An Optimal Protocol for Causally Consistent Distributed Shared Memory Systems*

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

Abstract

Distributed shared memory (DSM) is one of the main abstraction to implement data-centric information exchanges among a set of processes. Ensuring causal consistency means all operations executed at each process will be compliant to a cause effect relation. This paper first provides an optimality criterion for a protocol P that enforces causal consistency on a DSM. This criterion addresses the number of write operations delayed by P (write delay optimality). Then we present a protocol which is optimal with respect to write delay optimality and we show how previous protocols presented in the literature are not optimal with respect to such a criterion.

1. Introduction

Data-centric communication is one of the most interesting abstraction for exchanging information among a set of processes which are decoupled in time, space and flow. *Distributed Shared Memory* (DSM) is a classic model that provides such data-centric exchanges where decoupling in space, flow and time means that processes can execute wait-free read and write operations on a common replicated variable. To ensure *causal consistency* in a DSM processes have to *agree* on the relative ordering of operations (read/write) that have a *cause effect* relation [1]. There is a cause effect relation between two operations o_1 and o_2, denoted $o_1 \mapsto_{co} o_2$ iff one of the following conditions is true: (i) o_1 and o_2 are issued by the same process and o_1 precedes o_2 (*process 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 relation*) or (iii) there exists an operation o such that $o_1 \mapsto_{co} o$ and $o \mapsto_{co} o_2$. So two independent writes wrt $\mapsto_{co}$ can be perceived in different order by two different processes. This makes causal memory a low latency abstraction with respect to stronger consistency criteria such as sequential [9] and atomic consistency [10] (also called linearizability [6]) as it admits more executions and, hence, more concurrency.

When implemented over an asynchronous distributed system replicating memory locations at each process, the causal memory abstraction has been traditionally realized through wait-free local readings and broadcasting write operations to other processes (e.g., [1], [14], [2]). In this way when two operations are related by $\mapsto_{co}$ they are also related by the "happened-before" relation, denoted $\rightarrow$, introduced by Lamport in [8] (the viceversa is not necessarily true [1]).

Therefore ensuring *causal delivery of messages*[2] through a *Fidge-Mattern vector clocks* ([5, 11]) is a sufficient condition to guarantee causal consistent histories with respect to $\mapsto_{co}$. Moreover, protocols implementing a cause-effect abstraction usually delay each message carrying a write operation w arrived too early at a process p. This implies buffering of w at p. Such write will be executed after that all operations that causally affected w will be executed at p. Therefore such protocols can be compared among each other with respect to the number of messages associated with writes whose application at a process is delayed to enforce $\mapsto_{co}$. In the context of the implementation of the cause-effect abstraction through causal delivery of messages, the number of such delayed messages will be greater than necessary[3].

*This work is partially supported by the european project EU-Publi.com funded by the European Community and by the italian projects MAIS and IS-MANET funded by the Italian Ministry of Research.

[1] This comes from the well-known inability of the "happened-before" relation to model true cause-effect relations among events of a distributed computation. These cause-effect relations belong indeed to the semantics sphere of the underlying application (DSM in this paper) rather than the message pattern generated by the distributed computation.

[2] The causal message ordering abstraction states each process has to deliver messages according to the relation $\rightarrow$ of their sendings [3].

[3] A not necessary delayed message corresponds to the phenomenon of

Based on the above points this paper first states an optimality criterion for such protocols. Roughly speaking a protocol P that correctly implements causally consistent DSM is *write delay optimal w.r.t.* $\mapsto_{co}$ if it delays a message only if it is necessary. Then the paper presents a protocol $OptP$ which ensures causal consistent histories while being optimal at the same time with respect to write delay. 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 $\mapsto_{co}$. We also show that previous protocols presented in the literature are not optimal with respect to write delays. This implies that they buffer a number of messages at each process that is greater than necessary.

The rest of this paper is structured as follows: Section 2 presents the shared memory model, Section 3 describes the optimality criterion and shows why protocols appeared in the literature are not optimal w.r.t. write delays. Section 4 presents the protocol $OptP$ along with its correctness proof.

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* memory locations $x_1, x_2, ...x_m$. The memory can be accessed through *read* and *write* operations. A write operation executed by a process p_i, denoted $w_i(x_h)v$, stores a new value v in the location x_h. A read operation executed by a process p_i, denoted $r_i(x_h)v$, returns to p_i the value v stored in the location x_h [4]. Each memory location has an initial value $\perp$.

A *local history* of a process p_i, denoted h_i, is a set of read and write operations. If p_i executes two operations o_1 and o_2 and o_1 is executed first, then o_1 *precedes* o_2 in the *process order of* p_i. This precedence relation is denoted by $o_1 \mapsto_{po_i} o_2$. Operations done by distinct processes are related by the *read-from* relation. Formally read-from relation, denoted $\mapsto_{ro}$, is defined as follows [1]:

- 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.

A *global history* (from now on simply a history) is a partial order $\widehat{H} = (H, \mapsto_{co})$ such that:

- H=$\langle h_1, h_2, \ldots h_n \rangle$, i.e. H is the collection of local histories (one for each process).
- $o_1 \mapsto_{co} o_2$ ($\mapsto_{co}$ is the *causal order* relation) if:
  - $\exists p_i$ s.t. $o_1 \mapsto_{po_i} o_2$ (process order),
  - $\exists p_i, p_j$ s.t. o_1 is issued by p_i, o_2 is issued by p_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 by $o_1 ||_{co} o_2$, 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 $\widehat{H}$ *with respect to* $\mapsto_{co}$, denoted $\downarrow (o, \mapsto_{co})$, as follows:

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

2.1 Causally Consistent Histories

Let us now introduce a few properties of a history [12].

Definition 1 (Legal Read). *Given $\widehat{H} = (H, \mapsto_{co})$, a read event belonging to H, denoted $r(x)v$, is legal if $\exists\, w(x)v$: $w(x)v \mapsto_{co} r(x)v$ and $\nexists w(x)v' : w(x)v \mapsto_{co} w(x)v' \mapsto_{co} r(x)v$.*

Definition 2 (Causally Consistent History [1]). *A* history *$\widehat{H} = (H, \mapsto_{co})$ is causally consistent iff all read operations in $\widehat{H}$ are legal.*

As processes are sequential the definition of causal memory allows 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.

Example 1. Let us consider a system composed by three processes. The following history $\widehat{H}_1$ is causally consistent:

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 $w_1(x_1)c \,||_{co} w_2(x_2)b$, $w_1(x_1)c \,||_{co} w_3(x_2)d$.

a "*false causality*" in an implementation of a distributed shared memory system. The false causality notion has been originally pointed out in the context of distributed predicate detection by Tarafdar and Garg in [15].

[4] 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 executed by process p_i etc.

3 Distributed Shared Memory

3.1 Distributed System Model

The shared memory model of the previous section is implemented through a finite number of sequential processes $\Pi \equiv \{p_1, p_2, \ldots p_n\}$ which communicate using messages that are sent over reliable channels. Each message sent by a process is eventually received exactly once and no spurious message can ever be delivered. There is no bound to the relative process speeds, however, the time taken by a process to execute a computational step is finite.

We assume each process p_i endows a copy of the shared variables $x_1^i, x_2^i, \ldots x_h^i, \ldots x_m^i$. The execution of each operation (read or write) at a process produces a set of events in one or more processes. A history (defined in section 2) produces a sequence of events E_i at each process p_i ordered by the relation $<_i$. $e <_i e'$ means e and e' have happened 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 processes is denoted as $E = \bigcup_{i=1}^{n} E_i$. Such events are ordered by Lamport's "happened before" relation [8], denoted $\rightarrow$, defined as follows: let e and e' be two events of E, $e \rightarrow e'$ iff (i) $e <_i e'$ (ii) e is the sending of a message m and e' is the receipt of m and (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 $\neg(e' \rightarrow e)$. Finally, we denote a distributed computation as $\widehat{E} = (E, \rightarrow)$ and the causal past of an event e as follows:

$$\downarrow (e, \rightarrow) = \{e' \in E | e' \rightarrow e\}$$

.

3.2 A Class of Protocols $\mathcal{P}$ implementing DSM

In this section we point out the common features of a large class of protocols implementing distributed shared memory abstraction. Every protocol P belonging to $\mathcal{P}$ class behaves as follows: each time a process p_i, implementing P, executes a write operation $w_i(x_h)v$, an $apply_k(w_i(x_h)v)$ event is produced at each process p_k $(\forall k \in \{1 \ldots n\})$[5]. Each time a process p_i executes a read operation $r_i(x)v$, p_i eventually produces an event $return_i(x, v)$. Therefore, the operation $w_i(x_h)v$ at p_i is associated with a $send_i(w_i(x_h)v)$ event which can be seen as the starting point of the propagation of $w_i(x_h)v$ in the system. When a process p_j is notified about $w_i(x_h)$, an event $receipt_j(w_i(x_h)v)$ occurs. At this point p_j properly schedules the application of the write to its own copy, i.e. it will produce an event $apply_j(w_i(x_h)v)$. Let us note that, according to the description, we assume that any protocol belonging to $\mathcal{P}$ is live (each operation is eventually executed properly).

[5]Note that the communication mechanism used to propagate the operation from one process to another one (e.g. broadcast, multicast, point-to-point), does not matter at this abstraction level.

3.3 Enabling Event and Write Delaying

Let P denote a protocol belonging to the class $\mathcal{P}$ and e and e' be two events in E. e is an enabling event of e' if the occurrence of e' has to be postponed to the occurrence of e according to *P*. Therefore e is an *enabling event* of e'[6].

In our context, we are interested in characterizing the set of all events which are enabling events of each apply event e in E according to a protocol P. Therefore, we denote $\mathcal{X}_P(e) \subseteq E$ such a set. As a consequence when a process p_k receives the message associated to a write operation w, it postpones the w's application, i.e. $apply_k(w)$ (the message is buffered at p_k), till all enabling events of $apply_k(w)$ will occur. This is abstracted in our model by a write delay which is defined as follows:

Definition 3 (Write Delay). *Let P be a protocol in $\mathcal{P}$, w be a write operation in H, and $e \in \mathcal{X}_P(apply_k(w))$. Then w suffers a write delay at p_k iff $e \notin E_k|_{receipt_k(w)}$.*

3.4 A Protocol $P \in \mathcal{P}$ Compliant w.r.t. $\mapsto_{co}$

Safety. Let P denote a protocol belonging to $\mathcal{P}$. P is *safe* w.r.t. $\mapsto_{co}$ iff write operations are applied at each process according to the order induced by $\mapsto_{co}$. Formally:

$$\forall w_i, w_j \in H, \forall k \in \{1, \ldots n\}, (w_i \mapsto_{co} w_j \Rightarrow$$

$$\forall k \in \{1 \ldots n\}, apply_k(w_i) <_k apply_k(w_j))$$

For each apply event e, the safety property actually defines the set of its enabling events, denoted $\mathcal{X}_{co-safe}(e)$, with respect to $\mapsto_{co}$. Formally:

Definition 4. *$\forall\ e = apply_k(w) \in E$ generated by a protocol $P \in \mathcal{P}$, $\mathcal{X}_{co-safe}(e) \equiv \{apply_k(w') \in E$ s.t. $w' \in \downarrow (w, \mapsto_{co})\}$*

As an example, let us consider the history presented in Example 1 (where $w_1(x_1)a \mapsto_{co} w_2(x_2)b$, $w_2(x_2)b \mapsto_{co} w_3(x_2)d$ and $w_2(x_2)b||_{co} w_1(x_1)c$). In this history $apply_1(w_1(x_1)a)$ and $apply_1(w_2(x_2)b)$ are enabling events of $apply_1(w_3(x_2)d)$ at process p_1. Then

[6]We assume that the event e is an event belonging to E, as any general condition can be easily modelled through a proper event of the computation.

the set $\{apply_1(w_1(x_1)a), apply_1(w_2(x_2)b)\}$ corresponds to $\mathcal{X}_{co-safe}(apply_1(w_3(x_2)d))$.

It comes out that every safe protocol P satisfies the following property:

$$\forall e \in E, \mathcal{X}_{co-safe}(e) \subseteq \mathcal{X}_P(e).$$

At an operational level, a safe protocol $P \in \mathcal{P}$ must delay the application of a write w at process p_k each time an apply event of a write operation in the causal past of w has been not applied at p_k yet. Then in each P's run, if P is safe, it is possible to identify how many write delays have been occurred at each process to maintain safety. For example, let us consider a safe protocol P belonging to $\mathcal{P}$ and the history presented in Example 1. In Table 1 for each apply event the corresponding $\mathcal{X}_{co-safe}$ is described.

event e	$\mathcal{X}_{\mathbf{co-safe}}(\mathbf{e})$
$apply_1(w_1(x_1)a)$	$\emptyset$
$apply_2(w_1(x_1)a)$	$\emptyset$
$apply_3(w_1(x_1)a)$	$\emptyset$
$apply_1(w_1(x_1)c)$	$\{apply_1(w_1(x_1)a)\}$
$apply_2(w_1(x_1)c)$	$\{apply_2(w_1(x_1)a)\}$
$apply_3(w_1(x_1)c)$	$\{apply_3(w_1(x_1)a)\}$
$apply_1(w_2(x_2)b)$	$\{apply_1(w_1(x_1)a)\}$
$apply_2(w_2(x_2)b)$	$\{apply_2(w_1(x_1)a)\}$
$apply_3(w_2(x_2)b)$	$\{apply_3(w_1(x_1)a)\}$
$apply_1(w_3(x_2)d)$	$\{apply_1(w_1(x_1)a), apply_1(w_2(x_2)b)\}$
$apply_2(w_3(x_2)d)$	$\{apply_2(w_1(x_1)a), apply_2(w_2(x_2)b)\}$
$apply_3(w_3(x_2)d)$	$\{apply_3(w_1(x_1)a), apply_3(w_2(x_2)b)\}$

Table 1. $\mathcal{X}_{co-safe}$ of each event generated by a $P \in \mathcal{P}$ producing $\widehat{H}_1$

Figure 1 shows two distinct sequences that could occur at process p_3 during two different P runs and compliant with the history $\widehat{H}_1$ experienced by p_3 in Example 1. In run (1) p_3 does not experience any write delay while in run (2) $apply_3(w_2(x_2)b)$ suffers a write delay due to late arrival of the message associated with $w_1(x_1)$.

3.5 Write Delay Optimality for a Safe Protocol $P \in \mathcal{P}$

Let us consider the case in which $\exists\ e = apply_k(w) :: \mathcal{X}_{co-safe}(e) \subset \mathcal{X}_P(e)$. Clearly P is safe but it is not optimal w.r.t the number of write delays that could occur during a computation. P includes in its $\mathcal{X}_P(e)$, an enabling event e' such that e' is not an apply of a write belonging to w's casual past with respect to $\mapsto_{co}$. This leads, for at least one P's run, to execute *not necessary* write delays. Let us consider the sequence generated by P at p_3 compliant with

(1) $receipt_3(w_1(x_1)a) <_3 apply_3(w_1(x_1)a) <_3$
$receipt_3(w_2(x_2)b) <_3 apply_3(w_2(x_2)b) <_3$
$receipt_3(w_1(x_1)c) <_3 apply_3(w_1(x_1)c) <_3$
$return_3(x_2, b) <_3 apply_3(w_3(x_2)d)$

(2) $receipt_3(w_2(x_2)b) <_3$
$receipt_3(w_1(x_1)a) <_3 apply_3(w_1(x_1)a) <_3$
$apply_3(w_2(x_2)b) <_3 receipt_3(w_1(x_1)c) <_3$
$apply_3(w_1(x_1)c) <_3 return_3(x_2, b) <_3$
$apply_3(w_3(x_2)d)$

Figure 1. Two sequences that could occur at process p_3 compliant with $\widehat{H}_1$.

the history of Example 1 and shown in Figure 2.

$receipt(w_1(x_1)a) <_3 apply(w_1(x_1)a) <_3$
$receipt(w_2(x_2)b) <_3 receipt(w_1(x_1)c) <_3$
$apply(w_1(x_1)c) <_3 apply(w_2(x_2)b) <_3$
$return(x_2)b <_3 apply(w_3(x_2)d)$

Figure 2. A sequence that could occur at process p_3 compliant with $\widehat{H}_1$

We suppose $\mathcal{X}_P(apply_3(w_2(x_2)b)) = \{apply_3(w_1(x_1)a), apply_3(w_1(x_1)c)\}$. Therefore P is safe, however $apply_3(w_1(x_1)c)$ does not belong to $\mathcal{X}_{co-safe}(apply_3(w_2(x_2)b))$. By Definition 3, in this run the number of write delays executed at p_3 is one (i.e., $apply_3(w_2(x_2)b)$ is delayed till $apply_3(w_1(x_1)c)$). Note that this is a non-necessary delay w.r.t. Safety. In this case an optimal (and safe) protocol would not execute any write delay. Formally:

Definition 5. *Let P be a* safe *protocol belonging to $\mathcal{P}$. P is* optimal *on the number of write delays iff:*
$\forall e \in E,\ \mathcal{X}_P(e) \equiv \mathcal{X}_{co-safe}(e)$ *for each protocol run.*

Therefore, protocol P protocol in the above example is not optimal.

3.6 Related Work

ANBKH Protocol. The protocol proposed by Ahamad et al. in [1] (hereafter $ANBKH$) is an example of a protocol belonging to $\mathcal{P}$. In $ANBKH$ propagation of write operation is done through broadcast primitive. To get causal consistent histories $ANBKH$ orders all apply events at each

process according to the happened-before relation of their corresponding send events. In this way all apply events of operations such that $w_i(x)v \mapsto_{co} w_j(x)v'$ will be executed in a causal consistent way with respect to $\rightarrow$ as well. This is obtained by causally ordering message deliveries through a Fidge-Mattern system of vector clocks which considers apply events as relevant ones [4]. Therefore $\forall\ apply_k(w) \in E$, $\mathcal{X}_{ANBKH}(apply_k(w))$ can be defined as follows:

$$\mathcal{X}_{ANBKH}(apply_k(w)) \equiv \{apply_k(w') \in E \text{ s.t. } send(w') \in\downarrow (send(w), \rightarrow)\}.$$

To clarify this point, let us consider the scenario depicted in Figure 3 that can be produced by $ANBKH$. In this scenario $w_1(x_1)a \rightarrow w_2(x_2)b, w_1(x_1)a \rightarrow w_1(x_1)c, w_2(x_2)b \rightarrow w_3(x_2)d$. Since $ANBKH$ enforces casual message deliveries, then the set $\mathcal{X}_{ANBKH}(e)$ for each event e is described in Table 2.

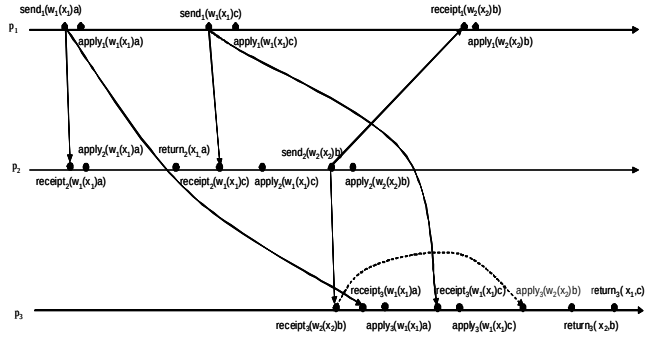

Figure 3. A run of $ANBKH$ compliant with $\widehat{H}_1$

event e	$\mathcal{X}_{\mathbf{ANBKH}}(\mathbf{e})$
$apply_1(w_1(x_1)a)$	$\emptyset$
$apply_2(w_1(x_1)a)$	$\emptyset$
$apply_3(w_1(x_1)a)$	$\emptyset$
$apply_1(w_1(x_1)c)$	$\{apply_1(w_1(x_1)a)\}$
$apply_2(w_1(x_1)c)$	$\{apply_2(w_1(x_1)a)\}$
$apply_3(w_1(x_1)c)$	$\{apply_3(w_1(x_1)a)\}$
$apply_1(w_2(x_2)b)$	$\{apply_1(w_1(x_1)a),\ apply_1(w_1(x_1)c)\}$
$apply_2(w_2(x_2)b)$	$\{apply_2(w_1(x_1)a),\ apply_2(w_1(x_1)c)\}$
$apply_3(w_2(x_2)b)$	$\{apply_3(w_1(x_1)a),\ apply_3(w_1(x_1)c)\}$
$apply_1(w_3(x_2)d)$	$\{apply_1(w_1(x_1)a), apply_1(w_1(x_1)c), apply_1(w_2(x_2)b)\}$
$apply_2(w_3(x_2)d)$	$\{apply_2(w_1(x_1)a), apply_2(w_1(x_1)c), apply_2(w_2(x_2)b)\}$
$apply_3(w_3(x_2)d)$	$\{apply_3(w_1(x_1)a), apply_3(w_1(x_1)c), apply_3(w_2(x_2)b)\}$

Table 2. $\mathcal{X}_{ANBKH}$ of Fig. 3 run's events

$ANBKH$ has been proved to be safe in [1], however by Definition 5, it is not optimal as there exists some event e in a run produced by $ANBKH$ (e.g. $apply_3(w_2(x_2)b$ in the run depicted in Figure 3) such that $\mathcal{X}_{ANBKH}(e) \supset \mathcal{X}_{co-safe}(e)$.[7]

Exploiting writing semantics. A few variants of $ANBKH$ have recently appeared in the literature addressing the *writing semantics* notion [2, 7, 14] introduced by Raynal-Ahamad in [14]. By using this write 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 overwrite w' (it is like the event $apply(w')$ is logically executed by a process immediately before $apply(w)$). This overwriting can happen only if does not exist 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 number of write delays according to the message pattern of the computation. More specifically protocols [2, 14] *apply writing semantics at the receiver side* i.e., when the process receives an overwritten value, it discards the relative message. The protocol proposed in [7] *applies writing semantics at the sender side*. This is done using a token system that allows a process p_i to apply the write operation $w_j(x)v$ only when the local token $t_i = j$ with $i \neq j$ and to send its set of updates only when $t_i = i$. When a process p performs several write operations on the same variable x and then $t_i = i$, it only sends the update message corresponding to the last write operation on x it has executed. This means that the other processes only see the last write of x done by p, missing all previous p's writes on x.

In both cases protocols exploiting writing semantics (either at sender or receiver side) could produce some run where some write operation is not applied by all processes, therefore these protocols do not belong by definition to $\mathcal{P}$.[8]

4 A Protocol ($OptP$) for Causally Consistent Distributed Shared Memory

The protocol presented in this section (hereafter $OptP$) relies on a system of vector clocks, denoted $Write_{co}$, which characterizes $\mapsto_{co}$ [9]. For the sake of simplicity we assume

[7] Figure 3 shows an example of false causality as defined in [15]. In particular, the application of $w_2(x_2)b$ is delayed till the application of $w_1(x_1)c$ *and* $w_1(x_1)a$ in order to respect $\rightarrow$. This delay is non necessary as $w_2(x_2)b$ and $w_1(x_1)c$ have no actual cause-effect relation with respect to $\mapsto_{co}$ even though $send_1(w_1(x_1)c) \rightarrow send_2(w_2(x_2)b)$.

[8] Let us note that the notion of writing semantics is orthogonal wrt the notion of optimal protocols. Therefore, writing semantics could be applied also to the protocol presented in the next section.

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

each write operation is broadcast to all processes. The procedures executed by a process are depicted in Figure 4, 5 and 5. In the following we detail first the data structures and then the protocol behavior.

4.1 Data Structures

Each process p_i manages[10]:

$Apply[1...n]$**:** an array of integer (initially set to zero). The component $Apply[j]$ is the number of write operations issued 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 the k-th write operation issued by process p_j precedes $w_i(x_h)a$ with respect to $\mapsto_{co}$.

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

4.2 Protocol Behavior

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

When a process p_i wants to perform a read operation on x_h, it atomically executes the procedure `read(`x_h`)` depicted in Figure 5.

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

Figure 4. Write procedure performed by p_i

At line 1, p_i incorporates in the local copy of $Write_{co}$ the causal relations contained in the $Write_{co}$ vector associated with 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 a message piggybacking a write operation $w_u(x_h)v$ issued by p_u arrives at p_i, a new thread is spawned. The code of this thread is depicted in Figure 5.

If the condition of line 2 in Figure 5 is verified the thread is executed atomically, otherwise p_i's thread waits until the condition 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 not bring any causal relationship unknown to p_i but the information about itself (i.e. $\forall\ t \neq u \in Write_{co} : w_u(x_h)v.Write_{co}[t] \leq Apply[t]$ 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 applied to p_i. Then the thread is suspended till the application of such a writing at p_i. Once the condition becomes true lines 3 to 5 are executed atomically.

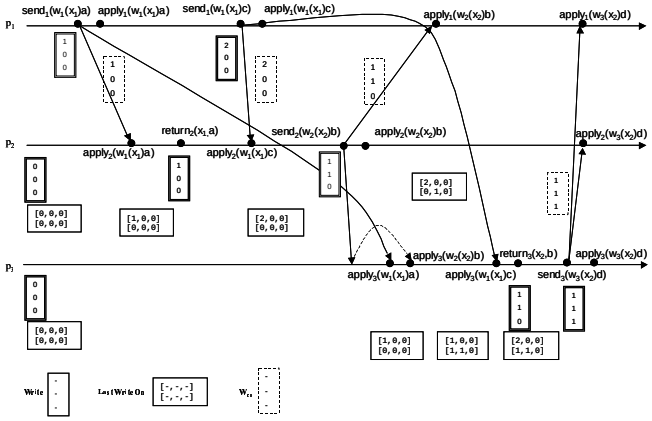

Figure 6. A run of $OptP$ compliant with $\widehat{H}_1$.

Figure 6 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. For the sake of clarity we do not show the evolution of $LastWriteOn$ at process p_1. When p_2 receives the message m notifying $w_1(x_1)a$, 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 corresponding update. Then p_2 executes the operation $r_2(x_1)$ which returns the value a and this establishes a read-from relation between $w_1(x_1)a$ and $r_2(x_1)$. When p_2 executes the broad-

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

```
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 %
```

```
1  Upon the arrival of m(x_h, v, W_co) from p_u                                % receipt event %
2    wait until ((∀t ≠ u ∈ W_co, W_co[t] ≤ Apply[t]) and (Apply[u] = W_co[u] − 1));
3    apply(v, x_h);                                                            % apply event %
4    Apply[u] := Apply[u] + 1;
5    LastWriteOn[h] := W_co;                                                   % storing w_u(x_h)v.Write_co %
```

Figure 5. Read procedure performed by p_i and p_i's synchronization thread

cast to notify the write operation $w_2(x_2)b$, it piggybacks $w_2(x_2)b.Write_{co} = [1, 1, 0]$ on the corresponding message. 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 been already applied at the time p_2 issues $w_2(x_2)b$. This is due to the fact that p_2 does not read the value $x_1 = c$ and thus $w_2(x_2)b||_{co}w_1(x_1)c$. When process p_3 receives the message notifying $w_2(x_2)b$, it cannot apply the corresponding update as there exists a write operation that is in the causal past of $w_2(x_2)b$ and that 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 5 is false. Let us finally remark that p_3 can apply $w_2(x_2)b$ even if it has not already applied $w_1(x_1)c$, because these two write operations are concurrent wrt $\mapsto_{co}$.

4.3 Correctness Proof

In this section we first prove that $Write_{co}$ is a system of vector clocks characterizing $\mapsto_{co}$ (as the classical vector clocks characterizes $\rightarrow$). Then we prove that $OptP$ is *safe*. Finally we show that it is also *write delay optimal*. Finally we prove that $OptP$ is *live*, showing thus that it belongs to $\mathcal{P}$.

Write Causality Graph. Let us introduce the notion of write causality graph that will be used in in the correctness proof of $OptP$. This graph is based on an equivalent formulation of $\mapsto_{co}$

The $\mapsto_{co}$ relation between two write operations w and w', $w \mapsto_{co} w'$, can be also expressed as a sequence of k $\mapsto_{co}$ relations $w \mapsto_{co} w_1 \mapsto_{co} \ldots w_h \mapsto_{co} w_{h+1} \mapsto_{co} \ldots w_{k-1} \mapsto_{co} w'$ (with $k \geq 0$), denoted $w \mapsto^k_{co} w'$, such that for any relation $w_h \mapsto_{co} w_{h+1}$ there not exist a write operation w'' such that $w_h \mapsto_{co} w'' \mapsto_{co} w_{h+1}$.

A write causality graph is a directed acyclic graph whose vertices are all write operations belonging to $\widehat{H}$. There is a direct edge from w to w' if $w \mapsto^0_{co} w'$. 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 7 shows the write causality graph associated with the history $\widehat{H}_1$ 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 $w_2(x_2)b$. Let us finally remark that the notion of causality graph was introduced by Prakash et al. in [13] in the context of causal deliveries in message passing systems.

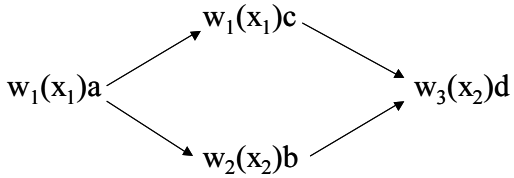

Figure 7. Causality graph of $\widehat{H}_1$

Observations. Let us introduce the following simple observations whose proofs follow directly from the inspection of the code of Section 4.2.

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

Observation 2. *w is the k-th write issued by $p_i \Leftrightarrow w.Write_{co}[i] = k$.*

$\mathbf{Write_{co}}$ characterizes $\mapsto_{co}$. This means that for any pair of writes 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 $w_i \mapsto_{co}^{k} w_j$ introduced above in this section. 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 issued by the same process p_i. Each time a process executes a write operation it performs write procedure in Figure 4. According to line 1 of Figure 4, each time p_i writes, it increments $Write_{co}[i]$. Due to Observation 1, if w_i precedes w_j in p_i process 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 must exist a read operation executed by p_j, denoted $r_j(x_h)$, such that $w_i(x_h) \mapsto_{ro} r_j(x_h)$ and $r_j(x_h) \mapsto_{po} w_j$. Reading the value updated by w_i, p_j has previously set $LastWriteOn[h] := w_i(x_h).Write_{co}$ (line 5 of the synchronization thread Fig. 5). Then when p_j executes line 1 of the read procedure (Figure 5) we have $Write_{co} \geq w_i(x_h).Write_{co}$. Since each time a process p_j writes, it increments $Write_{co}$ and from Observation 1, the next write operation issued 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 induction 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 process p_j has read the value updated by the k-th write operation issued by p_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, p_j has read the value updated by the k-th write operation issued by p_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.

□

Safety.

Theorem 3. *OptP is safe i.e.,* $\forall w_i, w_j \in H, \forall t \in \{1, \ldots n\}, (w_i \mapsto_{co} w_j \Rightarrow apply_t(w_i) \rightarrow apply_t(w_j))$

Proof. The proof uses 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 issued by the same process p_t, then p_t applies them in process order (line 3 of write procedure (fig. 4)). Each other process p can apply w_j only if:

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

Let us suppose that w_i is the $(m) - th$ write issued by p_i and w_j is the $(l) - th$ write issued by p_j. Then from Observation 2 $w_i.Write_{co}[i] = m$ and $w_j.Write_{co}[j] = l$. Two cases:

- $i = j$. 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 $h \geq 1$. The condition (1) can be explained as follows: for $t = j$, $Apply[j] = m + h - 1$. Then p has already been applied the $(m + h - 1) - th$ write operation issued by p_i and all write operations that precede it in p_i process order. As w_i is the $(m) - th$ write operation issued by p_i, before applying w_j, p has applied w_i.

- $i \neq j$. From Corollary 1 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 issued by p_i and all write operations that precede it in p_i process order. As w_i is the $(m) - th$ write operation issued 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 induction 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 process p_k.

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

Write Delay Optimality. Let us first introduce an observation whose proof derives directly from inspection of the protocol of Section 4.2.

Observation 3. *$\forall\ e \in E$, $\mathcal{X}_{OptP}(e)$ only contains apply events of some write operation.*

Lemma 3. *$OptP$ produces runs such that $\forall\ apply(w_i), apply(w_j) \in E$ such that $apply(w_j) \in \mathcal{X}_{OptP}(apply(w_i))$ and $w_i \neq w_j$, we have $w_j \mapsto_{co} w_i$.*

Proof. Let us proof the lemma by the way of contradiction. Let us assume there exists an apply event of a write operation w_i whose enabling event is an apply event of a write operation w_j s.t. $w_j \not\mapsto_{co} w_i$. This implies that if p_k receives w_i without having applied w_j yet, it will delay w_i. Two cases:

- p_k delays w_i and $w_i \mapsto_{co} w_j$. By Theorem 3 if $w_i \mapsto_{co} w_j$ then $apply_k(w_i) \rightarrow apply_k(w_j)$. In this case $apply_k(w_j) \notin E_k|_{receipt(w_i)}$. Then by Definition 3, p_k does not actually delays w_i. Therefore we fall in a contradiction with respect to the initial assumption and the claim follows.

- p_k delays w_i and $w_i||_{co}w_j$. In this case p_k never applies w_i unless w_j has been already applied. Then when p_k receives the message notifying w_i (line 1, Fig. 5), it waits to receive the message related to w_j. This means that the wait condition (line 2) holds because $Apply[j] < w_i.Write_{co}[j]$.[11] Suppose, without loss of generality, that when p_k receives the message notifying w_i, it should be able to apply w_j (from Theorem 3 it means p_k has already applied all writes belonging to w_j's causal past w.r.t $\mapsto_{co}$). Then, supposing that w_j is the $m - th$ write issued by p_j, we have (i) from line 2 of Fig. 5, $Apply[j] = m - 1$ and (ii) applying Corollary 2, $w_i.Write_{co}[j] < m$. Then if p_k can apply w_j because $Apply[j] = m - 1$, it should have applied w_i as well, as $Apply[j] \geq w_i.Write_{co}[j]$. This implies $apply_k(w_i) \rightarrow apply_k(w_j)$. In this case $apply_k(w_j) \notin E_k|_{receipt(w_i)}$. Then by Definition 3, p_k does not actually delays w_i. This contradicts the initial assumption and the claim follows.

□

Theorem 4. *$OptP$ is write delay optimal.*

Proof. By Theorem 3 it follows that $OptP$ is safe. The claim follows from Observation 3 (i.e., all events in all $\mathcal{X}_{OptP}(e)$ are apply events), Lemma 3 and Definition 5. □

Liveness. We show that $OptP$ belongs to $\mathcal{P}$ by showing that $OptP$ is live.

Theorem 5. *All write operations are eventually applied at each process (i.e., $OptP$ belongs to $\mathcal{P}$).*

Proof. Let us assume by the way of contradiction there exists a write operation w_j issued by p_j that can never be applied by p_i. This can happen if $\exists k \neq j \in \{1, \ldots n\} : Apply[k] < w_j.Write_{co}[k]$ or $k = j, Apply][k] < w_j.Write_{co}[k] - 1$. This means that exists at least a write operation w issued by p_k such that $w \mapsto_{co} w_j$, that has not been received at p_i yet. In this case we say that w blocks w_j.

Since communication channels are reliable, each process executes a computational step in a finite time and each operation is broadcast to all the processes, w update message 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.

□

[11] The second part of the wait condition is not considered as the case $i = j$ contradicts the hypothesis of concurrency between the two writes.

References

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

[2] R. Baldoni, C. Spaziani, S. Tucci-Piergiovanni, and D. Tulone. Implementation of causal memories using the writing semantic. In *6th International Conference On Principles Of DIstributed Systems*, pages 43–52, 2002.

[3] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, 1987.

[4] K. P. Birman, A. Schiper, and P. Stephenson. Lightweigt causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, 1991.

[5] C. J. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, 1991.

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

[7] E. Jimenez, A. Fernández, and V. Cholvi. A parametrized algorithm that implements sequential, causal, and cache memory consistency. In *in Brief Announcements of the 15th International Symposium on Distributed Computing*, 2001.

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

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

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

[11] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Proc. of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1988.

[12] J. Misra. Axioms for Memory Access in Asynchronous Hardware Systems. *ACM Transactions on Programming Languages and Systems*, 8(1):142–153, 1986.

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

[14] M. Raynal and M. Singhal. Exploiting Write Semantics in Implementing Partially Replicated Causal Objects. In *Proc. of 6th Euromicro Conference on Parallel and Distributed Systems*, pages 175–164, 1998.

[15] A. Tarafdar and V. Garg. Addressing False Causality while Detecting Predicates in Distributed Programs. In *Proc. 8th International Conference on Distributed Computing Systems*, pages 94–101, 1998.