Roberto Baldoni · Alessia Milani · Sara Tucci Piergiovanni

Optimal Propagation-based Protocols implementing Causal Memories

Received: date / Accepted: date

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 variables 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 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.

Keywords First keyword · Second keyword · More

1 Introduction

Distributed Shared Memory (DSM) is one of the most interesting abstractions providing data-centric communication among a set of application processes which are decoupled in time, space and flow. Consistency conditions define the rules for the ordering of read/write accesses in the shared memory model. For instance, sequential consistency [7] requires that application processes agree on a common order for all read/write operations; atomic consistency [7] (also called linearizability [7]) requires that this order also respects the real-time. Finally, PRAM requires each application process to see the writes by any process in program order [7]. 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 consistency is causal consistency introduced by Ahamad et al. [7]. 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 $o_1 \rightarrow_c o_2$, 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 $o_1$ completes before the issue of $o_2$ (program order relation), (ii) $o_1$ is a write operation on $x$ 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 $o_1 \rightarrow_c o$ and $o \rightarrow_c o_2$. An interesting property of that causality relation is that it allows wait-free read/write operations [7].

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 [7]. 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 variables at each MCS process and propagation of the variable updates [7]. In these protocols, namely Complete Replication and Propagation (CRP) based protocols, a read operation 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 strictly necessary.

In this paper, firstly, we formally define such optimality criterion for CRP protocols. This criterion relies on a predicate, namely the activation predicate, which is associated with each update thread. The predicate becomes true, reactivating the update thread, as soon as that update can be applied at a process without violating causal consistency. Secondly, an optimal CRP protocol is presented. Theoretically, an optimal CRP protocol exploits all the concurrency admitted by the causal consistency criterion. Third, we show that an optimal protocol exhibits a strong reduction in the message buffer overhead at the MCS level compared 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 protocols. This is accomplished through providing 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 the MCS level. In the same section we show that the well-known CRP protocol introduced by Alahmad et al. [?] (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 a main data structure capturing the read/write operation semantics of a causal memory. The paper formally shows that this vector is actually a system of vector clocks characterizing the causality order relation.

Finally, in Section 5 we compare the above said CRP protocols, \( OptP \) and \( ANBKH \), in terms of the buffer size necessary to store update messages at the MCS level. Simulation results clearly show that optimality has a strong impact on such message buffer overhead by minimizing the waiting time of an update message in a buffer due to its activation predicate. Specifically, in the simulated setting \( OptP \) outperforms \( ANBKH \) providing one order of magnitude buffer space saving.

## 2 Shared Memory Model

This model is based on the one proposed by Alahmad et al. [?]. We consider a finite set of sequential application processes \( \{ap_1, ap_2, \ldots, ap_n\} \) interacting via a shared memory \( M \) composed of \( m \) variables \( x_1, x_2, \ldots, x_m \). The memory can be accessed through read and write operations. A write operation invoked by an application process \( ap_i \), denoted \( w_i(x_b) \), stores a new value \( v \) in variable \( x_b \). A read operation invoked by an application process \( ap_i \), denoted \( r_i(x_b) \), returns to \( ap_i \) the value \( v \) stored in variable \( x_b \).

1. A system of vector clocks characterizing Lamport's happened-before relation [?] of a distributed computation has been introduced concurrently and independently by Fidge and Mattern in [?] and [?] respectively.

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

3. It must be noted that the read-from order relation just introduced is the same as the write-into relation defined in [?].
If \( o_1 \) and \( o_2 \) are two operations belonging to \( O_H \), we say that \( o_1 \) and \( o_2 \) are concurrent w.r.t. \( \rightarrow_{co} \), denoted \( o_1 \parallel_{co} o_2 \), if and only if \( \neg(o_1 \rightarrow_{c} o_2) \) and \( \neg(o_2 \rightarrow_{c} o_1) \).

**Properties of a history**

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

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

Let \( H_{i+1}w \) be the history containing all operation in \( h_i \) all write operations of \( H \).

**Definition 2 (Causally Consistent History)** A history \( H \) is causal consistent if for each application process \( a_{H_i} \) there is a serialization \( S_i \) of \( H_{i+1}w \) that respects \( \rightarrow_{co} \).

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

**Example 1.** Let us consider a system composed of three application processes. The following history \( H_1 \) is causal:

\[
\begin{align*}
h_1: & \quad w_1(x_1)a; w_1(x_1)c \\
h_2: & \quad r_2(x_1)a; w_2(x_2)b; r_2(x_2)d \\
h_3: & \quad w_3(x_2)d; r_3(x_2)b
\end{align*}
\]

Note that \( w_1(x_1)a \rightarrow_{co} w_2(x_2)b \), \( w_1(x_1)a \rightarrow_{co} w_1(x_1)c \) while \( w_1(x_1)a \parallel_{co} w_2(x_2)d \), \( w_2(x_2)b \parallel_{co} w_1(x_1)c \), \( w_2(x_2)b \parallel_{co} w_2(x_2)d \) and \( w_1(x_1)c \parallel_{co} w_2(x_2)d \).

**2.2 Distributed Computation at MCS**

Operationally, a 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 \(<_i \). Given two events \( e \) and \( e' \), \( e <_i e' \) means both \( e \) and \( e' \) occurred at \( p_i \) and \( e \) occurred first. We denote as \( E_i[e] \), the prefix of \( E_i \) until \( e \) (not included). The collection of sequences \( E_i \), one for each MCS process, is denoted as \( E = \langle E_1, \ldots, E_m \rangle \).

The events of \( E \) are also ordered by Lamport’s “happened before” relation \([7]\) denoted by \( \rightarrow \) and 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 \( e \parallel e' \), 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 \( \hat{E} = \langle E, \rightarrow \rangle \).

The set of messages sent in a distributed computation \( \hat{E} \) is denoted as \( M_{\hat{E}} \).

**2.3 Complete Replication and Propagation based Protocols**

We assume each MCS process \( p_i \) endows a copy of each variable \( x_h \in M \), denoted \( x_h \). We assume \( p_i \) exchanges messages through a reliable broadcast primitive \([7]\). To send a broadcast message a MCS process invokes the \texttt{RELcast}(m) primitive while the underlying layer of a MCS process invokes the \texttt{RELrcv}(m) primitive which is an upcall used to receive \( m \) by the MCS process.

Any protocol consists of procedures and message handlers. Each procedure/message handler is composed of a finite sequence of statements which can be blocking or non-blocking. A statement execution produces an event. In a complete replication and propagation based (CRP) protocol, procedures implementing read/write operations contain only non-blocking statements and they are atomically executed.

Runs of a CRP protocol generate the following list of events at a process \( p_i \):

- **Message send event.** The execution of \texttt{RELcast}(m) primitive at a process \( p_i \) generates the event \texttt{send}(m).
- **Message receipt event.** \texttt{receipt}(m) corresponds to the receipt of a message \( m \) by \( p_i \) through the execution of the \texttt{RELrcv}(m) primitive.
- **Apply event.** The event \texttt{apply}(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 \( x_h \) at \( p_i \).
- **Return event.** \texttt{return}(x_h, v) corresponds to the return of the value stored in \( p_i \)'s local copy \( x_h \).
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 \( \text{return}_{i}(x,v) \).
- Each time a MCS process \( p_i \) executes a write operation \( w_j(x,v) \), an update message corresponding to \( w_j(x,v) \), denoted as \( m_{w_j(x,v)} \), is dispatched to all other MCS processes through RELcast \( (m_{w_j(x,v)}) \), i.e. \( \text{send}_{i}(m_{w_j(x,v)}) \) is produced.
- Each time a MCS process \( p_i \) receives from the underlying network an update message sent during the execution of a write operation \( w_j(x,v) \), \( p_i \) produces an event \( \text{receive}_{j}(m_{w_j(x,v)}) \) and a new thread is spawned to handle the local application of the update (i.e., the occurrence of the event \( \text{apply}_{j}(w_j(x,v)) \). In this thread, \( p_i \) firstly, tests a local activation predicate \( A(m_{w_j(x,v)}) \) where \( m_{w_j(x,v)} \in \mathcal{M}_p \) and \( e \in E \). That predicate, initially set to \( false \), checks if the update \( m_{w_j(x,v)} \) is ready to be locally applied at \( p_i \) or not, just after the occurrence of the event \( e \). If \( A(m_{w_j(x,v)}) \), \( \text{receive}_{j}(m_{w_j(x,v)}) \) is true, then the \( \text{apply}_{j}(w_j(x,v)) \) 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. If \( A(m_{w_j(x,v)}) \), \( \text{receive}_{j}(m_{w_j(x,v)}) \) is false then the local update of \( x_i \) at \( p_i \) is delayed (by suspending the associated thread). A suspended thread handling \( m_{w_j(x,v)} \) is activated as soon as the predicate \( A(m_{w_j(x,v)}) \) flips to \( true \) and then the apply event is ready to be scheduled by the operating system.

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

Two CRP protocols using a reliable broadcast differ from each other on the definition of the local activation predicate used to control threads handling update messages at a process. Thus, in the following we denote as \( \mathcal{P} = \{ P_1, P_2, \ldots \} \) 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 (safety w.r.t. \( \rightarrow_{\text{ca}} \)). However, as will be seen later, an activation predicate may be stronger than necessary to ensure causal consistency. It can 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.

2.3.1 The \( \rightarrow_{\mathcal{M}} \) relation

The causality order relations among operations invoked at application level have to be preserved at MCS level to assure safety w.r.t. \( \rightarrow_{\text{ca}} \). To this aim, we define a relation denoted as \( \rightarrow_{\mathcal{M}} \) on events generated during a distributed computation by a protocol \( P \in \mathcal{P} \). In particular, the objective of \( \rightarrow_{\mathcal{M}} \) is inducing a (partial) order on the send events of update messages. Formally,

Definition 3 Let \( w(x,a) \) and \( w(y,b) \) be two write operations belonging to \( O_H \) and \( E \) be a computation generated by a protocol \( P \in \mathcal{P} \) executing \( H \). send\(_j(m_{w(x,a)}) \rightarrow_{\mathcal{M}} \) send\(_k(m_{w(y,b)}) \) iff one of the following conditions holds:

1. \( \exists j \in \mathcal{M}_p \cdot \) \( \text{send}_{j}(m_{w(x,a)}) \rightarrow_{\text{ca}} \text{send}_{k}(m_{w(y,b)}) \) and \( j \neq k \).
2. \( \forall j \in \mathcal{M}_p \cdot \text{send}_{j}(m_{w(x,a)}) \rightarrow_{\text{ca}} \text{return}_{i}(x,a) \quad \text{and} \quad \forall j \in \mathcal{M}_p \cdot \text{send}_{j}(m_{w(y,b)}) \rightarrow_{\text{ca}} \text{return}_{i}(y,b) \).
3. \( \forall j \in \mathcal{M}_p \cdot \text{send}_{j}(m_{w(x,a)}) \rightarrow_{\text{ca}} \text{send}_{j}(m_{w(y,b)}) \) and \( \forall j \in \mathcal{M}_p \cdot \text{send}_{j}(m_{w(y,b)}) \rightarrow_{\text{ca}} \text{send}_{j}(m_{w(x,a)}) \).

Two send events \( \text{send}_{j}(m_{w(x,a)}) \) and \( \text{send}_{k}(m_{w(y,b)}) \) are related by \( \rightarrow_{\mathcal{M}} \) then they are also related by the happened before. The converse is not necessarily true. If \( \text{send}_{j}(m_{w(x,a)}) \rightarrow_{\mathcal{M}} \text{send}_{k}(m_{w(y,b)}) \) and \( k \neq j \) but no return event occurs in the run, then \( \text{send}_{j}(m_{w(x,a)}) \rightarrow_{\text{ca}} \text{send}_{k}(m_{w(y,b)}) \). Therefore, the following property holds:

Property 1 Let \( w(x,a) \) and \( w(y,b) \) be two write operations belonging to \( O_H \) and \( E \) be a computation generated by a protocol \( P \in \mathcal{P} \) executing \( H \). We have:

\( \text{send}_{j}(m_{w(x,a)}) \rightarrow_{\mathcal{M}} \text{send}_{k}(m_{w(y,b)}) \) \( \Rightarrow \) \( \text{send}_{j}(m_{w(x,a)}) \rightarrow_{\text{ca}} \text{send}_{k}(m_{w(y,b)}) \)

Relation between \( \rightarrow_{\mathcal{M}} \) and \( \rightarrow_{\text{ca}} \). The relation \( \rightarrow_{\text{ca}} \) induces a partial order on the read/write operations executed at application level. For each protocol \( P \in \mathcal{P} \), the execution by \( a_P \) of a write operation \( w(x,a) \) corresponds to the execution by \( p_k \) of a write procedure generating an event sequence that contains \( \text{send}_{k}(w(x,a)) \). Moreover, the execution by \( a_P \) of a read operation \( r(x,a) \) corresponds to the execution by \( p_k \) of a read procedure generating an event sequence that contains \( \text{return}_{i}(x,a) \). Since, for each protocol \( P \in \mathcal{P} \), any procedure of \( P \) contains only non-blocking statements and it is atomically executed, then there exists a one-to-one mapping between (i) a write executed at application level and the send event of the update message, carrying the value written by that write, generated at MCS level, (ii) a read executed at application level and the return event, returning the value read by that read, generated at MCS level. That one-to-one mapping allows \( \rightarrow_{\mathcal{M}} \) to induce a partial order on send events reflecting the partial order induced by \( \rightarrow_{\text{ca}} \) on operations. Formally, the following property holds:
Property 2 Let $w(x) a$ and $w(y)b$ be two write operations belonging to $O_H$ and $E$ be a computation generated by a protocol $P \in \mathcal{P}$ executing $H$. We have 
\[
    \text{send}_j(m_{w(x)a}) \xrightarrow{\mathcal{E}} \text{send}_k(m_{w(y)b}) \iff w(x)a \rightarrow \epsilon w(y)b
\]

Proof

(\Rightarrow) \quad \text{send}_j(m_{w(x)a}) \xrightarrow{\mathcal{E}} \text{send}_k(m_{w(y)b}) \text{ means that one of the following condition holds:}

1. $\text{send}_j(m_{w(x)a}) \xrightarrow{\mathcal{E}} \text{send}_k(m_{w(y)b})$, $j = k$
2. \text{send}_j(m_{w(x)a})
3. $\exists\text{send}_j(m_{w(x)a})$:

\[
\text{send}_j(m_{w(x)a}) \xrightarrow{\mathcal{E}} \text{send}_k(m_{w(y)b}) \text{ and } j = k.
\]

In this case $p_k$ has generated two events of update message sending, one for the write $w(x)a$ and one for the write $w(y)b$. Since $p_k$ runs a protocol $P \in \mathcal{P}$, it means that $ap_k$ has issued both $w(x)a$ and $w(y)b$. Assuming a write execution as atomic and wait-free (write procedure constituted by non-blocking statements), then $w(x)a$ has been issued and completed before the issue and completion of $w(y)b$. It means that $w(x)a \rightarrow p_{w(x)a} w(y)b$.

Case 2. $\text{send}_j(m_{w(x)a}) \xrightarrow{\mathcal{E}} \text{send}_k(m_{w(y)b})$. In this case $p_j$ has generated an event of update message sending for $w(x)a$. Since $p_j$ runs a protocol $P \in \mathcal{P}$, it means that $ap_j$ has issued $w(x)a$. The same holds for $p_k$, i.e. $ap_k$ has issued $w(y)b$. Moreover, $p_k$ before generating $\text{send}_k(m_{w(y)b})$ has generated a return event returning the value written by $w(x)a$. Since $p_k$ runs a protocol $P \in \mathcal{P}$, it means that $ap_k$ has issued a read operation $r(x)a$. Assuming an operation execution as atomic and wait-free, it means that at $ap_k$ $r(x)a \rightarrow p_{w(x)a} w(y)b$. Moreover, as $ap_k$ reads a value written by a write $w(x)a$ issued by another process $ap_j$, it follows that $w(x)a \rightarrow r(x)a$. Then, by the transitive property of $\rightarrow_{\mathcal{E}}$, $w(x)a \rightarrow_{\mathcal{E}} w(y)b$.

Case 3. This is the transitive closure of $\xrightarrow{\mathcal{E}}$. From the above points it follows that $w(x)a \rightarrow_{\mathcal{E}} w(y)b$.

(\Leftarrow) When program order holds for writes invoked by an application process $ap_k$, the first condition of $\xrightarrow{\mathcal{E}}$ 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 $a$. From the second condition of $\xrightarrow{\mathcal{E}}$ all send events $\text{send}_k(m_{w(y)b})$ generated by $p_k$ after $\text{return}_k(x, a)$ are s.t. $\text{send}_j(m_{w(x)a}) \xrightarrow{\mathcal{E}} \text{send}_k(m_{w(y)b})$. When the transitive closure holds, the third condition of $\xrightarrow{\mathcal{E}}$ holds, as well. Then the claim follows.

2.3.2 Safety

By Property 2, $P$ is safe with respect to $\rightarrow_{\mathcal{E}}$ if and only if the order on local update applications at each MCS process is compliant with the order induced by $\xrightarrow{\mathcal{E}}$ on send events of updates.

Formally:

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

\[
\forall m_w, m_{w'} \in M_{\widehat{E}} : \text{(send}_j(m_w) \xrightarrow{\mathcal{E}} \text{send}_k(m_{w'}) \implies \forall i \in \{1 \ldots n\}, \text{apply}_i(w) < i \text{ apply}_i(w')
\]

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. $\xrightarrow{\mathcal{E}}$.

Then, $P$ allows the application of the delayed update (i.e., $A_p(m_w, e)$ flips to true) only after all $m_{w'}$ preceding updates have been applied. Therefore, $A_p(m_w, e)$ remains false at $p_i$ at least during the time in which there exists a message $m_{w'}$ such that $\text{send}_j(m_{w'}) \xrightarrow{\mathcal{E}} \text{send}_k(m_{w'})$ and $m_{w'}$ has not yet been applied at $p_i$.

2.3.3 Optimality

Informally, a protocol $P$ is optimal if its activation predicate $A_p(m_w, e)$ is false at a process $p_i$ only if there exists an update message $m_{w'}$ such that $\text{send}_j(m_{w'}) \xrightarrow{\mathcal{E}} \text{send}_k(m_{w'})$ and $m_{w'}$ has not yet been applied at $p_i$. Note that optimality does not imply safety. An optimal protocol may apply updates in arrival order regardless of the order imposed by $\xrightarrow{\mathcal{E}}$, however, if it delays the application of an update $m_w$ it does that for a “good reason”, since the message is out of order with respect to $\xrightarrow{\mathcal{E}}$.

An optimal protocol is formally defined as follows:

**Definition 5 (Optimal Protocol)** Let $\widehat{E}$ be a distributed computation generated by $P \in \mathcal{P}$. $P$ is optimal iff

\[
\forall m_w \in M_{\widehat{E}}, \forall e \in E : \text{when receive}_i(m_w) < i, e, \neg A_p(m_w, e) \implies \neg A_{Opt}(m_w, e)
\]

where

\[
A_{Opt}(m_w, e) \equiv \exists m_{w'} \in M_{\widehat{E}} : (\text{send}_j(m_{w'}) \xrightarrow{\mathcal{E}} \text{send}_k(m_{w'}) \text{ and } \land \text{apply}_i(w') \notin E_1 \}
\]

Then, from Property 2, each process running an optimal protocol, delays the application of an update message corresponding to a write $w$ only if there exists at least another update message not yet applied carrying a write $w'$ such that $w' \rightarrow_{\mathcal{E}} w$.
Relation between $A_{Opt}(m_w, e)$ and Safety. From the definition of $A_{Opt}(m_w, e)$, it 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 $p_l$, only after all updates, such that their corresponding send events preceding (for $\Rightarrow$ ) the send event of $m_w$, 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 $A_P \equiv A_{Opt}$.

2.4 $ANBK\bar{H}$ Protocol

In a seminal paper Alamad et al. [?] introduced the notion of causal memory abstraction. In that paper, the authors also proposed a CRP protocol (hereafter referred to by $ANBK\bar{H}$) implementing such an abstraction on top of a message passing system emulating a reliable broadcast primitive. $ANBK\bar{H}$ is actually an instance of the general protocol described in Section 2.3, i.e. $ANBK\bar{H} \in \mathcal{P}$. $ANBK\bar{H}$ 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 [?].

In $ANBK\bar{H}$ the activation predicate $A_{ANBK\bar{H}}(m_{w(y)}, e)$ for each received message $m_{w(y)}$ at each process $p_l$ is the following:

$$\exists m \in M_c, \exists e \in E_l: (A_{ANBK\bar{H}}(m, e) = false \land A_{Opt}(m, e) = true)$$

$ANBK\bar{H}$ 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” 5.

3 An optimal CRP Protocol (Opt $P$)

The CRP protocol presented in this section (hereafter Opt $P$) relies on a system of vector clocks, denoted $Write_{co}$, which recognizes $\rightarrow_{co}$ 6. The procedures executed by a MCS process are depicted in Figure ??, ?? and ?? In the following we detail first the data structures and then the protocol behavior.

3.1 Data Structures

Each MCS process $p_l$ manages7 the following local data structures:

- $Apply_{[1...n]}$: an array of integers (initially set to zero). The component $Apply_{i}$ is the number of write operations sent by $p_j$ and applied at $p_l$.

- $Write_{co} [1...n]$ : an array of integers (initially set to zero). Each write operation $w_i(x_k)$ is associated with a vector $Write_{co}$, denoted $w_i(x_k)$, $Write_{co}$. $Write_{co}[j] \equiv k$ means that $k$-th write operation invoked by the application process $ap_j$ is the last write operation invoked by $ap_j$ preceding $w_i(x_k)$ with respect to $\rightarrow_{co}$.

- $LastWriteOn_{[1...m, 1...n]}$: an array of vectors. The component $LastWriteOn_{i}$ indicates the $Write_{co}$ value of the last write operation on $x_k$ executed at $p_l$. Each component is initialized to $[0, 0, ..., 0]$

References

2. Author, I., Smith, J.: Book Title. Publisher, Place (year)

5 The false causality notion has been first identified by Lamport [?], then it has received more attention by Cheriton and Skeen [?] and by Tarafdar and Garg [?] in the context of causal message ordering and distributed predicate detection respectively.
6 The formal notion of system of vector clocks is given in Section ??.
7 For clarity of exposition, we omit the subscript related to the identifier of process $p_l$ from the data structures.
Fig. 1 A run of ANBKH compliant with the history of Example 1