages sent in a distributed computation $\widehat{E}$ is denoted as $M_{\widehat{E}}$.

3.2 Complete Replication and Propagation based Protocols

We assume each MCS process p_i endows a copy of the shared variables $x_1^i, x_2^i, \ldots x_h^i, \ldots x_m^i$. We assume p_i exchanges messages through a reliable broadcast primitive [13]. To send a broadcast message a MCS process invokes the `RELcast(`m`)` primitive while the underlying layer of a MCS process invokes the `RELrcv(`m`)` primitive which is an upcall used to receive m to the MCS process.

Runs of the complete replication and propagation based (CRP) protocols generate the following list of events at a process p_i:

- *Message send event.* The execution of `RELcast(`m`)` primitive at a process p_i generates the event $send_i(m)$.
- *Message receipt event.* $receipt_i(m)$ corresponds to the receipt of a message m by p_i through the execution of the `RELrcv(`m`)` primitive.
- *Apply event.* The event $apply_i(w_j(x_h)v)$ corresponds to the application of the value written by the write operation $w_j(x_h)v$ to the local copy, i.e., v is stored into x_h^i at p_i.
- *Return event.* $return_i(x_h, v)$ corresponds to the return of the value stored in p_i's local copy x_h^i.

Therefore, *apply events* and *return events* are internal events while the others involve communication. From the point of view of the mapping between operations and events, a CRP protocol communicating via reliable broadcast is characterized by the following pattern:

- *Each time a MCS process p_i executes a read operation $r_i(x)v$, p_i* eventually produces an event $return_i(x, v)$.

- *Each time a MCS process p_i executes a write operation $w_i(x_h)v$,* an update corresponding to $w_i(x_h)v$, denoted as $m_{w_i(x_h)v}$, is dispatched to all other MCS processes through a `RELcast(`$m_{w_i(x_h)v}$`)`, i.e. $send_i(m_{w_i(x_h)v})$ is produced.

- *Each time a MCS process p_i receives from the underlying network an update sent during the execution of a write operation $w_j(x_h)v$, p_i* produces an event $receipt_i(m_{w_j(x_h)v})$ and a new thread is spawned to handle the local application of the update (i.e., the occurrence of the event $apply_i(w_j(x_h)v)$. In this thread, p_i, firstly, tests a local activation" predicate [3], denoted $A(m_{w_j(x_h)v}, e)$ (initially set to *false*), to check if the update $m_{w_j(x_h)v}$ *is ready to be locally applied at p_i or not, just after the occurrence of the event e.* If $A(m_{w_j(x_h)v}, receipt_i(m_{w_j(x_h)v})$ is *true*, then the $apply_i(w_j(x_h)v)$ event can be scheduled by the local operating system underlying p_i. Note that when an activation predicate flips to true it will last true forever (stable property). If $A(m_{w_j(x_h)v}, receipt_i(m_{w_j(x_h)v})$ is *false* then the local update of x_h at p_i is delayed (actually the thread is suspended). A suspended thread handling $m_{w_j(x_h)v}$ is activated just after the occurrence of the first event e such that the predicate $A(m_{w_j(x_h)v}, e)$ flips to *true* and then the apply event is ready to be scheduled.

  This behavior can be abstracted through a wait statement, i.e., **wait until (**$A(m_{w_j(x_h)v}, e)$**)**. If a thread is suspended at p_i, it will spinning on the local activation predicate $A(m_{w_j(x_h)v}, e)$ till it will become true. We assume that the scheduler of the operating system is *fair*, i.e. it never consequently schedules the same type of event an infinite number of times.

Two CRP protocols using a reliable broadcast differ each other on the definition of the local activation predicate used to control threads handling the receipt of messages at a process. Thus, in the following we denote as $\mathcal{P} = \{P, P', ...\}$ all CRP protocols following the above pattern in which each one may have its own predicate A_P.

Clearly, an activation predicate of a protocol is required to activate threads in order to maintain causal consistency or *safety*. However, as will see, an activation predicate may be stronger than necessary to ensure causal consistency. It can actually suspend a thread for a time longer than necessary. In this case we say that the protocol is not optimal. In the following the notions of safety and optimality are formally stated.

3.2.1 Capturing causality order in the distributed computation

The causality order relations among operations invoked at application level have to be preserved at MCS level to assure safety. Actually, a causality order relation has to be mapped into a relation among events of a distributed computation. This mapping depends on the MCS protocol. In our case, each protocol has a local reads/writes and only during a write an update message is sent. From these features follows that the relations among operations reduce on relations among update messages.

[3] We assume that each process runs the same code, i.e. the activation predicate local at each process is the same for everyone.

For this reason, we define a relation denoted as $\overset{co}{\rightarrow}$ on the update messages sent during a distributed computation of a protocol $P \in \mathcal{P}$. The objective of $\overset{co}{\rightarrow}$ is capturing the causality order relations among *write* operations invoked by application processes into relations among update messages sent by MCS processes. Formally, the definition of $\overset{co}{\rightarrow}$ is the following:

Definition 3. *$m_{w(x)a} \overset{co}{\rightarrow} m_{w(y)b}$ iff one of the following conditions holds:*

1. $send_k(m_{w(x)a}) <_k send_k(m_{w(y)b})$
2. $send_k(m_{w(x)a}), send_j(m_{w(y)b}) : j \neq k, return_k(x,a) <_k send_k(m_{w(y)b})$
3. $\exists m_{w(z)c} : m_{w(x)a} \overset{co}{\rightarrow} m_{w(z)c} \overset{co}{\rightarrow} m_{w(y)b}$

Relation between $\overset{co}{\rightarrow}$ and $\rightarrow$. The relation $\overset{co}{\rightarrow}$ is actually a refinement of $\rightarrow$, i.e. given a run of a CRP protocol, $\overset{co}{\rightarrow}$ relates a set of messages contained in the set of messages related by the happened before. For instance, if two messages $m_{w(x)a}, m_{w(y)b}$ are $m_{w(x)a} \overset{co}{\rightarrow} m_{w(y)b}$ then they are also related by the happened before, i.e. $send_k(m_{w(x)a}) \rightarrow send_j(m_{w(y)b})$. The viceversa is not true. If $send_k(m_{w(x)a}) \rightarrow send_j(m_{w(y)b})$ and $k \neq j$ but no return events occurs in the run, then $send_k(m_{w(x)a}) \overset{co}{\not\rightarrow} send_j(m_{w(y)b})$.

Relation between $\overset{co}{\rightarrow}$ and $\mapsto_{co}$ The aim of $\overset{co}{\rightarrow}$ is *exactly* capturing the causality order relations between write operations in relations between update messages sent during the underlying distributed computation. Formally,the following property holds:

Property 1. $m_{w(x)a} \overset{co}{\rightarrow} m_{w(y)b} \Leftrightarrow \ w(x)a \mapsto_{co} w(y)b$

Proof. $m_{w(x)a} \overset{co}{\rightarrow} m_{w(y)b} \Rightarrow w(x)a \mapsto_{co} w(y)b$.
$m_{w(x)a} \overset{co}{\rightarrow} m_{w(y)b}$ means that one of the following condition holds:

1. $send_k(m_{w(x)a}) <_k send_k(m_{w(y)b})$
2. $send_j(m_{w(x)a}) : j \neq k, return_k(x,a) <_k send_k(m_{w(y)b})$
3. $\exists m_{w(z)c} : m_{w(x)a} \overset{co}{\rightarrow} m_{w(z)c} \overset{co}{\rightarrow} m_{w(y)b}$

From each protocol $P \in \mathcal{P}$: (i) for each MCS process p_k that sends an update message $m_{w(x)}$, the application process ap_k executes $w(x)$ (and viceversa) and (ii) for each MCS process p_h that returns a value a from the local copy x_h, the application process ap_h executes a read $r(x)a$ (and viceversa).
First condition. The application process ap_k has executed both $w(x)a$ and $w(y)b$ s.t. $w(x)a \mapsto_{po_k} w(y)b$.
Second condition. In this case an application process ap_k has executed $r(x)a$ and $w(y)b$, then $w(x)a \mapsto_{ro} r(x)a$ and $r(x)a \mapsto_{po_k} w(y)b$, i.e. $w(x)a \mapsto_{co} w(y)b$.
Third condition. This is the transitive closure of $\overset{co}{\rightarrow}$. Then $w(x)a \mapsto_{co} w(y)b$.

$w(x)a \mapsto_{co} w(y)b \Rightarrow m_{w(x)a} \overset{co}{\rightarrow} m_{w(y)b}$

When program order holds for writes invoked by an application process ap_k, the first condition of $\overset{co}{\rightarrow}$ holds for the corresponding update messages sent by the MCS process p_k. When read-from order holds, a MCS process p_k has returned the value a, i.e. p_k has already received and applied the update message

$m_{w(x)a}$ previously sent by another MCS process. From the second condition of $\xrightarrow{co}$ all messages sent by p_k after $return_k(x, a)$ are related by $\xrightarrow{co}$ (then even $m_{w(y)b}$). When the transitive closure holds, the third condition of $\xrightarrow{co}$ holds, as well. Then the claim follows. □

3.2.2 Safety

Let $P \in \mathcal{P}$. P is *safe* w.r.t. $\mapsto_{co}$ if and only if the order on local update applications at each MCS process is compliant with the order induced by $\xrightarrow{co}$. Formally:

Definition 4 (Safety). *Let $\widehat{E} = \{E, \rightarrow\}$, a distributed computation generated by $P \in \mathcal{P}$. P is safe iff:*

$$\forall\ m_w, m_{w'} \in M_{\widehat{E}} : (m_w \xrightarrow{co} m_{w'} \Rightarrow \forall i \in \{1 \dots n\}, apply_i(w) <_i apply_i(w'))$$

Any protocol P maintains safety through its activation predicate. An activation predicate of a safe protocol has to stop the immediate application of any update message m_w arrived out-of-order w.r.t. $\xrightarrow{co}$. Then it may allow the application of the delayed update only after all m_w's preceding updates are applied. It means that a protocol P is safe if its activation predicate $A_P(m_w, e)$ is *true* at a process p_i, if each update message m'_w such that $m_{w'} \xrightarrow{co} m_w$ has been already applied at p_i. However, any predicate of a safe protocol may provide that even something other event has to occur before the m_w's application. In this case the protocol is not optimal as the application is delayed even when it is not necessary, i.e. $A_P(m_w, e)$ is *false at least* until there exists a message a process p_i, m'_w such that $m_{w'} \xrightarrow{co} m_w$ and $m_{w'}$ has not yet been applied at p_i.

3.2.3 Optimality

Informally, a protocol P is optimal if its activation predicate $A_P(m_w, e)$ is *false* at a process p_i, *at most* until there exists an update message m'_w such that $m_{w'} \xrightarrow{co} m_w$ and $m_{w'}$ has not yet been applied at p_i. Note that the optimality does not imply safety. An optimal protocol may apply updates in arrival order regardless the order imposed by $\xrightarrow{co}$, however if it delays the application of an update m_w it does that for a "good reason", as the message is out of order with respect to $\xrightarrow{co}$.

An optimal protocol is formally defined as follows:

Definition 5 (Optimal Protocol). *P is optimal iff for any receipt of an update message m_w at p_i belonging to any $\widehat{E} = \{E, \rightarrow\}$, generated by a protocol $P \in \mathcal{P}$,*
$\forall e \in E : receipt_i(m_w) <_i e, \quad \neg A_P(m_w, e) \Rightarrow \neg A_{Opt}(m_w, e)$
where

$$A_{Opt}(m_w, e) \equiv \nexists\ m_{w'} \in M_{\widehat{E}} : \quad (\ m_{w'} \xrightarrow{co} m_w \ and\ \wedge apply_i(w') \notin E_i|_e\ )$$

Then, from Property 1, each process running an optimal protocol, *delays* the application of an update message m_w *until* there exists at least another update message carrying a write *not yet applied* that causally *precedes* w.

Relation between $A_{Opt}(m_w, e)$ and Safety. From the definition of $A_{Opt}(m_w, e)$ follows that each protocol P equipped with the activation predicate $A_{Opt}(m_w, e)$ is safe. In particular the activation predicate returns *true* at a process p_i, only after all updates preceding m_w have been also applied. Then we can also say that for each safe (even not optimal) protocol P, $A_P \Rightarrow A_{Opt}$.

For this reason an optimal and safe protocol P has a local activation predicate at each process $A_P \equiv A_{Opt}$.

3.2.4 Optimality for Propagation Based Protocols using Partial Replication

In a partial replicated environment a variable is replicated only to a subset of processes. These processes are the owners of the variable. Only a owner of a variable can read and write on it. Then, there are only two main differences between a propagation based protocol using partial replication and a one using complete replication: (i) upon a write on a variable x issued by an x's owner, an update message is *multicast* to all other x's owners; (ii) let m the number of memory locations, the control information piggybacked onto an update message is a matrix of size n x m.

Then, even a propagation based protocol using partial replication has to be equipped with an activation predicate to delay updates arrived, at an owner, in an order not compliant with $\stackrel{co}{\rightarrow}$.

For this reason, the optimality definition applies even in this case, by stating the condition to avoid a not necessary blocking of an applicable update.

3.3 $ANBKH$ Protocol

In a seminal paper Ahamad et al. [1] introduced the notion of causal memory abstraction. In that paper, the authors also proposed a CRP protocol (hereafter $ANBKH$) implementing such an abstraction on top of a message passing system emulating a reliable broadcast primitive. $ANBKH$ is actually an instance of the general protocol described in Section 3.2, i.e. $ANBKH \in \mathcal{P}$. $ANBKH$ schedules the local application of updates at a process *according to the order established by the happened-before relation of their corresponding send events*. This is obtained by causally ordering message deliveries through a Fidge-Mattern system of vector clocks which considers apply events as relevant events [8].

In $ANBHK$ the activation predicate $A_{ANBHK}(m_{w(y)b}, e)$ for each received message $m_{w(y)b}$ at each process p_i is the following:

$$\nexists\, m_{w(x)a} : (\ send_j(m_{w(x)a}) \rightarrow send_k(m_{w(y)b},) \wedge apply_i(w(x)a) \notin E_i|_e\ )$$

Such a predicate prevents $ANBKH$ to be optimal (but not to be safe). $A_{ANBHK}(m_w, e)$ may be false even in the case each apply event related to a write causally preceding w has occurred. To clarify this point, let us consider a possible $ANBKH$ run (see Figure 1), compliant with history of Example 1.

Let us consider the computation at p_3 process. When p_3 receives the update message $m_{w_2(x_2)b}$, $A_{ANBHK}(m_{w_2(x_2)b}, receipt_3(m_{w_2(x_2)b}))$ is set to false. By Definition 5, even an optimal protocol has the predicate set to false at p_3.

Let us now point out what happens upon the receipt of $m_{w_1(x_1)a}$ and the consecutive application of that update $apply_3(w_1(x_1)a)$. $A_{ANBHK}(m_{w_2(x_2)b}, apply_3(w_1(x_1)a))$ remains set to *false*. This is because

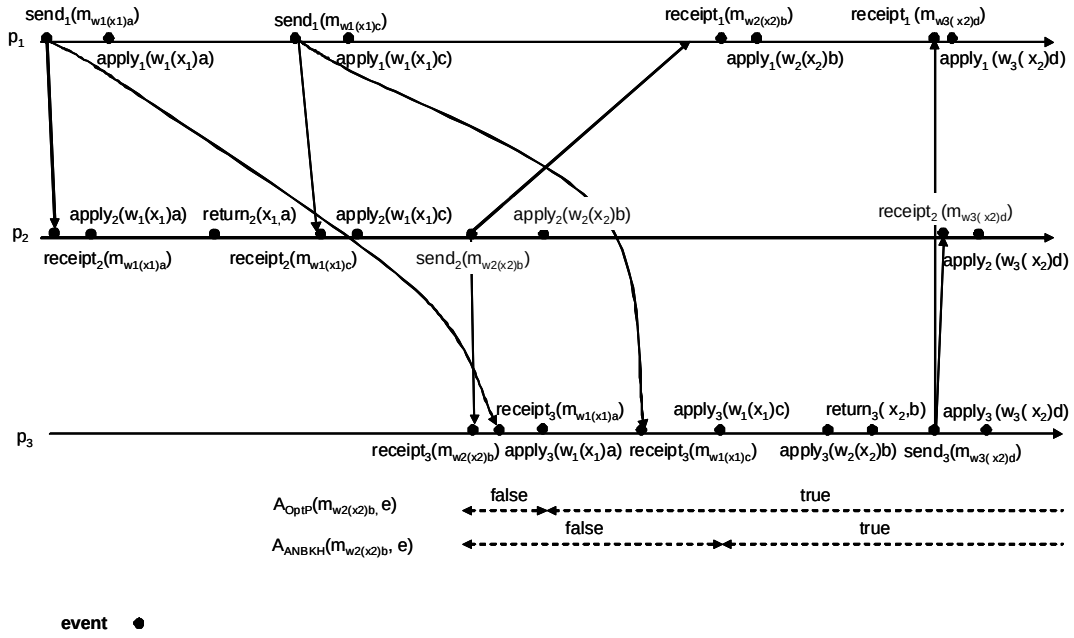

Figure 1: A run of $ANBKH$ compliant with the history of Example 1

there is another message, $m_{w_1(x_1)c}$, that belongs to causal past of $send_2(m_{w_2(x_2)b})$ not yet applied. However at this point, each optimal predicate $A_{Opt}(m_{w_2(x_2)b}, apply(w_1(x_1)a))$ will flip to *true*.

Formally, the non-optimality of $ANBHK$ can be expressed as follows: there exists a distributed computation $\widehat{E} = \{E, \rightarrow\}$ s.t.

$$\exists\ m \in M_{\widehat{E}}, \exists\ e \in E_i : (A_{ANBHK}(m, e) = false \wedge A_{Opt}(m, e) = true)$$

$ANBKH$ is not optimal because of the well-known inability of the "happened before" relation to map in a one-to-one way, cause-effect relations at the application level into relations at the implementation level. This phenomenon is called "*false causality*" [4].

3.4 Related Work

Several other protocols implementing causal consistency have appeared in the literature [2, 6, 16, 25].

The protocols in [6, 25] are propagation-based protocols assuming partial replication. These protocols address the *writing semantics* notion introduced by Raynal-Ahamad in [25]. By using writing semantics, a process can apply a write operation $w(x)$ even though another write $w'(x)$, such that $w'(x) \mapsto_{co} w(x)$, has not been applied yet at that process. In this case we say that w overwrites w' (it is like the event $apply(w')$ is logically executed by a process immediately before $apply(w)$). This overwriting can happen

[4]The false causality notion has been first identified by Lamport in [18], then it has received more attention by Cheriton-Skeen in [10] and by Tarafdar and Garg in [28] in the context of causal message ordering and distributed predicate detection respectively.

only if there is not a write operation $w''(y)$ (with $x \neq y$) such that $w'(x) \mapsto_{co} w''(y)$ and $w''(y) \mapsto_{co} w(x)$. Writing semantics is therefore a heuristic that can help to improve $ANBKH$ by reducing, on the average, the buffer overhead according to the message pattern of the computation. As will see in Section 6 even the optimality allows to reduce the buffer overhead with respect to not optimal protocols. However, writing semantics and optimality are orthogonal notions. Then, writing semantics could also be applied to any optimal protocols.

The protocol presented in [16] is a propagation-based protocol using complete replication. However, differently from protocols in [1, 6, 25], the propagation is not immediate. The propagation is token-based, i.e. each process, until obtains the token, locally executes its write operations without propagate them. When the process obtains the token, it broadcasts a message containing the last write locally executed (overwritten values are not sent). Actually, that mechanism controls the way in which causally ordered relations between operations are created. This control assures that, in each protocol's run, write messages arrive causally ordered at each process. For this reason the protocol does not need a vector clock to track causality and does not need to be equipped with a wait condition.

The protocol presented in [2] copes with dynamic replication and mixes propagation and invalidation-based approaches. The protocol works in a client/server environment and implements causally consistent distributed objects. A distributed object is dynamically replicated: object copies can be created and deleted. A client never holds a copy (it only requests the last value to a server), a permanent server always holds a copy and a potential server can create and delete copies. The copy updating mechanism is propagation-based inside the set of permanent servers and is invalidation-based for potential servers.

In [15] another optimality criterion has been given by Kshemkalyani and Singhal in the context of causal message ordering. This criterion formulates necessary and sufficient conditions on the information to be piggybacked onto messages to guarantee causal message ordering. This optimality criterion differs from the one we have just defined in two aspects. Firstly it has been given on a different ordering relation, namely the "happened before" relation, and, secondly, this criterion is orthogonal to the optimality criterion defined in Section 3.2.3. The latter aims indeed at eliminating any causality relation between messages (i.e., operations) created at the MCS level by the "happened before" relation and that does not have a correspondence at the application level.

The optimality criterion we introduce has some relation with the one shown in [12]. The authors introduce a new relation $\stackrel{s}{\rightarrow}$ that tracks only true causality given an application semantics. From an implementation point of view, they *approximate* $\stackrel{s}{\rightarrow}$ by formulating rules to manage a new system of vector clocks. As a consequence, they only reduce false causality. The inability to remove all false causalities is due to the lack of precise application semantics knowledge. In our case, the application semantics is ruled by $\mapsto^{co}$ and $\stackrel{co}{\rightarrow}$ tracks the application semantics among the events of distributed computations generated by a CRP protocol exploiting a reliable broadcast primitive. That leads to a complete removal of false causality created by the "happened-before" relation at the MCS level.

4 An optimal CRP Protocol ($OptP$)

The CRP protocol presented in this section (hereafter $OptP$) relies on a system of vector clocks, denoted $Write_{co}$, which characterizes $\mapsto_{co}$ [5]. The procedures executed by a MCS process are depicted in Figure 2, 3 and 4. In the following we detail first the data structures and then the protocol behavior.

4.1 Data Structures

Each MCS process p_i manages[6] the following data structures:

$Apply[1...n]$**:** an array of integer (initially set to zero). The component $Apply[j]$ is the number of write operations sent by p_j and applied at p_i.

$Write_{co}[1..n]$**:** an array of integer (initially set to zero). Each write operation $w_i(x_h)a$ is associated with a vector $Write_{co}$, denoted $w_i(x_h)a.Write_{co}$. $w_i(x_h)a.Write_{co}[j] = k$ means that the k-th write operation invoked by an application process ap_j precedes $w_i(x_h)a$ with respect to $\mapsto_{co}$.

$LastWriteOn[1..m, 1..n]$**:** an array of vectors. The component $LastWriteOn[h, *]$ indicates the $Write_{co}$ value of the last write operation on x_h executed at p_i. Each component is initialized to $[0, 0, ..., 0]$.

4.2 Protocol Behavior

When a MCS process p_i wants to perform $w_i(x_h)v$, it atomically executes the procedure `write(`x_h`,v)`, depicted in Figure 2. In particular, p_i increments by one the $Write_{co}[i]$ component to take the program order relation of ap_i into account (line 1) and then it sends an update message $m_{w_i(x_h)v}$ to all MCS processes (line 2). This message piggybacks the variable x_h, the value v and the current value of $Write_{co}$, i.e. the $Write_{co}$ *associated with* $w_i(x_h)v$. Then p_i stores value v in its local variable x_h [7],(line 3), and updates the control structures (lines 4,5). In particular, $LastWriteOn[h]$ is set equal to the $w_i(x_h).Write_{co}$.

When a MCS process p_i wants to perform a read operation on x_h, it atomically executes the procedure `read(`x_h`)` depicted in Figure 3. At line 1, p_i incorporates in the local copy of $Write_{co}$ the causality order relations tracked in the $Write_{co}$ vector associated with the last write operation w, which wrote x_h and stored in $LastWriteOn[h]$. This is done through a component wise maximum between the two vectors. Then the requested value is returned.

Each time an update message $m_{w_u(x_h)v}$ sent by p_u arrives at p_i, a new thread is spawned. The code of this thread is depicted in Figure 4.

If the condition of line 2 in Figure 4 is verified, i.e. the activation predicate of $OptP$ holds, the thread is executed atomically, otherwise p_i's thread waits until the activation predicate at line 2 is verified to guarantee the respect of $\mapsto_{co}$. This means that the vector W_{co} in m, i.e. $w_u(x_h)v.Write_{co}$, does

[5] The formal notion of system of vector clocks is given in Section 4.3.

[6] For clarity of exposition, we omit the subscript related to the identifier of process p_i from the data structures.

[7] When p_i receives a self-sent message, it discards the message. In this way it applies their messages following the order of their sends.

```
WRITE(x_h, v)
1  Write_co[i] := Write_co[i] + 1;                          % tracking ↦_po_i %
2  RELcast [m(x_h, v, Write_co)];                           % send event %
3  x_h := v;                                                % apply event %
4  Apply[i] := Apply[i] + 1;
5  LastWriteOn[h] := Write_co;                              % storing w_i(x_h)v.Write_co %
```

Figure 2: Write procedure performed by the MCS process p_i

```
READ(x_h)
1  ∀k ∈ [1..n], Write_co[k] := max(Write_co[k], LastWriteOn[h].Write_co[k]);   % tracking ↦_ro %
2  return(x_h);                                             % return event %
```

Figure 3: Read procedure performed by the MCS process p_i

not bring any causality order relation unknown to p_i (i.e. $\forall\ t \neq u : w_u(x_h)v.Write_{co}[t] \leq Apply[t]$ and for the sender component u, $Apply[u] = w_u(x_h)v.Write_{co}[u] - 1$). If there exists $t \neq u$ such that $w_u(x_h)v.W_{co}[t] > Apply[t]$ or $Apply[u] < w_u(x_h)v.Write_{co}[u] - 1$, this means that p_u is aware of a write operation w which precedes $w_u(x_h)v$ with respect to $\mapsto_{co}$ and that has not been yet executed at p_i. Then the thread is suspended till the execution of such a write at p_i. Once the condition becomes true, lines 3 to 5 are executed atomically.

Figure 5 shows a run of the protocol with the evolution of the local data structures related to $Write_{co}$. In particular, a local data structure is depicted each time its value changes. To make Figure 5 readable, we do not show the evolution of $LastWriteOn$ at process p_1. When p_2 receives the update message $m_{w_1(x_1)a}$, it compares $w_1(x_1)a.Write_{co}$ with its local $Apply$. Since $w_1(x_1)a.Write_{co}$ is equal to $[1, 0, 0]$, p_2 can immediately apply the value a to its own copy x_1^2. Then ap_2 executes $r_2(x_1)$ which returns the value a establishing in this way a read-from relation between $w_1(x_1)a$ and $r_2(x_1)a$. During the $w_2(x_2)b$'s execution, p_2 broadcast $m_{w_2(x_2)b}$ piggybacking $w_2(x_2)b.Write_{co} = [1, 1, 0]$. It must be noticed that $w_2(x_2)b.Write_{co}$ does not take track of $w_1(x_1)c$ even though it has already been applied when p_2 issues $w_2(x_2)b$. This is due to the fact that the application process ap_2 does not read the value $x_1 = c$ and thus $w_2(x_2)b||_{co}w_1(x_1)c$. When MCS process p_3 receives the update message $m_{w_2(x_2)b}$, it cannot apply b to its copy x_2^3 as there exists a write operation that is in the causal past of $w_2(x_2)b$ whose update has not arrived at p_3 yet (i.e., $w_1(x_1)a$). Therefore the predicate triggering the wait statement at line 2 in Figure 4 is false. Let us finally remark that p_3 can apply b to its own x_2's copy even if it has not already received $m_{w_1(x_1)c}$, because $w_2(x_2)b$ and $w_1(x_1)c$ are concurrent w.r.t. $\mapsto_{co}$.

```
1  when (receipt(m(x_h, v, W_co)) occurs and m was sent by p_u and (u ≠ i)) do
2      wait until ((∀ t ≠ u W_co[t] ≤ Apply[t]) and (Apply[u] = W_co[u] − 1))   % A_opt_P(m,e) %
3      x_h := v;                                                                % apply event %
4      Apply[u] := Apply[u] + 1;
5      LastWriteOn[h] := W_co;                                                  % storing w_u(x_h)v.Write_co %
```

Figure 4: p_i's communication thread

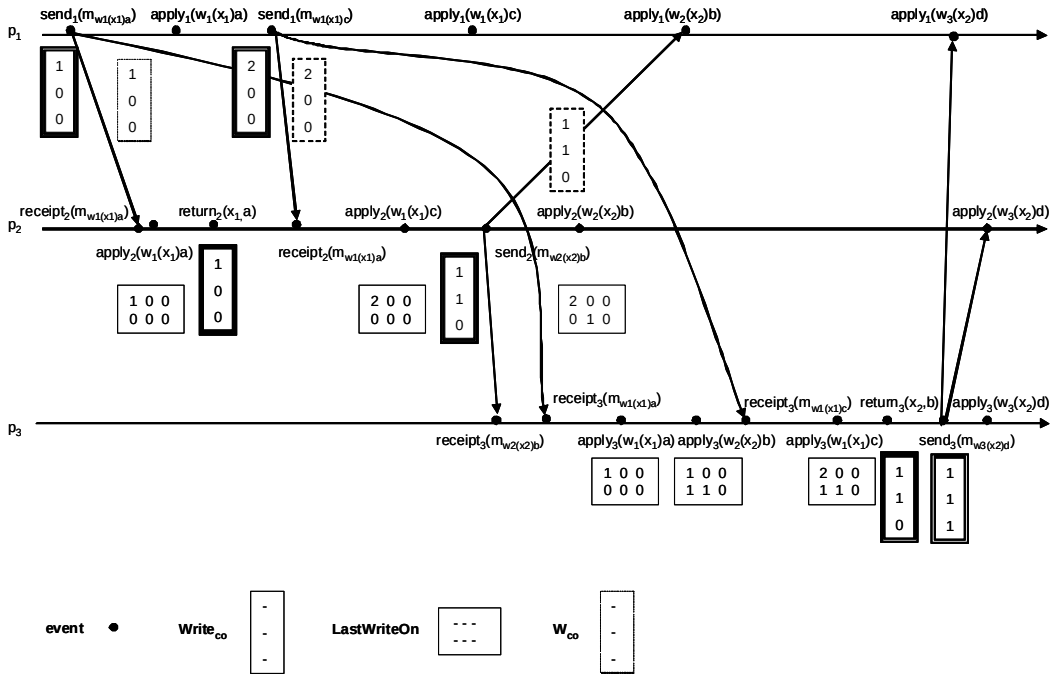

Figure 5: A run of *OptP* compliant with the history of Example 1.

Comparison with $ANBKH$. $ANBKH$ and $OptP$ employ a system of vector clocks: $t[1..n]$ and $Write_{co}[1..n]$ respectively. $ANBKH$ uses it to track the happened-before" relation established at the MCS level while $OptP$ uses $Write_{co}$ to track at the MCS level the causality order relation established at the application level. Therefore, both protocols need to piggyback $O(n)$ control information which is necessary to update the own vector clock system [4, 9]. The local control structure of $ANBKH$ is $O(n)$ while the one of $OptP$ is $O(n * m)$ (i.e. $LastWriteOn$), where n is the number of processes and m the number of variables. This difference is due to the fact that read-from order relations are established during the execution of a read operation (line 1 Figure 3) and this forces to store for each variable x_h, the vector clock associated to the last write operation on x_h.

4.3 Correctness Proof

In this section we first prove that $Write_{co}$ is a system of vector clocks characterizing $\mapsto_{co}$ and $\stackrel{co}{\rightarrow}$ (as the classical vector clock system characterizes $\rightarrow$). We finally prove that $OptP$ is *safe*, *live* and *optimal*. Let us first introduce a notation that we will use in the rest of the paper and two observations whose proofs follow directly from the inspection of the code of Section 4.2.

Notation $w \mapsto_{co}^{k} w'$ with $k \geq 0$ means there exists a sequence of $k \mapsto_{co}$ relations $w \mapsto_{co} w_1 \mapsto_{co} \ldots w_h \mapsto_{co} w_{h+1} \mapsto_{co} \ldots w_{k-1} \mapsto_{co} w_k \mapsto_{co} w'$ and for any relation $w_h \mapsto_{co} w_{h+1}$ does not exist a write operation w'' such that $w_h \mapsto_{co} w'' \mapsto_{co} w_{h+1}$.

Observation 1. *Each component of $Write_{co}$ does not decrease.*

Observation 2. *w is the $k-th$ write operation invoked by the application process $ap_i \Leftrightarrow w.Write_{co}[i] = k$.*

Proof. Since $OptP \in \mathcal{P}$, if w is the $k-th$ write operation invoked by the application process ap_i, then w is the $k-th$ write operation executed by the MCS process p_i. During w's execution, i.e. during the write procedure of Fig. 2, p_i assigns to w the vector clock $Write_{co}[i] = k$ (line 1). □

$Write_{co}$ is a system of vector clocks characterizing $\mapsto_{co}$. This means that for any pair of write operations w and w', it is possible to understand if $w \mapsto_{co} w'$ or $w' \mapsto_{co} w$ or $w||_{co}w'$ comparing $w.Write_{co}$ and $w'.Write_{co}$.

Let $Write_{co} = (w.Write_{co}|w \in H)$ denote the set of vector clocks values associated to each write by the protocol of Section 4.2. Let V and V' be two vectors with the same number of components. We define the following relations on these vectors:

- $V \leq V' \Leftrightarrow \forall k \ : \ V[k] \leq V'[k]$ and
- $V < V' \Leftrightarrow (V \leq V' \wedge (\exists k : V[k] < V'[k])$.

We denote as $V||V' \Leftrightarrow \neg(V < V')$ *and* $\neg(V' < V)$.

We will now show that the system of vector clocks $(Write_{co}, <)$ characterizes $\mapsto_{co}$. Formally:
$\forall\ w, w' : w \neq w', (w \mapsto_{co} w' \Leftrightarrow w.Write_{co} < w'.Write_{co}) \wedge$
$\forall\ w, w' : w \neq w', (w||_{co}w' \Leftrightarrow w.Write_{co}||w'.Write_{co})$.

Lemma 1. $\forall\ w_i, w_j \ \in H : \ w_i \neq w_j, (w_i \mapsto_{co} w_j \Rightarrow w_i.Write_{co} < w_j.Write_{co})$

Proof. Let us consider the notation $w_i \mapsto_{co}^{k} w_j$. The proof is by induction on the value of k.
Basic step. $w_i \mapsto_{co}^{0} w_j \Rightarrow w_i.Write_{co} < w_j.Write_{co}$
We distinguish two cases:
(1) $i = j$. This means that w_i and w_j have been invoked by the same application process ap_i. Then, w_i and w_j have been executed by the same MCS process p_i according to the program order. Each time a MCS process executes a write operation, it performs the write procedure in Figure 2. According to line 1 of Figure 2, each time p_i executes a write operation, it increments $Write_{co}[i]$. Due to Observation 1, if w_i precedes w_j in ap_i program order then $w_i.Write_{co}[i] < w_j.Write_{co}[i]$. Therefore the claim follows (i.e., $w_i.Write_{co} < w_j.Write_{co}$).
(2) $i \neq j$. There exists a read operation invoked by the application process ap_j, denoted $r_j(x_h)$, such that $w_i(x_h) \mapsto_{ro} r_j(x_h)$ and $r_j(x_h) \mapsto_{po} w_j$. ap_j can read the value written by w_i because (1) w_i has been applied at p_j and (2) w_i is the last write on x_h before p_j executes $r_j(x_h)$. For this reason, p_j has set $LastWriteOn[h] := w_i(x_h).Write_{co}$ (line 5 of the synchronization thread Fig. 4). Then, when p_j executes line 1 of the read procedure (Figure 3) we have $Write_{co} \geq w_i(x_h).Write_{co}$. Since each time a process p_j executes a write operation, it increments $Write_{co}$ and from Observation 1, the next write operation executed by p_j, denoted w_j, is associated with a $Write_{co}$ such that $w_j.Write_{co} > w_i.Write_{co}$. Therefore the claim follows.
Inductive Step. $w_i \mapsto_{co}^{k>0} w_j$ then: (i) $\exists\ w^{'} : w_i \mapsto_{co}^{k-1} w^{'}$. By inductive hypothesis we have: $w_i.Write_{co} < w^{'}.Write_{co}$, and (ii) $w^{'} \mapsto_{co}^{0} w_j$. Because of *Basic Step* $w^{'}.Write_{co} < w_j.Write_{co}$. From (i) and (ii), it follows: $w_i.Write_{co} < w_j.Write_{co}$. □

Lemma 2. $\forall\ w_i, w_j\ \in H:\ w_i \neq w_j, (w_i.Write_{co} < w_j.Write_{co} \Rightarrow w_i \mapsto_{co} w_j)$

Proof. The proof is made by contradiction. We have two cases:
1) let us suppose $w_i.Write_{co} < w_j.Write_{co}$ and $w_j \mapsto_{co} w_i$. From Lemma 1, if $w_j \mapsto_{co} w_i$ then $w_j.Write_{co} < w_i.Write_{co}$, therefore we have a contradiction.
2) let us assume $w_i.Write_{co} < w_j.Write_{co}$ and $w_i||_{co}w_j$. The first condition implies $(w_i.Write_{co}[i] = h) \leq (w_j.Write_{co}[i] = k)$. We have two cases:
2.1) $k = h$. From Observation 2, $w_j.Write_{co}[i] = k$ means that the application process ap_j has read the value written by the k-th write operation invoked by ap_i (i.e., w_i), therefore $w_i \mapsto_{co} w_j$. This contradicts the hypothesis that $w_i||_{co}w_j$.
2.2) $k > h$. In this case, ap_j has read the value written by the k-th write operation invoked by ap_i (from Observation 2), denoted w', and then it has written w_j. This means that $w' \mapsto_{co} w_j$. Since $h < k$, $w_i \mapsto_{po_i} w'$ and then $w_i \mapsto_{co} w_j$ contradicting the initial assumption. □

Theorem 1. $\forall w_i, w_j\ \in H:\ w_i \neq w_j, (w_i \mapsto_{co} w_j \Leftrightarrow w_i.Write_{co} < w_j.Write_{co})$

Proof. The claim follows from Lemma 1 and Lemma 2. □

Corollary 1.

$$\forall w_i, w_j\ \in H:\ w_i \neq w_j, (w_i \mapsto_{co} w_j \Leftrightarrow w_i.Write_{co}[i] \leq w_j.Write_{co}[i])$$

Proof. The claim immediately follows from Theorem 1 and from the code of the protocol of Section 4.2. □

Theorem 2. $\forall w_i, w_j\ \in H:\ w_i \neq w_j, (w_i||_{co}w_j \Leftrightarrow w_i.Write_{co}||w_j.Write_{co})$

Proof. The claim immediately follows from Theorem 1 and Definition of concurrency w.r.t. $\mapsto_{co}$. □

Corollary 2.

$$\forall w_i, w_j\ \in H:\ w_i \neq w_j, (w_i||_{co}w_j \Leftrightarrow w_j.Write_{co}[i] < w_i.Write_{co}[i]\ \wedge\ w_i.Write_{co}[j] < w_j.Write_{co}[j])$$

Proof. The claim immediately follows from Theorem 2 and from the code of the protocol of Section 4.2. □

Corollary 3. *$Write_{co}$ is a system of vector clock characterizing $\overset{co}{\rightarrow}$.*

Proof. The claim immediately follows Property 1, taking into account that the update message $m_{w(x)a}$ broadcast during $w(x)a$'s execution is associated with the vector value $w(x)a.Write_{co}$ piggybacked. □

Safety.

Theorem 3. *OptP is safe i.e., let* $\widehat{E} = \{E, \rightarrow\}$*, a distributed computation generated by* $P \in \mathcal{P}$*,*

$$\forall\ m_w, m_{w'} \in M_{\widehat{E}} : (m_w \overset{co}{\rightarrow} m_{w'} \Rightarrow \forall i \in \{1 \dots n\}, apply_i(w) <_i apply_i(w'))$$

Proof. Since Property 1 holds, we develop the proof referring to write operations and $\mapsto_{co}$, to use the same notation (i.e., $w_i \mapsto_{co}^{k} w_j$) and the structure of Lemma 1. The proof is thus by induction on the value of k.

Basic Step. $w_i \mapsto_{co}^{0} w_j$. Let us immediately show that if both w_i and w_j are invoked by the same application process ap_t, then the corresponding MCS process p_t executes them according to the program order (line 3 of write procedure (fig. 2)). Each other MCS process p can execute w_j only if:

$$\forall\ t \neq j \quad w_j.Write_{co}[t] \leq Apply[t] \ \wedge \ for\ t = j \quad Apply[j] = w_j.Write_{co}[j] - 1 \quad (1)$$

Let us suppose that w_i is the $(m) - th$ write invoked by ap_i and w_j is the $(l) - th$ write invoked by ap_j. Then from Observation 2 $w_i.Write_{co}[i] = m$ and $w_j.Write_{co}[j] = l$. There are two possible cases:

- $i = j$. From Corollary 1 and Observation 1 $w_i.Write_{co}[i] < w_j.Write_{co}[i]$, then $w_i.Write_{co}[i] = m$, $w_j.Write_{co}[i] = m + h$ with $h \geq 1$. The condition (1) can be explained as follows: for $t = i$, $Apply[i] = m + h - 1$. Then p has already applied the $(m + h - 1) - th$ write operation issued by p_i and all write operations that precede it in ap_i program order. As w_i is the $(m) - th$ write operation executed by p_i, before applying w_j, p has applied w_i.
- $i \neq j$. From Corollary 1 and Observation 1 $w_i.Write_{co}[i] < w_j.Write_{co}[i]$, then if $w_i.Write_{co}[i] = m$, $w_j.Write_{co}[i] = m + h$ with $h \geq 0$. In this case the condition (1) can be explained as follows: for $t = i$, $Apply[i] \geq m + h$. Then p has already applied the $(m + h) - th$ write operation invoked by the application process ap_i and all write operations that precede it in ap_i program order. As w_i is the $(m) - th$ write operation executed by p_i, before applying w_j, p has applied w_i.

Inductive Step. $k > 0$. (i) $\exists\ w' : w_i \mapsto_{co}^{k-1} w'$. By inductive hypothesis we have: $apply_k(w_i) \rightarrow apply_k(w')$ at process p_k. (ii) $w' \mapsto_{co}^{0} w_j$. Because of *Basic Step* $apply_k(w') \rightarrow apply_k(w_j)$ at the MCS process p_k.

From (i) and (ii), it follows: $apply_k(w_i) \rightarrow apply_k(w_j)$ at process p_k. □

Liveness.

Theorem 4. *All write operations invoked by an application process are eventually applied at each MCS process.*

Proof. Let us assume by the way of contradiction there exists a write operation w_j invoked by the application process ap_j and then issued by the MCS process p_j that can never be applied by p_i on its local copy. This can happen if $\exists k \neq j \in \{1, \dots n\} : Apply[k] < w_j.Write_{co}[k]$ or $k = j, Apply][k] < w_j.Write_{co}[k] - 1$. This means that there exists at least one update message m_w sent by p_k (carrying a

write w executed by p_k) such that $w \mapsto_{co} w_j$, which has not been received at p_i yet. In this case we say that w blocks w_j.

Since (i) communication channels are reliable, (ii) each process executes a computational step in a finite time, (iii) the operative system scheduler is fair and (iv) each operation is reliably broadcast to all processes, the update message m_w will be eventually received by p_i. Now we have two cases:

1. w can be applied at p_i unblocking w_j, therefore the assumption is contradicted and the claim follows;
2. there exists a write operation w' that blocks w. In this case we can apply the same argument to w' and due to the fact that (i) the number of write operations that precede w_j wrt $\mapsto_{co}$ is finite and (ii) $\mapsto_{co}$ is a partial order, then in a finite number of steps we fall in case 1.

□

Optimality.

Theorem 5. *OptP is optimal (see Definition 5).*

Proof. The proof is made by contradiction. Let us assume that a receipt event of the update message $m_{w(x)v}$ sent by p_u occurs at p_i and belongs to a distributed computation $\widehat{E}$ generated by $OptP$. Let us then suppose that there exists an event $e \in E_i$ such that: $receipt_i(m_{w(x)v}) <_i e$ and $A_{OptP}(m_{w(x)v}, e) = false$ while $A_{Opt}(m_{w(x)v}, e) = true$. If $A_{OptP}(m_{w(x)v}, e) = false$, it means that the application of the update corresponding to $m_{w(x)v}$ is delayed at p_i. In this sense, according to line 2 of Figure 4, a message $m_{w_u(x)v}$ is delayed by $OptP$ iff one of the following conditions holds:

1. $\exists\ t \neq u\ w_u(x)v.Write_{co}[t] > Apply[t]$
2. $Apply[u] \neq w_u(x)v.Write_{co}[u] - 1$.

¿From Corollary 3, $Write_{co}$ is a system of vector clock characterizing $\xrightarrow{co}$. Then, in both cases, there exists an update message $m_{w_k(y)b} \in M_{\widehat{E}}$, respectively with $k \neq u$ or $k = u$, that precedes $m_{w_u(x)v}$ w.r.t. $\xrightarrow{co}$ and that has not yet been applied at p_i. In this case even $A_{Opt}(m_w, e)$ is false, contradicting the initial hypothesis. □

5 An Efficient Implementation of the Protocol $OptP$ ($OptP_{ef}$)

Thanks to Theorem 1 and to the transitivity property of the causality order relation $\mapsto_{co}$, we can apply to $OptP$ optimization techniques studied in the context of efficient representations of a system of vector clocks capturing the happened-before relation. More specifically, we derive an efficient implementation of $OptP$, namely $OptP_{ef}$, by using the Singhal-Kshemkalyani (SK) [27] and the direct dependency tracking techniques [15, 24]. Both techniques aim to reduce the number of entries of a system of vector clocks to be piggybacked onto protocol messages. Let us remark that they pursue their aims through two orthogonal principles:

- SK avoids an MCS process sends vector clock entries that did not change between two successive messages sent out to the same MCS process [27].
- In direct dependency tracking a MCS process piggybacks onto a protocol message corresponding to a write operation w those write operations $w^{'}$ on which w execution *directly* depends, that is all $w^{'}$ such that $w^{'} \mapsto^{k}_{co} w$ with $k = 0$ [15, 24].

5.1 Adopting Singhal-Kshemkalyani (SK) technique

Between two successive messages (send events) sent by p_i to p_j, only a few component of vector $Write_{co}$ are expected to change. In such a case, it suffices to piggyback these entries, i.e., the pairs $(proc_{id}, Write_{co}[proc_{id}])$, on protocol messages. Therefore, even though this technique saves communication bandwidth, it introduces local memory overhead. In fact, a MCS process must keep track of the last $Write_{co}$ sent to each distinct process in order to select the set of pairs to piggyback onto each message, denoted W_{SK}[8] (this set is initialized to $\emptyset$). Figure 6 shows the write procedure and the synchronization thread of $OptP$ augmented with SK technique (the read operation is the same of $OptP$) and Figure 7 depicts the behavior of this protocol on the run depicted in Figure 5. For example when p_3 sends the message notifying $w(x_2)d$, W_{SK} corresponds to the set $W_{SK} = \{(1,1);(2,1);(3,1)\}$ as all three components of $Write_{co}$ changed since the last p_3 sending (the value of such $Write_{co}$ vectors are depicted in gray rectangles in 7). Let us note that the last sending of p_3 is virtual as it corresponds to the initial value of $Write_{co}$.

As each message piggybacks only a subset of the vector clock entries, an additional data structure is necessary at each process to rebuild the $Write_{co}$ vector associated with a write and to be stored in $LastWriteOn$ (see line 5 of Figure 4 and line 6 of Figure 6(b)) in order to correctly track read-from relations (see Line 1 of Figure 3). To this aim, each MCS process p_i manages the following additional data structure:

$LastWrite_{co}By[1..m, 1..n]$**:** The component $LastWrite_{co}By[u]$ is the $Write_{co}$ vector associated with the last write operation invoked by the application process ap_u and executed by the MCS process p_i. Each component of $LastWrite_{co}By$ is initialized to zero. Each time a message is received from p_u and the corresponding write w executed at p_i, the vector $w.Write_{co}$ is rebuilt using the set of pair $w.W_{SK}$ which are changed since the last write operation issued by p_u (see line 5 of Figure 6(b)). $LastWrite_{co}By[i]$ corresponds to the vector clock associated to the last send event of p_i and it is therefore used to select pairs in W_{SK} at lines 2-3 of write procedure in Figure 6(a).

When process p_1 executes $w_1(x_1)a$, it compares $Write_{co}$ with $LastWrite_{co}By[1]$ to find the set of couples, denoted $W_{SK} = \{(1,1)\}$, to be piggybacked onto the message (lines 2-3 Fig. 6(a)). When a MCS process p_i with $i = 2, 3$ receives a message, it must verify that the elements of W_{SK} verify the apply condition. Then, p_i first rebuilds the $Write_{co}$ of $w_1(x_1)a$ (line 5 Fig. 6(b))and finally it stores this vector in $LastWriteOn[1]$ as in $OptP$ protocol (line 6).

[8] In a broadcast context, this means a process has to keep track of the last $Write_{co}$ sent.

```
WRITE(x_h, v)
1   Write_co[i] := Write_co[i] + 1;                                   % tracking ↦_{po_i} %
2   ∀j  if (Write_co[j] ≠ LastWrite_coBy[i][j])
3         then W_SK := W_SK ∪ (j, Write_co[j]);     % selecting pairs that changed since the last write by p_i %
4   RELcast [m(x_h, v, W_SK)];                          % send event piggybacking selected pairs %
5   x_h := v;                                                                          % apply event %
6   Apply[i] := Apply[i] + 1;
7   LastWrite_coBy[i] := Write_co;
8   LastWriteOn[h] := Write_co;                                  % storing w_i(x_h)v.Write_co %
9   W_SK := ∅;
```

(a)

```
1  when (receipt(m(x_h, v, W_SK)) occurs and m was sent by p_u and (u ≠ i)) do
2    % exec. of RELrcv %
3    wait until (∀(t, k) ∈ W_SK : (t ≠ u   k ≤ Apply[t])   and   (t = u   Apply[u] = k − 1));
4      x_h := v;                                                                    % apply event %
5      Apply[u] := Apply[u] + 1;
6      ∀(t, k) ∈ W_SK   LastWrite_coBy[u][t] := k;                % rebuilding w_u(x_h)v.Write_co %
7      LastWriteOn[h] := LastWrite_coBy[u];                         % storing w_u(x_h)v.Write_co %
```

(b)

Figure 6: p_i's write procedure and communication thread of $OptP$ augmented with SK technique

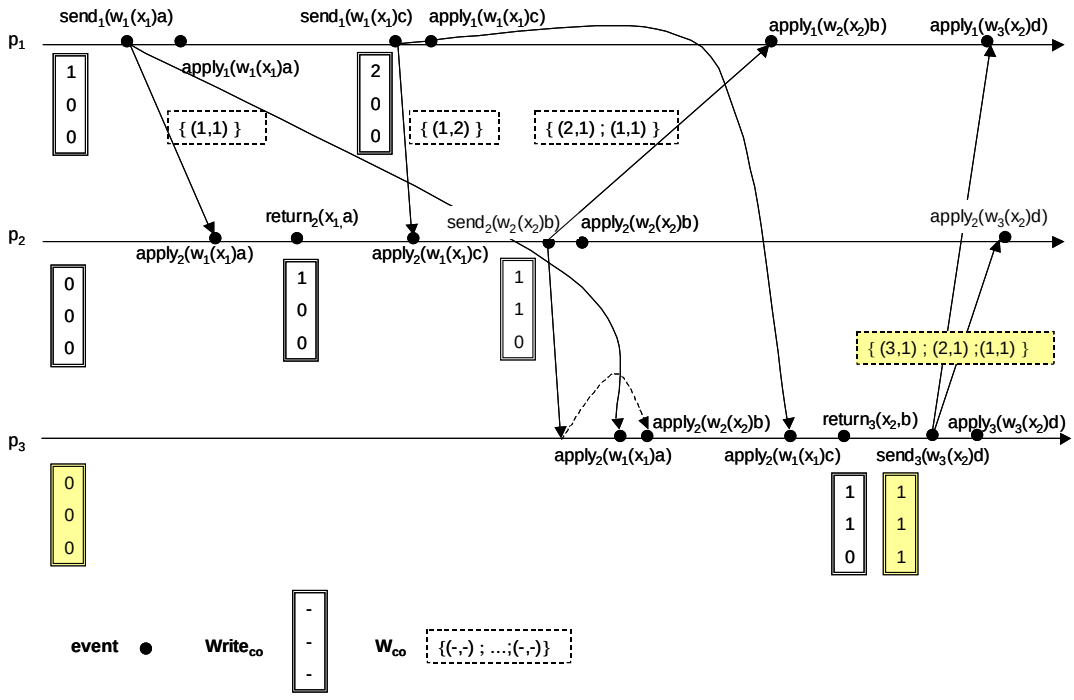

Figure 7: Run of $OptP$ augmented with SK technique compliant with the history of Example 1

5.2 Exploiting the direct dependency tracking

This technique is based on the notion of causality graph introduced in [15, 24], in the context of causal deliveries in message passing systems. A write causality graph is a directed acyclic graph whose vertices are all write operations belonging to H. According to the notation introduced in Section 4.3, there is a direct edge from w to w' if $w \mapsto^0_{co} w'$ [9]. In this case we also say that w is an immediate predecessor of w' in the write causality graph. It trivially follows that each write operation can have at most n immediate predecessors, one for each process. Figure 8 shows the write causality graph of the history of Example 1. The write $w_1(x_1)c$ is a $w_3(x_2)d$'s immediate predecessor while $w_1(x_1)a$ is an immediate predecessor of $w_1(x_1)c$ and $w_2(x_2)b$.

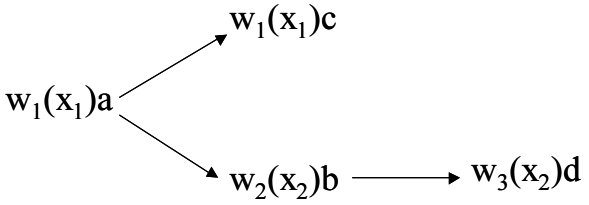

Figure 8: The causality graph associated to the history of Example 1

[9] The write causality graph is actually the "anti-transitive closure" (otherwise known as Hasse Diagram) of the restriction of relation $\mapsto_{co}$ to the write operations.

The protocol. Let us consider again Figure 8, from an operational point of view this graph says that the execution of $w_3(x_2)d$ at each MCS process cannot occur before the execution of $w_2(x_2)b$ ($w_3(x_2)d$'s immediate predecessor). Due to transitivity, the execution of $w_2(x_2)b$ cannot occur before the execution of $w_1(x_1)a$. This simple principle allows to reduce on the average the amount of control information on each message as each write operation has only to piggyback information concerning its immediate predecessors. Immediate predecessors of a write operation act therefore as a *causal barrier (CB)* of that write (e.g.$w_2(x_2)b$ represents $w_3(x_2)d$'s causal barrier) and each protocol message needs only to piggyback information related to members of its causal barrier. To this aim:

$LastWriteOn$**:** becomes a vector of pairs $(Write_{co}, W_{CB})$ one for each variable. Therefore, let $w(x_h)v$ be the last write operation executed by p_i on x_h, $LastWriteOn_i[h] = (Write_{co}, W_{CB})$ means that the vector clocks and the causal barrier associated with $w(x_h)v$ are $Write_{co}$ and W_{CB} respectively.

W_{co}^{-}**:** is an additional data structure endowed at each MCS process. It stores the union of the sets representing the causal barriers of write operations whose value has been read by the application process ap_i since last ap_i's write operation (this set is initialized to $\emptyset$). This means that once a write operation w becomes a member of W_{co}^{-}, w will be no longer an immediate predecessor of any successive write operation executed by p_i. This set is updated each time the read procedure is executed (line 2) and is set to an empty set just before executing a write operation (line 11).

The protocol is obtained by adding few lines to the protocol shown in Fig.6 and modifying the read procedure shown in Fig 3. Each time a write operation $w_i(x_h)v$ is executed, first the SK technique is applied and the set W_{SK} is computed (line 5 Fig.9(b)). Then, $w_i(x_h)v$'s causal barrier, W_{CB}, is determined by removing from W_{SK} all those pairs that are not immediate predecessors of $w_i(x_h)v$ and that are stored in W_{co}^{-} (line 6 Fig. 9(b)). Then W_{CB} is piggybacked onto the message notifying $w_i(x_h)v$ as control information (line 7 Fig. 9(b)). Once a MCS process p_i receives this message, it first checks that any member of $w_i(x_h)v$'s W_{CB} has been already executed at p_i (line 2 Fig.9(c)), then $w_i(x_h)v.Write_{co}$ is rebuilt according to SK technique (line 5 Fig.9(c)) and finally this vector and $w_i(x_h)v$'s causal barrier are stored into $LastWriteOn[h]$ (line 6 Fig.9(c)).

Line 5 of Figure 9(b) actually proves that the pairs piggybacked by the protocol $OptP_{ef}$ are fewer than or equal to the ones shown in Figure 6. As an example, let us consider the write operation $w_3(x_2)d$ executed by p_3 and shown in Figure 7, the corresponding message piggybacks the set $W_{SK} = \{(1,1);(2,1);(3,1)\}$. However, the pair $(1,1)$ refers to the first write operation $w_1(x_1)a$ executed by p_1. This operation is not an immediate predecessor of $w_3(x_2)d$ (see Figure 8) as it belongs to W_{co}^{-} at the time of $w_3(x_2)d$'s writing (the pair $(1,1)$ is stored into W_{co}^{-} during the read operation $read(x_2)$ executed at p_3). Therefore when executing line 5 of Figure 9(b) this pair is removed from W_{SK} and the set $\{(2,1);(3,1)\}$ is piggybacked onto the protocol messages as $w_3(x_2)d$'s causal barrier. This run is shown in Figure 10.

```
READ(x_h)
1  ∀k ∈ [1...n], Write_co[k] := max(Write_co[k], LastWriteOn[h].Write_co[k]);   % tracking ↦_ro %
2  W⁻_co := W⁻_co ⋃{LastWriteOn[h].W_co \ W⁻_co}; % write operations whose value has been read by p_i since last p_i's write. %
3  return(x_h);                                                                 % return event %
```

(a)

```
WRITE(x_h, v)
1   write(x_h, v)
2     Write_co[i] := Write_co[i] + 1;                                  % tracking ↦_po_i %
3     LastWriteBy[i] := Write_co;
4     ∀j if (Write_co[j] ≠ LastWriteBy[i][j])
5           then W_SK := W_SK ∪ (j, Write_co[j]);   % selecting pairs that changed since the last write issued by p_i %
6     W_CB := W_SK − W⁻_co;                          % computing the CB of w_i(x_h)v in the write causality graph %
7     RELcast [m(x_h, v, W_CB)]Π − p_i;   % send event piggybacking on the message the causal barrier of w_i(x_h)v %
8     x_h := v;                                                                % apply event %
9     Apply[i] := Apply[i] + 1;
10    LastWriteOn[h] := (Write_co, W_CB);          % storing w_i(x_h)v.Write_co and the CB of w_i(x_h)v %
11    W_SK := ∅;
12    W⁻_co := ∅;
```

(b)

```
1  when (receipt(m(x_h, v, W_CB)) occurs and m was sent by p_u and (u ≠ i)) do
2          % execution of RELrcv %
3    wait until (∀(t, k) ∈ W_CB : (t ≠ u   k ≤ Apply[t])   and   (t = u   Apply[u] = k − 1));
4      x_h := v;                                                              % apply event %
5      Apply[u] := Apply[u] + 1;
6      ∀(t, k) ∈ W_CB   LastWriteBy[u][t] := k;                % rebuilding w_u(x_h)v.Write_co %
7      LastWriteOn[h] := (LastWriteBy[u], W_CB);     % storing w_u(x_h)v.Write_co and the CB of w_u(x_h)v %
```

(c)

Figure 9: $OptP_{ef}$ pseudo code

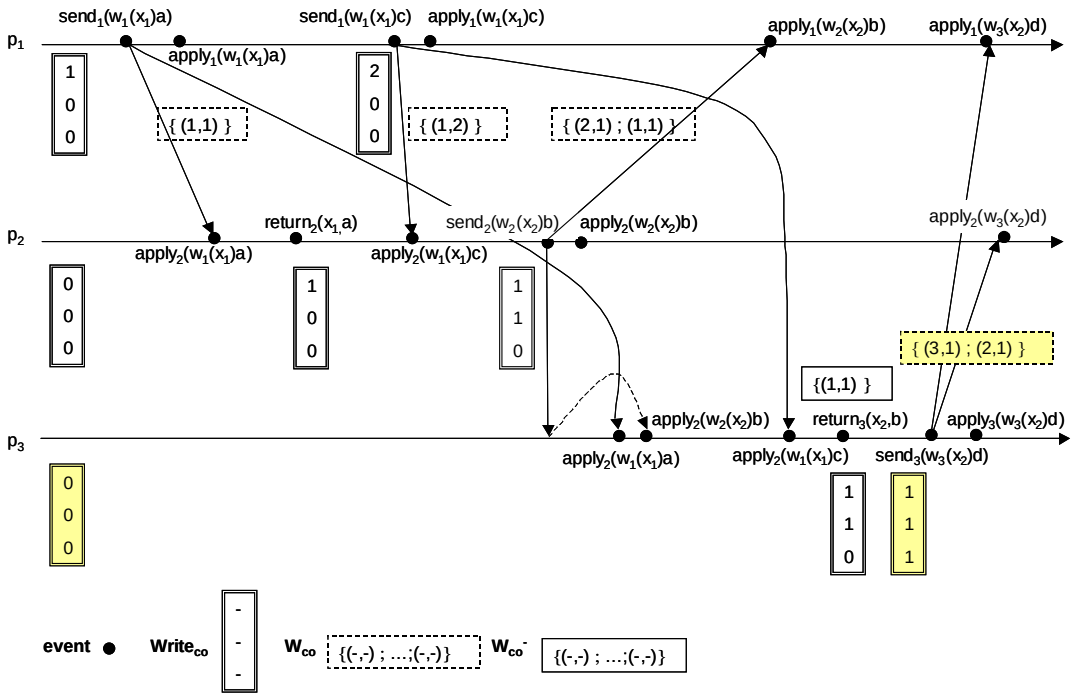

Figure 10: A run of $OptP_{ef}$ compliant with the history of Example 1.

6 Simulation

This section presents simulation results performed to compare $ANBKH$ and $OptP$ in terms of message buffer overhead.

6.1 Message Buffer Overhead.

Message buffer overhead measures the utilization of buffers local at each MCS process used to store update messages. A process stores a message when this message is not immediately applicable upon its receipt. The message remains in the buffer until it is applied. Upon a message m_w receipt, an activation predicate equals to false determines that m_w enters the buffer. Ideally, the activation predicate should determine even when the message m_w exits from the buffer, i.e. as soon as the activation predicate returns true, $apply(w)$ immediately occurs. However, this assumption ignores that any operative system could choose to schedule other events, concurrent with $apply(w)$, before it. These events may be $send, receive$ or either $apply$ events of writes concurrent with w. Then, the buffer utilization depends on (i) the activation predicate and (ii) the operative system scheduling policies.

We describe the relation between message buffer overhead and the communication primitive [7, 8, 13] used by a CRP protocol in the following two comments.

Reliable Broadcast and FIFO broadcast. Any safe MCS protocol that communicates via reliable/FIFO broadcast has to reorder incoming update messages at each process to respect causal con-

sistency. This reorder is managed by the activation predicate. In this case we focus only on the buffer utilization due to the activation predicate behavior of any MCS protocol, without further considering the operative system behavior. In practice, we assume the impact of the operative system on the buffer utilization as negligible with respect to the one due to the MCS protocol activation predicate. This assumption is justified by the fact that an activation predicate of a MCS protocol relying on a reliable/FIFO broadcast has to reorder messages by keeping a message in the buffer until some other message arrives from the network. This time is some order of magnitude greater than the time taken by a good operative system to schedule an event.

Causal Broadcast. Any MCS protocol relying on a causal broadcast can apply each message upon its arrive. An activation predicate at MCS level is not necessary, the reordering necessary to ensure causal consistency is embedded in the communication primitive. Indeed, the communication primitive uses buffering to perform reorder. In other words, an activation predicate works at communication level deciding for the delivery of messages. The goodness of the activation predicate affects the buffer overhead exactly as in the case of the MCS activation predicates. However, since this buffering is transparent at MCS level, it is no further considered.

Then the evaluation of an MCS protocol implementing causal consistency in terms of message buffer overhead makes sense only if the protocol uses a reliable/FIFO broadcast.

Clearly, the comparison between different MCS protocols in terms of message buffer overhead makes sense only if these two protocols employ the same broadcast primitive. We consider in the simulation the comparison between $ANBKH$ and $OptP$ both using a reliable broadcast primitive [10].

6.2 Simulation Description

We adopted the discrete event simulator OMNeTpp 2.3b1 [23]. We simulated distributed computations with 10, 20, 30 and 50 processes. The message propagation time and the time to execute an operation by a process are truncnormally distributed with mean value equal to 1 and deviation equal to 1.2 time units. Analogously, time between two successive operations in a process is truncnormally distributed with mean value equal to 9 and deviation equal to 4 time units. Each simulation issues 2000 operations per process on a single replicated variable and the percentage of write operations is defined by a bernoulli distribution with probability p. Simulation experiments were conducted by varying the percentage of write operations (denoted $\%write$) at each application process from 10% to 100% w.r.t. the overall sum of read and write operations. The results are expressed in terms of:

- the percentage of the ratio $\%B$ between the average number of buffered messages by a MCS process of a given protocol (i.e., $OptP$ or $ANBKH$) and the average number of messages received by a MCS process;

For each value shown in the plots we did 40 simulation runs with different seeds and the results were within four percent each other, so variance is not reported.

[10] The broadcast is just a useful abstraction, the protocols may employ even n point-to-point communications.

$\%B$ vs. $\%write$ Figure 11 shows eight plots: four of $ANBKH$ and four of $OptP$. These plots rise three interesting observations:

- *$\%B$ performance gap.* There is roughly one order of magnitude gap between the messages buffered by $ANBKH$ and the ones buffered by $OptP$. This is due to the fact that $ANBKH$ guarantees causal histories by ensuring causal ordering on message deliveries (independently of the read/write semantics of a shared memory system). This results in a weak sufficient condition which leads to very poor performance with respect to $\%B$.

- *$\%B$ of OptP is independent of the number of MCS processes of the computation n.* As far as $ANBKH$ is concerned $\%B$ depends on n. The higher is this number, the larger $\%B$ is. This is again due to the fact that $ANBKH$ guarantees causal histories by ensuring causal ordering on message deliveries. Therefore the higher the number of MCS processes is, the higher is the number of messages from distinct processes received by a MCS process p before the co-located application process invokes a write operation w. This means that the number of sending events (issued by MCS processes different from p) that will precede the sending of w with respect to $\rightarrow$ will tend very quickly to n. This implies, in its turn, a probability of causal message ordering violations which is monotonically increasing with n as well.

  In $OptP$, $\%B$ is almost independent of n (all the plots are very closed each other within the interval of the variance of the simulation). This is due to the fact that the delivery of a message by a process p_i does not always lead to the creation of a causality order relation. In particular, this happens only if the application process ap_i reads the value written, that is the one piggybacked by the message delivered.

- *$\%B$ monotonically increases with the number of write operations in H both in OptP and in $ANBKH$ plots.* As $ANBKH$ ensures causal message ordering, the larger is the number of messages in a computation, the higher the probability of a causal order violation among those messages is and this, in its turn, increases $\%B$. In $OptP$ plots, $\%B$ also increases with the number of write operations of a computation. However, this increment is lesser than the one of $ANBKH$. Let us consider indeed an ideal scenario of a computation with 100% of write operations. This corresponds to the worst case scenario for $ANBKH$ (i.e., the computation with the presence of the highest number of messages and therefore the highest probability of causal message ordering violations). If we consider this scenario with respect to the relation $\mapsto_{co}$, it is easy to see that all $\mapsto_{co}$ relations between write operations are due to the program order $\mapsto_{po}$. Because of the absence of read operations, there is indeed no $\mapsto_{co}$ relation due to the read-from order $\mapsto_{ro}$. From the point of view of the computation this means that each pair of update messages are concurrent wrt $\stackrel{co}{\rightarrow}$. Therefore $OptP$ only buffers those messages which are out of FIFO order.

From the previous discussion, it comes out that performance of $ANBKH$ in terms of buffered messages is a function of the underlying message pattern generated by a history. More messages in this pattern means worse performance and this is dominated by two factors: the number of writes w.r.t. the reads and the number of processes of the computation. Performance of $OptP$ is primarily dominated by

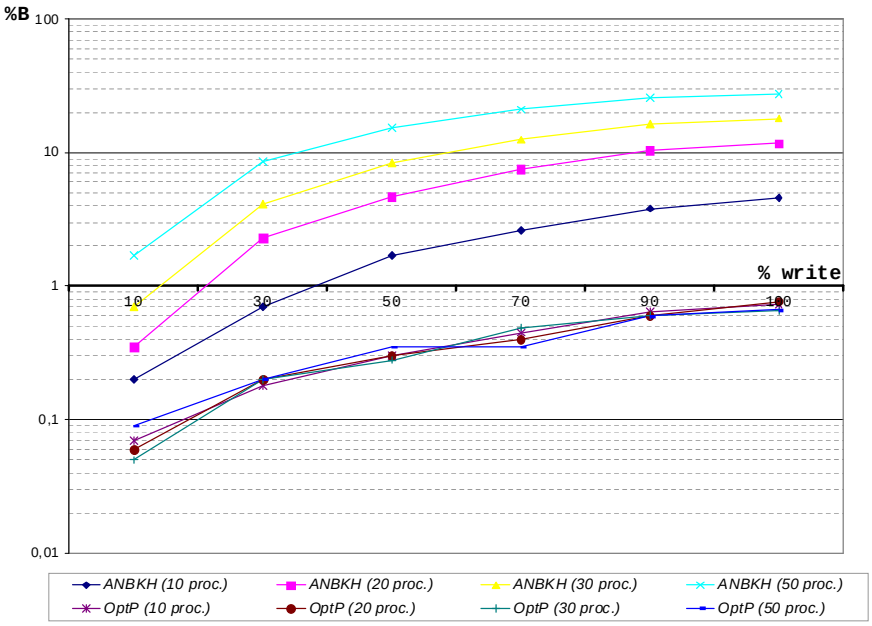

Figure 11: %B versus %write

the number of read-from relations established in a history which of course depends on the semantics of the underlying application. The number of messages of the computation and the number of processes have only a second order impact.

7 Conclusion

This paper has introduced an optimality criterion for protocols based on complete replication and propagation of the updates, enforcing causally consistent histories at the application level. This optimality criterion is based on a predicate evaluated on the computation generated by the protocol. An optimal CRP protocol ensures the maximum allowed concurrency by the causality criterion. Operationally, this fact influences the number of update messages buffered at each MCS process. An optimal CRP protocol buffers only for the strictly necessary time those updates arrived too early at a process and whose immediate local update execution would violate the causality order relation.

We proposed a CRP protocol $OptP$ exploiting a reliable broadcast primitive which is optimal with respect to that criterion. We showed, through a simulation study, that optimality has a strong impact on message buffer overhead at a MCS process. $OptP$ allows a buffered message saving of one order of magnitude with respect to $ANBKH$. Moreover, we derive an efficient implementation of $OptP$, namely $OptP_{ef}$, based on the formal proof that $OptP$ embeds a system of vector clocks capturing the causality-order relation among operations. $OptP_{ef}$ has been derived following two orthogonal principles: (i) avoiding a process sends vector clock entries that did not change between two successive messages sent out to the same process [27], and, (ii) a process piggybacks onto a protocol message only those vector clock entries that correspond to the immediate predecessors of the current write operation with respect

to $\mapsto_{co}$ in the write causality graph of the computation [15, 24].

The paper has also formally showed that $OptP$ uses a system of vector clocks characterizing $\mapsto_{co}$. This proof and the transitivity property of $\mapsto_{co}$ give us powerful optimization tools, borrowed from efficient representation of Fidge-Mattern vector clocks [11, 22], to derive efficient implementations of $OptP$.

Acknowledgements

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions that greatly improved the content and the presentation of this work.

References

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

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

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

[4] R. Baldoni and G.Melideo: On the Minimal Information to Encode Timestamps in a Distributed Computation. *Information Processing Letter* 84(3): 159-166, 2002.

[5] R. Baldoni, A. Milani, S. Tucci Piergiovanni. Optimal Propagation based Protocols implementing Causal Memories. *Technical Report 19-04*, Dipartimento di Informatica e Sistemistica, Universita' di Roma "La Sapienza", 2004.

[6] R. Baldoni, C. Spaziani, S. Tucci Piergiovanni and D. Tulone. An Implementation of Causal Memories using the Writing semantics. *Proceedings of 6th International Conference On Principles Of Distributed Systems*, HERMES Press, 43-52, 2002.

[7] K. P. Birman, T. A. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1): 47-76, 1987.

[8] K. P. Birman, A. Schiper, P. Stephenson. Lightweigt Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems* 9(3): 272-314, 1991.

[9] B. Charron-Bost: Concerning the Size of Logical Clocks in Distributed Systems. *Information Processing Letter* 39(1): 11-16, 1991

[10] D. R. Cheriton and Dale Skeen. Understanding the Limitations of Causally and Totally Ordered Communication. *Proceedings of 14th ACM Symposium on Operating Systems Principles*, ACM press, 44-57, 1993.

[11] C. J. Fidge. Logical Time in Distributed Computing Systems. *IEEE Computer* 24(8): 28-33, 1991.

[12] P. Gambhire, AD Kshemkalyani. Reducing false causality in causal message ordering, *Proceedings of International Conference on High Performance Computing*, LNCS 1970, 61-72, 2000.

[13] V. Hadzilacos, S. Toueg. Fault Tolerant Broadcasts and Related Problems, *Distributed Systems* (second edition), Mullender, 5, 1994.

[14] M. Herlihy, J.M. Wing: Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems.* 12(3): 463-492, 1990.

[15] A.D. Kshemkalyani, M. Singhal. Necessary and sufficient conditions on the information for causal message ordering and their optimal implementation, *Distributed Computing*, Vol 11, 91-111, 1998.

[16] E. Jimenez, A. Fernández and V. Cholvi. A Parametrized Algorithm that Implements Sequential, Causal, and Cache Memory Consistency, *in Brief Announcements of the Proceedings of 15th International Symposium on Distributed Computing*, 2001. An extended version appeared in *Proceedings of 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing*, 2002.

[17] E. Jimenez, A. Fernández and V. Cholvi. On the interconnection of causal memory systems, *in press in Journal of Parallel and Distributed Computing*, 2004.

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

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

[20] L. Lamport. On Interprocess Communication. Part I: Basic Formalism. *Distributed Computing* 1(2): 77-85, 1986.

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

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

[23] OMNeT++ home page: http://www.omnetpp.org/

[24] R. Prakash, M. Raynal, M. Singhal. An Adaptive Causal Ordering Algorithm Suited to Mobile Computing Environments. *Journal of Parallel and Distributed Computing* 41(2): 190-204, 1997.

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

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

[27] M. Singhal, A. D. Kshemkalyani. An Efficient Implementation of Vector Clocks. *Information Processing Letters* 43(1): 47-52, 1992.

[28] A. Tarafdar, V.K. Garg. Addressing False Causality while Detecting Predicates in Distributed Programs. *8th International Conference on Distributed Computing Systems*, 94-101, 1998.

[29] http://www.dis.uniroma1.it/ midlab