Università degli Studi di Roma "La Sapienza"

Dottorato di Ricerca in Ingegneria Informatica

XIX Ciclo – 2006

Causal Consistency in Static and Dynamic Distributed Systems

Alessia Milani

Università degli Studi di Roma "La Sapienza"

Dottorato di Ricerca in Ingegneria Informatica

XIX Ciclo - 2006

Alessia Milani

Causal Consistency in Static and Dynamic Distributed Systems

Thesis Committee	Reviewers
Prof. Roberto Baldoni (Co-Advisor)	Prof. Mustaque Ahamad
Prof. Jean-Michel Hélary (Co-Advisor)	Prof. Vijay K. Garg
Prof. Fabrizio D'Amore	

Author's address:
Alessia Milani
Dipartimento di Informatica e Sistemistica
Università degli Studi di Roma "La Sapienza"
Via Salaria 113, I-00198 Roma, Italy
e-mail: `milani@dis.uniroma1.it`
www: `http://www.dis.uniroma1.it/~milani/`

To Corentin, Mum and Vale

Contents

Chapter 1

Introduction

Shared memory is one of the most interesting interprocess communication models among a set of application processes which are decoupled in time, space and flow. Application processes communicate by writing and reading shared variables. There are a lot of problems (in numerical analysis, image or signal processing, to cite just a few) that are easier to solve by using the shared variables paradigm rather than using the message passing one.

Distributed Shared Memory (DSM) emulates shared-memory systems in distributed asynchronous message passing systems. This abstraction provides a strong support for distributed computation, i.e. it allows programmers to design solutions by considering the well-known shared variables programming paradigm, independently of the system (centralized or distributed) that will run their programs. Distributed Shared Memories have 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 [12]. The implementation of MCS enforces a given consistency criterion. A consistency criterion defines the semantics of the memory, that is the value to be returned by any read operation.

Many consistency criteria have been considered, e.g., from more to less constraining ones: Atomic [52], Sequential [51], Causal [7] and PRAM [53]. Less constraining criteria are easier to implement, but, conversely, they offer a more restricted programming model. In detail, atomic and sequential consistency give to processes the illusion to access the memory one at time. On the contrary, causal consistency allow write operations not causally related to be seen by different processes in different order. 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 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 a variable 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 $o \mapsto_{co} o_2$ (*transitivity*).

The causal consistency model has gained interest because it offers a good tradeoff between memory access order constraints and the complexity of the programming model as well as of the complexity of the memory model itself.

Differently from strict criteria, causal consistency allows non-blocking operations, i.e. processes may complete read or write operations without waiting for global computation. Thus it overcomes the major limit of stronger criteria: communication latency. Moreover, several application semantics are precisely captured by causal consistency, e.g. collaborative tools. So, implementing a stricter consistency criterion, not only induces unnecessary complexity to maintain consistency but also reduces the level of concurrency permitted. Weaker semantics may consider acceptable executions that are not assumed to be correct for stronger consistency criteria.

In this thesis we investigate the power and the limits of causal memories to support distributed computation. We provide several results in the context of traditional causal memories.

Moreover, we adapt traditional causal consistency to obtain a memory semantics weak enough to be implemented in in emerging distributed systems paradigms (e.g. peer-to-peer systems) but strong enough to allow computational progress. Such new systems are characterized by nodes continuously joining or leaving. This dynamics induces a continuous change of system membership and imposes to deal with a number of processes that may be infinite as time tends to infinity. According to this, let us notice that in high dynamic systems, stronger consistency criteria may be hard or even not possible to implement, because of possible disconnections or because a big amount of nodes may simultaneously leave the system deliberately or by crashing.

Part of the results presented in this thesis have been published in [15], [16], [18], [41] and [42].

Research Results

Optimal Causal Memory In the context of traditional distributed shared memory implementations, 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 [47]. 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. Ideally, we would like a CRP protocol to apply each update at a MCS process *as soon as* the causal consistency criterion is not violated. In other words, no update thread is kept suspended for a period of time more than strictly necessary.

According to this, in chapter 3 we formally define an 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 MCS process without violating causal consistency. Theoretically, an optimal CRP protocol exploits all the concurrency admitted by the causal consistency criterion. From an operational point of view, an optimal protocol strongly reduces message buffer overhead.

Then we present an optimal CRP protocol built on top of a reliable broadcast communication primitive. Interestingly, we prove that the optimal protocol embeds a system of vector clocks which captures the read/write semantics of a causal memory. Finally, we show how previous protocols based on complete replication presented in the literature are not optimal. We also demonstrate through simulation results that an optimal protocol exhibits a strong reduction in the message buffer overhead at the MCS level compared to a non-optimal one.

The Notions of Share Graph and Hoops Let us consider a set of application processes interacting via a causal consistent shared memory. Since application processes interact by reading and writing shared variables, it is conceivable that the computation at one process will not affect the computation at a different process, if these processes do not access a same shared variable. Unfortunately, this is not the case because of the transitivity property of the causality order relation. From this observation the necessity of a formal model to support inter-processes communication analysis. This last aims at pointing out the connection between variables distribution among the processes in the system and causal ordering relations that may arise between operations generated by such processes on the above said variables. While focusing on the causality order relation, it has been shown that the same reasoning may be applied to analyze other consistency criteria.

In detail, in chapter 2 we introduce the concepts of *share graph* and *hoop* to respectively model variables distribution among processes and their dependencies. Then, we introduce the concept of *dependency chain* with the aim at pointing out causality order relations that may arise between operations issued by different processes on a variable x and due to operations made by different processes on different variables.

Efficient Partial Replication Implementation In the context of traditional distributed shared memory implementation, we formally define necessary and sufficient conditions on MCS processes intended to manage information in order to guarantee causal consistency (chapter 4). This theoretical result leads to several considerations about the efficiency in implementing distributed shared memory supporting partial replication. Notice that minimizing the costs (in terms of communication and information needed) of keeping distributed memories consistent has always being a key problem in building efficient software DSM systems.

Especially, we show the following drawback: in absence of variable distribution knowledge, partial replication implementations require each MCS process to manage information about all shared variables in order to keep the memory causal consistent. On the other hand, if a particular distribution of variables is assumed, while being extremely costly an ad-hoc implementation of causal DSM can be optimally designed.

Write Persistency and Weakly-Persistent object Emerging distributed systems paradigms are characterized by an intrinsic uncertainty due to the continuous joining and departure of nodes. Usually user-driven, such dynamic behavior introduces a complex time-varying nature of nodes' availability. So any process in the system can leave the computation at any time deliberately or by crashing. Since a process usually leaves the system without notifying it, departure are often modelled as crashes. This make not meaningful the notion of correct process in such environments. On the other hand, when defining shared memory semantics in a failure prone model, there is the necessity of considering which write operations we have to assume to take effect.

From what said, the necessity of reasoning at the granularity of operations in order to define memory correctness. To this aim, similarly for what done when introducing the concept of legal read [65], we propose a notion of persistent write.

Let us notice that in distributed systems characterized by high dynamics, ensuring persistency for any written value may be hard or even impossible to implement. So, since some forms of persistency is necessary to ensure computational progress, in chapter 5 we introduce the notion of weak-persistency, i.e. persistency is guaranteed in *quiescent* periods of the systems (arrivals and departures subside). Weak persistency on one hand, is strong enough to rule out trivial implementations and on the other hand, is weak enough to be implemented in these system models. More specifically, weak persistency ensures that in periods in which the system is quiescent, the computation as perceived by application processes (later sometimes referred as client) continually makes progress.

Weakly-Persistent Causal Consistent Distributed Shared Memory
We implement a distributed shared memory over an asynchronous message passing system characterized by (i) infinitely many processes and (ii) high dynamics: processes may join or leave the computation at any time.

In order to implement the memory in this environment, we adopt the client/server paradigm and the related failure model proposed in [26]. More specifically, the memory is implemented by a fixed set of virtual servers. At any time a process incarnates a virtual server. Upon the crash (or the leave) of such a process, a new process replaces the old one in incarnating the virtual server and the old state of the failed process is completely lost. Client processes coordinate the access to the shared variables through servers and no communication among clients is assumed. Then, the set of clients may be infinitely large. Our shared memory implementation enjoys the desirable property of maintaining legal read with respect to the $\mapsto_{co}$ *all the time* regardless of periods affected by high dynamics and of leveraging *quiescent periods* to bring forward a computation perceived in the same way by all clients joining the system along the time.

Thesis Organization

Chapter 2 We define the shared memory model and we revisit the main consistency criteria (sections 2.1, 2.2 and 2.3) and we propose our notion of persistent write (section 2.4). In section 2.4 we introduce the notion of shared graph and hoop. Finally, we present the distributed system models where we investigated our theoretical and practical results (sections 2.5).

Chapter 3 The main result presented in this chapter is the optimality criterion for *Complete Replication and Propagation* based protocols implementing the Memory Consistency System (section 3.4). We also provide an optimal CRP protocol implementing a causal memory and we compare it with existing solutions.

Chapter 4 We characterize MCS processes intended to manage information to implement causal consistency (section 4.2). and we prove the impossibility of efficient partial replication implementations (section 4.3).We then propose some solutions to circumvent such problem, e.g. we investigate new consistency criteria weaker than causal consistency but stronger than Pram (section 4.4). Finally, we give a practical example, i.e. Bellmann-Ford algorithm, to point out the interest in considering both weak consistency criteria and partial replication (section 4.5).

Chapter 5 In this chapter, we point out the difficulty to implement values persistency in high dynamic system and the consequent interest in defining a weak form of persistency, namely weak-persistence. We define a weakly persistent causal memory and we provide its implementation in such high dynamic systems.

Chapter 2

Causal Memory

This chapter first introduces a formal model for shared memory abstraction. Shared memory is one of the main interprocess communication models among application processes decoupled in time, space and flow. Application processes interact by reading and writing to shared memory.

From concurrency in memory accesses the necessity to define the intended correct behavior, i.e. which value a read operation has to return. This is generally stated by the consistency criterion chosen for a given memory. In this chapter, we revisit the main consistency criteria presented in the literature, from more to less constraining ones: Atomic [52], Sequential [51], Causal [7] and PRAM [53]. In particular, this work focuses on causal consistency because it offers a good tradeoff between memory access order constraints and the complexity of the programming model as well as of the complexity of the memory model itself.

Moreover, in order to deal with the complexity of emerging distributed systems paradigms (i.e. peer-to-peer systems) we point out the persistency property usually implicity stated in consistency criteria. Persistency says that written values may be later retrieved if no overwritten. In failure-free memory model, every written value has to be persistent. This is trivially ensured in traditional distributed shared memory implementations [7],[8], [18].

On the contrary, when considering memory models where process may fail, it is necessary to precisely formalize which write operations have to be persistent. This aims to rule out trivial (i.e. not allowing computational progress) or not desired solutions. Ahamad et al. in [6] propose a more general definition of causal consistency in the context of a crash-stop failure model. Roughly speaking their model requests persistency only for write operations "known" by some correct process.

But in new distributed system paradigms not persistent by nature, the concept of correct process is not so clear. Such new systems dynamically change and expand over time imposing to deal with continuous processes joins and departures and with a number of processes that may grow to infinite as time tends to infinity. Each process may leave the computation at any time deliberately or by crashing. Moreover, nodes dynamics are often user-driven.

On the other hand, these systems appear to be promising because of the big amount of computation and storage resources they offer. From what said, the necessity to define some form of memory consistency strong enough to guarantee some computational progress and weak enough to be implemented in such environments.

To this aim, in this chapter we formalize the notion of persistent write operation that we will use in chapter 5, to introduce what we call *weakly-persistent causal memory.*

In the last part of this chapter we introduce some concepts useful to prove the results presented in next chapters. Finally, we provide the distributed system models where we investigate the power and the limits of causal consistency and of existing causal memories solutions.

Part of the results presented in this chapter have been published in [16], [42].

2.1 Memory Model

We consider a set of sequential application processes interacting via a shared memory, $\mathcal{X}$=$\{x_1, x_2, ...x_m\}$, consisting of a finite set of read/write shared objects (later usually referred as variables). Each application process (or simply process when confusion may not arise) is univocally identified by a positive integer, i.e. ap_i will denote the application process whose identity is i. Processes may access the shared memory through read and write operations. A write operation invoked by an application process ap_i on variable x_h, aims at storing a new value v in x_h, denoted $w_i(x_h)v$. A read operation invoked by an application process ap_i, denoted $r_i(x_h)v$, is supposed to return to ap_i the value stored in x_h[1]. Each variable has an initial value $\perp$.

As variables may be concurrently accessed by processes through read and write operations, it is necessary to define which is assumed to be the correct behavior of the shared memory under concurrent operations. In other words,

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

applications processes must be provided with a consistency criterion that precisely defines the semantics of the shared memory, that is the value each read operation has to return. A consistency criterion defines correctness in terms of histories.

2.2 Properties of a History

Different notations have been proposed to model histories and ordering relations in shared memory systems, we derive our formalism by the one introduced by Ahamad et al. in the seminal paper [7].

Each application process ap_i generates a sequence of read and write operations called *local history* and denoted h_i. If an operation o_1 precedes an operation o_2 in h_i, we say that o_1 precedes o_2 in *ap_i program order.* This precedence relation, denoted by $o_1 \mapsto_i o_2$, is a total order. A history H is the union of local histories, one for each application process, i.e. $H = \bigcup_i h_i$. Let us denote as O_H the set of operations belonging to H.

Operations done by distinct application processes can be directly related by the *read-from order* relation. The read-from order relation associates each write operation with the corresponding read operations. Notice that different write operations may store the same value into the same variable. Thus in order to simply associate a write operation with a particular read operation, we adopt, where not differently stated, Misra convention, i.e. all values written by write operations are distinct [60].

Formally, given two operations o_1 and o_2 in O_H, the read-from order relation, $\mapsto_{ro}$, on some history H is any relation with the following properties [7][2]:

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

Given a history H, we denote as H_{i+w} the history containing all operations in h_i and all write operations of H.

[2]It must be noted that the read-from order relation just introduced is the same as the writes-into relation defined in [7].

Definition 2.2.1 (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 on a variable x returns the value written by the most recent preceding 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.

2.3 Causal Consistency in the Hierarchy of Consistency Criteria

Many consistency criteria have been proposed in order to the define shared memory semantics. We consider the main consistency criteria proposed in the literature, e.g. from more to less constraining ones: Atomic [52], Sequential [51], Causal [7] and PRAM [53]. For sake of completeness let us remember that Herlihy et al. in [43] generalize atomic consistency for concurrent shared objects proposing a consistency condition called linearizability; Garg et al. in [36] propose a consistency condition for shared objects weaker than linearizability since it does not refer to real-time constraints and finally, Garg et al. in [61] extend atomic and sequential consistency for multi-object distributed operations (i.e. each operation may span different objects).

Atomic and sequential consistency define correctness of concurrent objects in terms of acceptable sequential behavior, thus giving the illusion to processes to access the object one at time. Roughly speaking, sequential consistency requires that all operations appear to be executed atomically, in some sequential order that is consistent with every process program order. In addition, atomic consistency requests that the real-time relations between operations is also preserved. Formally,

Definition 2.3.1 (Atomic Consistent History) *A* history *H is atomic consistent if there is a serialization S of H that respects i) all program orders $\mapsto_i$ and ii) real-time ordering between operations.*

A memory is atomic if it admits only atomic consistent histories.

Definition 2.3.2 (Sequential Consistent History) *A* history *H is sequential consistent if there is a serialization S of H that respects all program orders $\mapsto_i$.*

A memory is sequential if it admits only sequential consistent histories.

First stated and proved by Lipton and Sandberg in [53] and later formalized through precise timing assumptions by Attiya and Welch [13], the main limit of strong consistency criteria (atomic and sequential) is that no matter how clever or complex a protocol is, if it implements sequential consistency, it must be "slow", i.e. memory access time is dependent on latency due to global actions (e.g. information delivery time). In other words, a memory cannot be sequential and scalable at the same time. To overcome such a problem, Lipton and Sandberg proposed a weaker form of consistency, Pipelined RAM (PRAM). Formally,

Definition 2.3.3 (PRAM Consistent History) *A history H is PRAM consistent if for each application process ap_i there is a serialization S_i of H_{i+w} that respects all program orders $\mapsto_i$.*

A memory is PRAM if it admits only PRAM consistent histories.

Notice that while PRAM, as sequential consistency, requires serializations that respect program orders of every process, it does not request an agreement among processes about the order of all operations belonging to a given history, i.e. write issued by different processes may appear differently ordered in different serializations.

To fill the gap among sequential consistency and PRAM, Ahamad et al. in [7] proposed a consistency criterion stronger than PRAM and weaker than sequential consistency, namely causal consistency. Causal consistency is central in this work. This is mainly due to the fact that causal consistency is the stronger consistency criterion that allow non-blocking operation implementations, i.e. each operation may return independently of global computation. This aspect of causal consistency is essential to guarantee that the system scales well with the number of processes. Moreover, it appears really promising in new system models where implementations of stricter consistency criteria may be hard or even not possible because of high dynamics.

Causal Consistency

Causal consistency requires that serializations respect the causality order relation.

Causality Order Relation Given a history H, the causality order relation induced by $\mapsto_{ro}$ for H, denoted $\mapsto_{co}$, is a partial order that is the transitive closure of the union of the history's program order and the read-from order [7], denoted $\widehat{H}= (O_H, \mapsto_{co})$.

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

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

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

Definition 2.3.4 (Causal Past [63]) *Given a history H, the causal past of an operation $o \in O_H$ is the set of all operations $o' \in O_H$ such that $o' \mapsto_{co} o$.*

Causal Memory Causal consistency allows different processes to perceive concurrent write operations (w.r.t. $\mapsto_{co}$) in a different order. Formally,

Definition 2.3.5 (Causal Consistent History) *A* history *H is causal consistent if for each application process ap_i there is a serialization S_i of H_{i+w} that respects $\mapsto_{co}$.*

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

In the following we consider a system composed of three application processes, ap_1, ap_2 and ap_3 accessing a shared memory $\mathcal{X}=\{x_1, x_2\}$.

Example 1. A history H_1 that is causal but not sequential:

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

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

Example 2. A history H_2 that is PRAM but not causal:

h_1: $w_1(x_1)a$
h_2: $r_2(x_1)a; w_2(x_2)b$

h_3: $r_3(x_2)b; r_3(x_1)\perp$

2.4 Causal Consistency in Presence of Failures

According to the definition of causal consistency given by Ahamad et al. in [7] and reformulated in def. 2.3.5, given a history H each process $ap_i \in H$ has to find a serialization which contains all write operations in H. This means that the effects of each write cannot be indefinitely delayed. So each process has to eventually be able to read any written and not overwritten value.

In a failure free environment, any write operation executed by any process during a run is assumed to appear in the corresponding history. On the contrary, in presence of failures some ambiguities can arise due to failures. Let us for example considering the case of a write operation $w(x)v$ whose process crashed during its execution. It seams reasonable to assume that $w(x)v$ does not belong to the history. However, dependently on implementation details, it could happen that some application process read the value written by $w(x)v$. So $w(x)v$ has to appear in the history otherwise there exists at least a read operation in the history that returns a value never written.

On the other hand, it is not always possible to ensure that any write operation takes effect (e.g. a process executes a write and then it crashes without any other witness process).

From this the necessity to clearly define which write operations are expected to take effect in some finite time. According to this, in the following we first analyze uniform and not-uniform behavior of causal consistency in a crash-stop model. Then we point out the necessity to reason at the granularity of operations instead of processes when dealing with dynamic systems. So, we propose the notion of persistent write, with the aim at defining a variant of causal consistency more suited to such new systems.

Non-uniform Causal Consistency In [6] Ahamad et al. give a more general definition of causal memory that takes into account process failures. They consider crash-stop failure model. So they define a process to be correct if it does never crash. In this failure-prone memory model, their definition of causal memory implies that *all correct processes* have to be able to read values written by write operations that are *known* by *some correct process*. As explained by Ahamad et al. in [6], an operation o is *known* to process p if o is an operation of p or if there exists another operation o' of p such that $o \mapsto_{co} o'$.

From here onwards, we refer to such definition as *non-uniform causal consistency.* It states that each write operation executed by some correct process or whose value has been read by some correct process, has to appear in the history and then in the serialization of every correct process. So it is non-uniform because any correct process is not concerned with operations that are not known by at least a correct process.

In absence of failure or when dealing with non-uniform causal memory, operations may be implemented without blocking. Operations can complete before other processes learn about them.

Uniform Causal Consistency Let us now consider a definition of a causal memory that ensures a uniform behavior: every write operation that is *known* by some process (correct or not), has to appear in the serialization of each correct process. This new definition referred as *uniform causal memory* imposes read and write operations to be implemented in a blocking way. This is due to fact that to implement a uniform behavior, there must exists at least a correct witness process to the written value.

2.5 Causal Consistency and the Notion of Persistent Write

Emerging distributed system paradigms (e.g. peer-to-peer systems) are characterized by an intrinsic unreliability strictly related to their dynamic behavior, i.e. continuous joining and departure of nodes. Moreover, several distributed applications (e.g. file sharing) running on such systems have an user-driven dynamics. In other words, joins and departures distribution is connected to user necessities, introducing a complex time-varying nature of nodes' availability. So, any node may leave the system deliberately or by crashing at any time during the computation. Usually, leaves are treated as crashes. From this, the notion of correct process is not meaningful in such environments. Thus, new system models impose to reason at the granularity of operations.

To this aim we revise the notion of legal read proposed by Raynal et al. in [65] and we introduce the notion of persistent write. This last aims at formalizing the fact that write effects may not be indefinitely delayed and that it has to persist in the system once it takes effect.

Then we will use these notions in chapter 5 to introduce a new memory semantics, namely *weakly-persistent causal memory.* The latter is a variant of causal memory, strong enough to guarantee some computational progress and weak enough to be implemented in a high dynamic system (peer-to-peer systems). Contrarily to traditional causal memory definition (see def. 2.3.5), weakly-persistent causal memory definition does not rely on operations serializations.

Legal read The legality property of a read operation explains which is the intended correct value a read operation has to return *in presence of concurrency.* A read is legal if it returns a value previously written and not overwritten according to some logical or temporal ordering relation. In the following we implement the notion of legal read proposed by Raynal et. al in [65] in the context of causal consistency.

Formally,

Definition 2.5.1 (Legal Read.[65]) *Given a history H, if it exists a read operation $r(x)v$ belonging to H, then i) there must exist a write operation $w(x)v \in H$ such that $w(x)v \mapsto_{co} r(x)v$ and ii) there must not exist an operation $o(x)v' \in H$ such that $w(x)v \mapsto_{co} o(x)v'$ and $o(x)v' \mapsto_{co} r(x)v$.*

Persistent write Persistency is necessary to guarantee communication among asynchronous processes (i.e. processes access the shared object at any time without any form of a priori synchronization) and then for computational progress. Persistency states that *in absence of concurrent or successive* write operations, a written value has to be eventually returned by a subsequent read operation.

In other words, a given write operation $w(x)v$ is said to be *persistent* if in absence of successive and concurrent write operations, any application process that reads infinitely many times x, will eventually read v forever. In this context, the notions of *successive*, and *concurrently* written are w.r.t. the causality order relation $\mapsto_{co}$.

Formally,

Definition 2.5.2 (Persistent write) *Given a history H, if $\exists$ a write operation $w(x)v \in H$ such that both the following conditions hold:*

1. *$\nexists$ $w(x)v'$ belonging to H s.t. $w(x)v \mapsto_{co} w(x)v'$ or $w(x)v ||_{co} w(x)v'$*
2. *$\exists$ i such that H_{i+w} contains infinitely many read operations $r(x)u$*

then infinitely many $r(x)u$ are such that $u = v$.

Let us remark that the notion of persistent write may be presented in a more general way (as done for the notion of legal read by Raynal et al. in [65]) in order to be useful for the main consistency criteria (atomic, sequential, causal and PRAM).

Causal Consistency in terms of Legality and Persistency. In the following, we prove that given a causal consistent history H, every read $\in H$ is legal and every write $\in H$ is persistent.

Formally,

Lemma 2.5.1 *Given a history H, if H is causal consistent then each read operation $\in H_{i+w}$ is legal for any $ap_i \in H$.*

Proof We consider a given history H. We assume H be causal consistent and we prove that any read operation in H_{i+w} is legal. As H is a causal consistent history by definition 2.3.5, for each process $ap_i \in H$ there exists a serialization S_i of H_{i+w} that respects $\mapsto_{co}$. Moreover, according to definition 2.2.1, each read operation $r(x)$ in S_i returns the value written by the most recent precedent write on x in S_i. Let $r(x)v$ be a read operation in H_{i+w}. For definition 2.2.1 there exists a write operation $w(x)v$ that appears before $r(x)v$ in S_i. Moreover because of the definition of $\mapsto_{ro}$ and of $\mapsto_{co}$ and the fact that we assume that all values written by write operations are distinct [60], we have that given $r(x)v$, $\exists w(x)v \in H$ such that $w(x)v \mapsto_{co} r(x)v$. Thus, we have proved point i) of definition 2.5.1.

Now let us prove point ii) of definition 2.5.1. Assume by contradiction that $\exists$ an operation $o(x)u \in H_{i+w} : (u \neq v) \wedge (w(x)v \mapsto_{co} o(x)u \mapsto_{co} r(x)v)$. As a read $r(x)$ returns the value written by the last write operation that precedes $r(x)$ in S_i, it follows that $o(x)u$ precedes $w(x)v$ in S_i. Consequently, as the ordering S_i respects the relation $\mapsto_{co}$, we have $(o(x)u \mapsto_{co} w(x)v) \vee (op(x)u || w(x)v)$. This leads to a contradiction.

Lemma 2.5.2 *Given a history H, if H is causal consistent then each write operation $\in H_{i+w}$ is persistent.*

Proof We prove that given a history H if H is causal consistent then there could not exist a write operation $w(x)v$ such that $w(x)v$ is not persistent. By contradiction let us assume that there exists a write operation $w(x)v$ in H, such that $w(x)v$ is not persistent. According to definition 2.5.2, as we consider a write operation $w(x)v'$ belonging to H, we have $w(x)v' \mapsto_{co} w(x)v$. Moreover there exists an infinite number of read operations $r_i(x)$ issued by a process $ap_i \in H$ such that each $r_i(x)$ returns a value u with $u \neq v$. H_{i+w} contains infinitely many read operation on variable x. By definition of causal consistent history (see def. 2.3.5), for each process ap_i there exists a serialization S_i of H_{i+w} that respect $\mapsto_{co}$. So, let α be the rank in S_i of the last write operation on x, namely $w_{last}(x)v_{last}$. Since each write operation may not indefinitely be delayed α is well defined. On another hand, it follows from the fact that ap_i reads x infinitely many times that there exist infinitely many read operations $r_i(x)$ that are ordered after $w_{last}(x)$ (i.e., the rank of such a read operation in S_i is greater than α.). Because of definition 2.2.1, the value returned by a read operation $r_i(x)$ has to be the last value written accordingly to S_i. So since $w_{last}(x)$ is the last write operation in S_i every read operation $r_i(x)$ ordered after $w_{last}(x)$ has to return v_{last}. It consequently follows that process ap_i eventually read the value v_{last} forever. This means that the write $w_{last}(x)$ is persistent. Finally, definition 2.3.5 implies that S_i has to respect $\mapsto_{co}$. So because, any write operation $w(x)v'$ belonging to H is such that $w(x)v' \mapsto_{co} w(x)v$, we have that $w(x)_{last}$ must be equal to $w(x)v$. So we arrive to a contradiction. Since this reasoning may be applied to any write operation in H_{i+w} and to any H_{i+w}, we have proved the lemma.

Theorem 2.5.3 *Given a causal consistent history H, every read operation $\in H_{i+w}$ is legal and every write operation $\in H_{i+w}$ is persistent, $\forall ap_i \in H$.*

Proof Proved by lemma 2.5.1 and lemma 2.5.2.

2.5.1 Sources of Non-Persistent Behavior

By definition 2.5.2, persistency predicates on object status as perceived by the application processes that access it. In other words, stating that after a write operation $w(x)v$ a process has to eventually be able to read v is stronger than assuming object status to be v. According to this, let us explain possible causes of non persistency.

As introduced by Afek et al. in [1], the more general fault for a memory object is a spontaneous change in its value. As a particular case, if the object spontaneously changes its value into the initial one, written value are not persistent (definition 2.5.2). So an object that suffers the above said fault is not persistent. On the other hand, the viceversa is not always true: an object may be not persistent because of an incorrect implementation of read and write operations. As a trivial example, consider a protocol that implements a read/write object x in such way: any time a write operation of a value v is issued on x, the corresponding value is stored into x; any time a process invokes a read operation on x, the initial value is returned independently of object real status. It is simple to understand, that the proposed implementation does not allow reads to return any written value into x. So x is not persistent despite the fact that the object does never spontaneously change its value. In the following section, we deeply analyze our definition of persistency relating it with existing works on faulty memory objects.

2.5.2 Related Work

Attiya et al. in [11] give the definition of persistency for a single writer/multi-readers atomic read/write object(register), that is: “once a process reads a particular value, then, unless the value of this register is changed by a write, every future read of this register may retrieve such a value, regardless of process slow-down or failure”. Herlihy et al. in [43] formalize the concept of persistency for a multi-writer/multi-reader atomic object. In detail, modelling each operation as a pair of events (invocation and response), they state that “ if I is an interval (sequence of contiguous events) that does not overlap any write operations, then all read operations that lie within I return the same value”,[43]. Obviously, the above said definitions are not compliant for an object where write operations can take effect arbitrarily slowly. According to this, we introduce a more general definition 2.5.2, which captures the notion of persistent write without referring to any particular consistency semantics.

Moreover, looking at the lack of persistency as defined in 2.5.2, as a kind of failure a read/write shared object may suffer, we can relate our work to the one proposed by Afek et al. in the seminal paper [1] and the one presented by Jayanti et al. in [44]. In detail, Afek et al. investigate the effect of shared memory failures in distributed systems. They explicitly defines the problem of faulty shared objects and they show how to construct fault-tolerant ones. They consider several memory objects such as safe, regular and atomic read/write shared objects and more powerful ones such as test-and-set, read-write-modify etc. The same problem has been independently study by Jayanti et al. in [44].

Afek et al. in [1] recognize a spontaneous change in its value as the most general fault a shared object may suffer. As explained in section 2.5.1, this is a possible cause of non persistency. Moreover, even the less general fault considered by Afek et al., namely occasional miss of a write (a written value is never available to read), does not match the fact that an object is not persistent. In detail, a non persistent object may allow some read operations to return a written value while a miss write $w(x)v$ implies that no read may return the corresponding value v. Notice that, a non persistent object could be more dangerous than an object that misses some writes when a uniform behavior is desirable.

When considering the omission failure proposed by Jayanti et al. in their classification of responsive failures, we can find some analogies with the lack of persistency. In detail, they denote as $\perp$ a special value that allows a process to discover object failure. So "when an object fails by omission, it returns normal responses to some operation and $\perp$ to others, and satisfies the following property: the object would seem non-faulty if every operation that obtained the response $\perp$ were treated like an incomplete operation that never obtained a response." After the write of a value v and in absence of other write operations, a non persistent object could return to some read operations value v while returning $\perp$ to other ones, where $\perp$ is the object initial value. Differently from omission failure, a process accessing the object will not perceive that the object is faulty if legality is not violated.

Finally, let us observe that, while considering atomic objects, in their works Afek et al. and Jayanti et al. do not address sequential, causal and PRAM consistent read/write shared object.

2.6 Modelling Causality Order and Inter Processes Communication

Let us consider a set of application processes Π interacting via a given shared memory $\mathcal{X}$. Given any two processes ap_i and ap_j belonging to Π, we aim at understanding which may be the impact of ap_i computation on ap_j computation and viceversa. In detail, we want to avoid misunderstandings that may arise in scenarios where each application process ap_i accesses only a subset of shared variables $\mathcal{X}$, denoted $\mathcal{X}_i$. Let us explain. Application processes interact only via shared variables. So, given two different processes ap_i and ap_j such that $\mathcal{X}_i \bigcap \mathcal{X}_j = \emptyset$, it is conceivable that the computation at ap_i will not affect the one at ap_j and viceversa. Unfortunately, this is not the case if $\mathcal{X}$ has to be causal consistent. Due to the transitivity property of the causality order relation, an operation made by process ap_i on a variable $x_h \in \mathcal{X}_i$ may influence the result of an operation made by ap_j on a variable $x_k \in \mathcal{X}_j$.

In this section, we first introduce the notions of *share graph* and *hoop* to capture variable distribution among processes and their dependencies. Then we introduce the concept of *dependency chain* with the aim at pointing out causality order relations that may arise between operations made by different processes on a variable x and due to operations made by different processes on different variables. We will use such formalism in chapter 4 to investigate the cost of maintaining causal consistency in a partially replicated environment.

For sake of clarity, let us first visualize the partial order induced by the causality order relation as a graph, [35], that we call *causal graph.* Notice that Baldoni et al. in [19] analogously introduce the causality graph to model causality order relations in shared memory model. However, differently from this, we consider all operations belonging to a given history H, that is both read and write.

Causal Graph Given a finite history H, the irreflexive partial order induced by the causality relation $\mapsto_{co}$ on H, may be represented through an acyclic directed graph , denoted CG and that we call causal graph. Formally,

Definition 2.6.1 *Given a history H, a causal graph on H, denoted $CG = (V, E)$, is an acyclic directed graph such that:*

- *there exists a vertex $o \in V$, for each operation $o \in O_H$ and*
- *there exists a directed edge between two vertices o_1 and o_2 if $o_1 \mapsto_{co} o_2$ and $\nexists\ o_3 \in O_H$ such that $o_1 \mapsto_{co} o_3$ and $o_3 \mapsto_{co} o_2$.*

Figure 2.1 depicts the causal graph corresponding to history H_1 of example 1. Our way to depict the graph is simply to point out program order (vertical arrows) and read-from order relations (diagonal arrows).

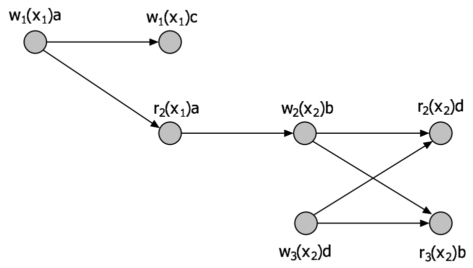


Figure 2.1: Causal graph of History H_1

2.6.1 The Notion of Share Graph

The share graph is an undirected (symmetric) graph whose vertices are *application processes*, and an edge (i, j) exists between ap_i and ap_j iff there exists a variable x accessed by ap_i and ap_j (i.e. $x \in \mathcal{X}_i \cap \mathcal{X}_j$). Formally,

Definition 2.6.2 *Given a set of application processes Π and a set of shared variables $\mathcal{X}$, the share graph is an undirected (symmetric) graph $SG = (V, E)$ such that the following conditions arise:*

- ❖ *there exists a vertex ap_i in V for each* application process $ap_i \in \Pi$ *and*
- ❖ *for each pair of vertices ap_i and $ap_j \in V$, there exists an edge (i, j) between ap_i and ap_j iff there exists at least a variable x such that $x \in \mathcal{X}_i \cap \mathcal{X}_j$.*

Possibly, each edge (i, j) is labelled with the set of variables shared by ap_i and ap_j.

Figure 2.2 depicts an example of share graph representing a system of three processes ap_i, ap_j and ap_k interacting through the following set of shared variables $\mathcal{X} = \{x_1, x_2\}$. In particular, $\mathcal{X}_i = \{x_1,\ x_2\}$, $\mathcal{X}_k = \{x_2\}$ and $\mathcal{X}_j = \{x_1\}$.

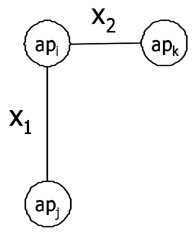


Figure 2.2: A share graph

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

In the example depicted in Figure 2.2, we have the following cliques:
i) $C(x_1) = (V_{x_1}, E_{x_1})$ where $V_{x_1} = \{ap_i,\ ap_j\}$ and $E_{x_1} = \{(i, j)\}$,
ii) $C(x_2) = (V_{x_2}, E_{x_2})$ where $V_{x_2} = \{ap_i,\ ap_k\}$ and $E_{x_2} = \{(i, k)\}$.

Given a variable x, we call *x-hoop*, any path of SG, between two distinct processes in $C(x)$, whose intermediate vertices do not belong to $C(x)$ (figure 2.3). Formally,

Definition 2.6.3 (Hoop) *Given a variable x and two processes ap_a and ap_b in $C(x)$, we say that there is a x-hoop between ap_a and ap_b (or simply* a hoop, *if no confusion arises), if there exists a path $[ap_a = ap_0, ap_1, \dots, ap_k = ap_b]$ in SG such that:*
i) $ap_h \notin C(x)$ $(1 \le h \le k-1)$ and
ii) each consecutive pair (ap_{h-1}, ap_h) shares a variable x_h such that $x_h \neq x$ $(1 \le h \le k)$

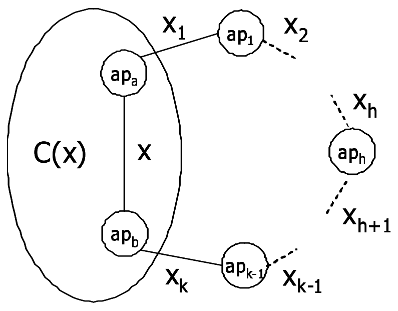


Figure 2.3: An x-hoop

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

Definition 2.6.4 (Minimal Hoop) *An x-hoop* $[ap_a = ap_0, ap_1, \ldots, ap_k = p_b]$ *is said to be* minimal, *iff i) each edge of the hoop is labelled with a different variable and ii) none of the edge label is shared by processes* ap_a *and* ap_b*.*

2.6.2 Hoops and Dependency Chain

The following concept of dependency chain aims at capturing the causality order relations that may arise between operations performed by processes belonging to an hoop.

Definition 2.6.5 (Dependency chain) *Let* $[p_a, \ldots, p_b]$ *an x-hoop in a share graph SG and H be a history where CG is the corresponding causality graph. We say that H includes an x-dependency chain along this hoop if there exist two operations* $o_a(x)$ *and* $o_b(x) \in O_H$ *such that the two following conditions hold:*

1. $o_a = w_a(x)v$;
2. $\exists$ *a path* $[o_a(x), \ldots, o_b(x)]$ *in CG that contains at least one operation for each* ap_i *belonging to the hoop.*

We also say that $o_a(x)$ and $o_b(x)$ are the *initial* and the *final* operations of the x-dependency chain from $w_a(x)v$ to $o_b(x)$. Notice that by definition 2.6.1, such dependency chain implies that $o_a(x) \mapsto_{co} o_b(x)$. So, since $o_b(x)$ may be both a read or a write operation, to not violate causal consistency we respectively have to guarantee that i) if $o_b(x) = r_b(x)$ then it has to return a

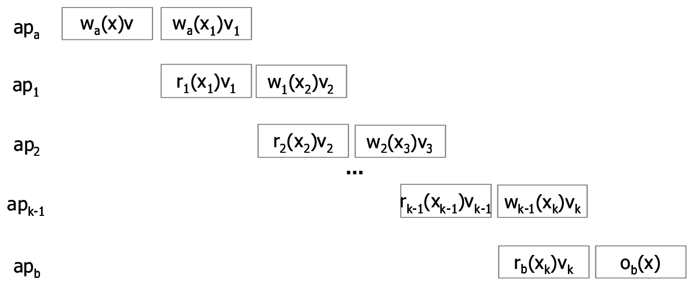


Figure 2.4: A history H containing an x-dependency chain from $w_a(x)v$ to $o_b(x)$

value that is not causally precedent to v and ii) if $o_b(x) = w_b(x)v'$ then any application process that reads both value v and v', has to read v and then v'.

Figure 2.4 depicts a history H containing an x-dependency chain from $w_a(x)v$ to $o_b(x)$. Let us finally remark that while the causality order relation is between two operations performed on variable x, namely $w_a(x)v$ and $o_b(x)$, it has arisen because of operations performed by other processes on variables $x_h \neq x$.

2.7 Static and Dynamic Distributed Systems

In this work, we study causal memory implementations provided that application processes are running in a distributed system consisting in a set of nodes interconnected by a communication network. In particular, we consider two main system models, respectively a static fault-free distributed system characterized by a fixed set of processes (section 2.7.1) and a dynamic one, in which there may be an infinite number of processes joining and leaving the system deliberately or by crashing (section 2.7.2). In both cases, the system is asynchronous, i.e. message transfer delay is unpredictable but is finite and there is no bound on the relative process speeds, despite the fact that the time taken by a process to execute a computational step is finite.

2.7.1 Static System Model

The shared memory abstraction is traditionally implemented by a *memory consistency system*(MCS) on top of a message passing system[12]. Roughly speaking the *memory consistency system* provides shared memory emulation by using local memory at each node and some protocol executed by the MCS processes.

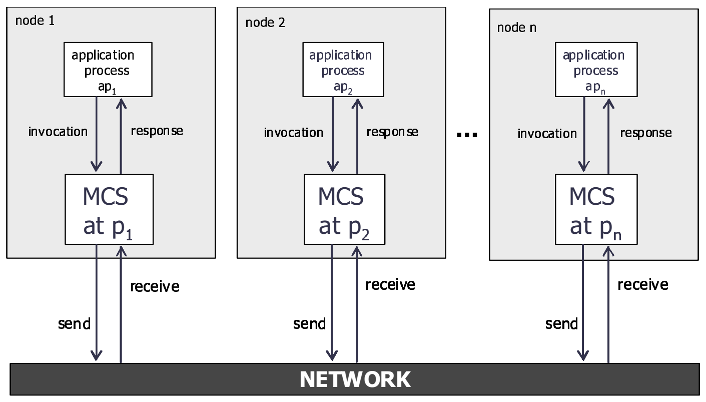


Figure 2.5: Static system architecture

On each node i there is an application process ap_i and a MCS process p_i [12], as depicted in Figure 2.5. An application process ap_i invokes an operation through its local MCS process p_i which is in charge of the actual execution of the operation. From now onwards we shall usually refer to MCS process as simply a process.

Replication towards concurrency Most protocols run by the MCS protocol to implement causal memory support replication of variables in order to exploit concurrency permitted by causal consistency. Replicating variables in the local memory of nodes allows processes to simultaneously access a same variable. On the other hand, replication introduces complexity in consistency management, i.e. MCS system must guarantee consistency of value read despite of concurrent variable updates. According to this, protocols run by MCS processes may be classified in invalidation protocols (outdated replicas are invalidated) or propagation-based protocols (written values are propagated in the system)[21], [29].

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. Events belonging to E_i are ordered by the relation $<_i$. In detail, $e <_i e'$ means that 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 $E = \langle E_1, ...E_m \rangle$. Events belonging to E are ordered by Lamport's "happened before" relation [50] denoted by $\rightarrow$. Formally, given two events e and e' belonging to E, $e \rightarrow e'$ iff one of the following conditions holds: (i) e

$<_i e'$; (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 $e'' \rightarrow e'$. Then, 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 $M_{\widehat{E}}$.

2.7.2 Dynamic System Model

We consider the *infinite arrival* model proposed by Aguilera et al. in [3]: the system consists of possibly infinitely many processes, runs can have infinitely many processes, but in each time interval only finitely many processes take steps.

In order to implement objects in this environment, we adopt the client/server paradigm and the related failure model proposed in [26]. In detail, as depicted in Figure 2.6, components of the system are logically separated into: `client processes`[3], `object entities` and `object manager processes`.

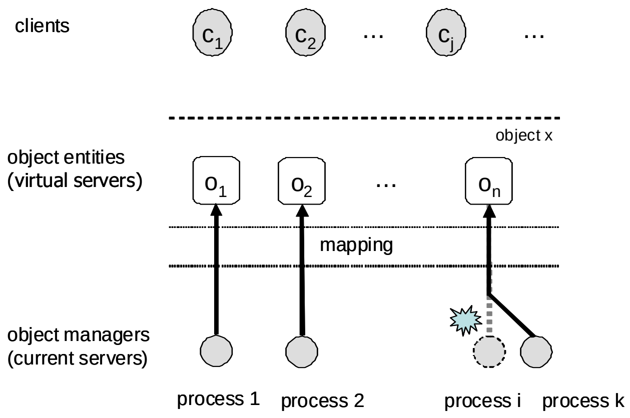


Figure 2.6: Dynamic system architecture

Each object x is implemented by a finite number n of virtual servers, also called `object entities` $\{o_1, o_2, \ldots, o_n\}$. Each `object entity` is characterized by an univocal virtual identifier and a state. In particular, o_j denotes the $j-th$ `object entity` and its state is its current value.

Each object entity o_i is implemented by an `object manager process` which is in charge of the actual execution of read/write operations invoked

[3]We refer to client process, both to refer to the application process that invokes the operations and to the client side process that together with object manager processes, is in charge of the actual execution of such operations.

by `client processes`. An object manager process is identified by the identity of the object entity it is in charge of. Since at each time, each object entity is incarnated by a single object manager process, sometimes we denote as o_i both the object entity and the corresponding object manager process.

Failure Model

A process (client or object manager) may crash, that is, it halts prematurely. A crashed process does not recover. This means that, from a practical point of view, a process that crashes, can re-enter the system with a new identity. A process that does not crash is correct otherwise it is faulty.

We treat the deliberate leave of an object manager as a crash. If an `object manager` leaves the system, deliberately or by crashing, if a new `object manager` will replace that previous one it will assume the same virtual identity. As an example, in Figure 2.6, the process i crashes and it is replaced by process k. Moreover, the new object manager process is not able to retrieve any state the crashed process passed through during its execution. According to this, processes crashes are associated with object entity *memory losses*, i.e. the object entity returns to its initial status. To model possible infinite alternation of peers incarnating an object entity, these losses can be an infinite large number. This dynamic system model nicely captures for example the basic behavior of structured P2P systems, [67], [71], [75].

We assume that each time an object manager process leaves the system, there exists a new one that replaces the previous one. For what said, each object entity o_i is characterized by a sequence of object managers, denoted $\widehat{o_i}$. Let us remark that the mapping between object entities and object manager processes can be realized through well-known technologies such as Domain Name Server (DNS), Distributed Hash Table (DHT) etc. This technologies include mechanisms providing a good support for maintaining a stable set of server processes.

Comparison with the Crash-Recovery Model The failure model we consider is strictly related to the crash-recovery model where no stable storage is provided. The crash of an object manager process and the corresponding process replacement may be seen as the object entity incarnated by such processes, crashes and then recoveries without being able to retrieve any state it passed through. Notice that in order to rule out undesirable process behavior, in a crash recovery model it is usually assumed that the process is aware it has crashed and recovered. We do not make such assumption.

Communication Model

Processes communicate exchanging messages over *fair-loss point-to-point channels* [54]. Fair-loss channels abstract the natural unreliability of dynamic networks where nodes can join, leave or crash at any time. They capture the possibility of messages to be lost when transiting in such networks while ensuring a non zero probability for a message to be received. As an example in structured peer-to-peer overlay networks (e.g., [71], [75]) messages are routed in a small number of hops along the overlay. When nodes join or leave, the overlay automatically reconfigures. But messages may be incorrectly routed or lost because a large fraction of nodes leaves (possibly simultaneously) or crash or the network partitions.

Chapter 3

Optimal Propagation-based Protocols implementing Causal Memories

In this chapter, in the context of traditional distributed systems (section 2.7.1) we define optimality for distributed causal memory implementations. Especially, we first formally characterize Complete Replication and Propagation based protocols (CRP), then we provide an optimality criterion for such class of protocols. Finally, we present an optimal protocol to implement a distributed causal consistent shared memory and we point out the non-optimality of previously proposed protocols belonging to the given class.

Part of the results presented in this chapter have been published in [15] and [18].

3.1 Complete Replication and Propagation based Protocols

We consider traditional distributed systems implementations 2.7.1. We assume complete replication of variables at each MCS process p_i. This means that each process p_i endows a copy of each variable $x_h \in \mathcal{X}$, denoted x_h^i. We also assume that the protocol run by MCS processes is a propagation-based protocol. Roughly speaking, this means that every time an application process ap_i issues a write operation, the corresponding MCS process p_i updates the local copy of the memory and propagates such write by sending update messages to all other MCS processes in the system. On the other hand, every time an application process ap_i issues a read operation on a variable x_h, the corresponding MCS process p_i simply returns to ap_i the value locally stored in x_h^i.

We assume MCS processes communicate by exchanging messages through a reliable broadcast primitive [40]. 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 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 protocol (CRP), 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 `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.

Operations towards Events 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 message corresponding to $w_i(x_h)v$, denoted as $m_{w_i(x_h)v}$, is dispatched to all other MCS processes through `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 message 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 message handler thread is spawned to handle the local application of the update (i.e., the occurrence of the event $apply_i(w_j(x_h)v)$.

Activation predicate When an MCS process p_i receives an update message, it chooses which is the appropriate instant to locally apply the update, as long as the resulting computation is causal consistent. From an operational point of view, upon the receipt of an update message $w_j(x_h)v$, a message handler thread is spawned at p_i to test a local activation predicate [1] aiming at maintaining causal consistency, denoted $A(m_{w_j(x_h)v}, e)$ where $m_{w_j(x_h)v} \in M_{\widehat{E}}$ and $e \in E$.

That predicate, initially set to *false*, checks 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. 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 (by suspending the associated thread). A suspended thread handling $m_{w_j(x_h)v}$ is activated as soon as the predicate $A(m_{w_j(x_h)v}, e)$ 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., **wait until** $(A(m_{w_j(x_h)v}, e))$. If a thread is suspended at p_i, it will spin on the local activation predicate $A(m_{w_j(x_h)v}, e)$ 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 $\mathcal{P} = \{P, P', ...\}$ 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 (*safety* w.r.t. $\mapsto_{co}$). However, as will be seen later, an activation predicate may be stronger than necessary to ensure causal consistency, i.e. it can suspend a thread for a time longer than necessary. In this case we say that the protocol is not optimal. To formally state the notions of safety and optimality, we need to map causality order relations at application level to some ordering

[1] We assume that each process runs the same code.

relation between events at MCS level. To this aim, first we introduce a relation denoted as $\overset{co}{\rightarrow}$ on events generated by a protocol $P \in \mathcal{P}$ executing some history H. Second we point out the relation between $\overset{co}{\rightarrow}$ and $\mapsto_{co}$ and we use such relation to define and prove the notions of safety and optimality.

3.2 The $\overset{co}{\rightarrow}$ relation

The $\overset{co}{\rightarrow}$ relation induces a partial order on the send events of update messages at MCS level. Formally,

Definition 3.2.1 *Given a history H, let $w(x)a$ and $w(y)b$ be two write operations belonging to O_H and $\widehat{E}$ be a computation generated by a protocol $P \in \mathcal{P}$ executing H. $send_j(m_{w(x)a}) \overset{co}{\rightarrow} send_k(m_{w(y)b})$ iff one of the following conditions holds:*

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

Two send events $send_j(m)$, $send_k(m') \in E$ are concurrent w.r.t. $\overset{co}{\rightarrow}$, denoted by $send_j(m)||_{\overset{co}{\rightarrow}}send_k(m')$, if and only if $\neg(send_j(m) \overset{co}{\rightarrow} send_k(m'))$ and $\neg(send_k(m') \overset{co}{\rightarrow} send_j(m))$.

Relation between $\overset{co}{\rightarrow}$ and Lamport's happened before If two send events $send_j(m_{w(x)a})$ and $send_k(m_{w(y)b})$ are related by $\overset{co}{\rightarrow}$ then they are also related by the happened before. The converse is not necessarily true. If $send_j(m_{w(x)a}) \rightarrow send_k(m_{w(y)b})$ and $k \neq j$ but no return event occurs in the run, then $send_j(m_{w(x)a})||_{\overset{co}{\rightarrow}}send_k(m_{w(y)b})$. Therefore, the following property holds:

Property 3.2.1 *Let $w(x)a$ and $w(y)b$ be two write operations belonging to O_H and $\widehat{E}$ be a computation generated by a protocol $P \in \mathcal{P}$ executing H. We have: $send_j(m_{w(x)a}) \overset{co}{\rightarrow} send_k(m_{w(y)b}) \Rightarrow send_j(m_{w(x)a}) \rightarrow send_k(m_{w(y)b})$*

Relation between $\overset{co}{\rightarrow}$ and $\mapsto_{co}$.

The relation $\mapsto_{co}$ induces a partial order on the read/write operations executed at application level. For each protocol $P \in \mathcal{P}$, the execution by ap_k of a write operation $w(x)a$, corresponds to the execution by p_k of a write procedure generating a sequence of events that contains $send_k(w(x)a)$. Moreover, the execution by ap_i of a read operation $r_i(x)a$ corresponds to the execution by p_i of a read procedure generating an event sequence that contains $return_i(x)a$. 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 $\overset{co}{\to}$ to induce a partial order on send events reflecting the partial order induced by $\mapsto_{co}$ on operations. Formally, the following property holds:

Property 3.2.2 *Let $w(x)a$ and $w(y)b$ be two write operations belonging to O_H and $\widehat{E}$ be a computation generated by a protocol $P \in \mathcal{P}$ executing H. We have*

$$send_j(m_{w(x)a}) \overset{co}{\to} send_k(m_{w(y)b}) \Leftrightarrow \; w(x)a \mapsto_{co} w(y)b$$

Proof

(⇒) $send_j(m_{w(x)a}) \overset{co}{\to} send_k(m_{w(y)b})$ means that one of the following condition holds:

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

Case 1. $send_j(m_{w(x)a}) <_k send_k(m_{w(y)b})$ 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 only 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 $w(x)a \mapsto_k w(y)b$.

Case 2. $send_j(m_{w(x)a}) : j \neq k, \; return_k(x,a) <_k send_k(m_{w(y)b})$. 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 $send_k(m_{w(y)b})$ 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, $r(x)a \mapsto_k w(y)b$. Moreover, as ap_k reads a value written by a write $w(x)a$ issued by another process ap_j, it follows that $w(x) \mapsto_{ro} r(x)a$. Then, by the transitive property of $\mapsto_{co}$, $w(x)a \mapsto_{co} w(y)b$.

Case 3. This is the transitive closure of $\overset{co}{\to}$. From the above points it follows that $w(x)a \mapsto_{co} w(y)b$.

($\Leftarrow$) When program order holds for writes invoked by an application process ap_k, the first condition of $\overset{co}{\to}$ 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 $\overset{co}{\to}$ all send events $send_k(m_{w(x)b})$ generated by p_k after $return_k(x, a)$ are such that $send_j(m_{w(x)a}) \overset{co}{\to} send_k(m_{w(y)b})$. When the transitive closure holds, the third condition of $\overset{co}{\to}$ holds, as well. Then the claim follows.

3.3 Safety

Roughly speaking, by definition 2.3.5, an MCS level running a safe protocol w.r.t. $\mapsto_{co}$ has to ensure that, as we consider a history H, given two write operations $w(x)v$ and $w(x)v'$ belonging to H and such that $w(x)v \mapsto_{co} w(x)v'$, if an application process ap_i reads both values v and v', then ap_i first reads v and then v'. In order to assure safety w.r.t. $\mapsto_{co}$, the causality order relations among operations invoked at application level have to be preserved at MCS level. In detail, a protocol $P \in \mathcal{P}$ is safe with respect to $\mapsto_{co}$, if for every pair of write operations $w(x)v$ and $w(x)v'$ such that $w(x)v \mapsto_{co} w(x)v'$, $apply_i(w(x)v) <_i apply_i(w(x)v')$ for every $i \in \{1 \ldots n\}$. By Property 3.2.2, this means that P is *safe* with respect to $\mapsto_{co}$ if and only if the order on local update applications at each MCS process is compliant with the order induced by $\overset{co}{\to}$ on send events of updates.

Formally:

Definition 3.3.1 (Safe Protocol P**)** *Let $\widehat{E}$ be a distributed computation generated by $P \in \mathcal{P}$. P is safe iff*

$$\forall\ m_w, m_{w'} \in M_{\widehat{E}} : (send_j(m_w) \overset{co}{\to} send_k(m_{w'}) \Rightarrow$$

$$\forall\ i \in \{1 \ldots n\},\ apply_i(w) <_i apply_i(w'))$$

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. $\overset{co}{\rightarrow}$. Then, P allows the application of the delayed update (i.e., $A_P(m_w, e)$ flips to *true*) only after all m_w's 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 $send_j(m_{w'}) \overset{co}{\rightarrow} send_k(m_w)$ and $m_{w'}$ has not yet been applied at p_i.

3.4 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 $send_j(m_{w'}) \overset{co}{\rightarrow} send_k(m_w)$ 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 $\overset{co}{\rightarrow}$, 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 $\overset{co}{\rightarrow}$. An optimal protocol is formally defined as follows:

Definition 3.4.1 (Optimal Protocol) *Let $\widehat{E}$ be a distributed computation generated by $P \in \mathcal{P}$. P is optimal iff*

$$\forall\ m_w \in M_{\widehat{E}}, \forall\ e \in E \ \textit{where}\ receipt_i(m_w) <_i e,$$

$$\neg A_P(m_w, e) \Rightarrow \neg A_{Opt}(m_w, e)$$

where

$$A_{Opt}(m_w, e) \equiv \nexists\ m_{w'} \in M_{\widehat{E}} :$$

$$(\ send_j(m_{w'}) \overset{co}{\rightarrow} send_k(m_w)\ \wedge apply_i(w') \notin E_i|_e\ )$$

Then, from Property 3.2.2, 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 $w' \mapsto_{co} w$.

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_i, only after all updates, such that their corresponding send events preceding (for $\overset{co}{\rightarrow}$) 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 $A_P \equiv A_{Opt}$.

3.4.1 $ANBKH$ Protocol

In the seminal paper [7], Ahamad et al. proposed a CRP protocol (hereafter referred to by $ANBKH$) implementing a causal consistent memory on top of a message passing system emulating a reliable broadcast primitive. $ANBKH$ is actually an instance of the general protocol described in Section 3.1, 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 [24]. In $ANBKH$ the activation predicate for each received message $m_{w(y)b}$ at each process p_i, denoted $A_{ANBKH}(m_{w(y)b}, e)$, is the following:

$$\nexists\ m_{w(x)a} :$$

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

Because of such predicate, $ANBKH$ is not optimal even though it is safe. $A_{ANBKH}(m_w, e)$ may be false even in the case each apply event related to a write causally preceding w has occurred. To clarify this point, Figure 3.1 depicts a run generated by $ANBKH$ compliant with history of Example 1.

Let us consider the computation at process p_3. When p_3 receives the update message $m_{w_2(x_2)b}$, the thread associated with the receipt is spawned and an instance of the predicate A_{ANBKH} is created and is set to false. Figure 3.1 shows the evolution of the values of the predicates A_{opt} and A_{ANBKH} associated with the message $m_{w_2(x_2)b}$ at p_3.

Upon the receipt of $m_{w_1(x_1)a}$ and the consecutive application of that update $apply_3(w_1(x_1)a)$, $A_{ANBKH}(m_{w_2(x_2)b}, apply_3(w_1(x_1)a))$ remains *false*. This is because there is another message, namely $m_{w_1(x_1)c}$, which has not yet arrived at p_3 and such that $send_1(m_{w_1(x_1)c}) \rightarrow send_2(m_{w_2(x_2)b})$. However at this point, each $A_{Opt}(m_{w_2(x_2)b}, apply(w_1(x_1)a))$ will flip to *true*. Formally, the non-optimality of $ANBKH$ can be expressed as follows: there exists a distributed computation $\widehat{E} = \{E, \rightarrow\}$ produced by $ANBKH$ such that

$$\exists\ m \in M_{\widehat{E}}, \exists\ e \in E_i :$$

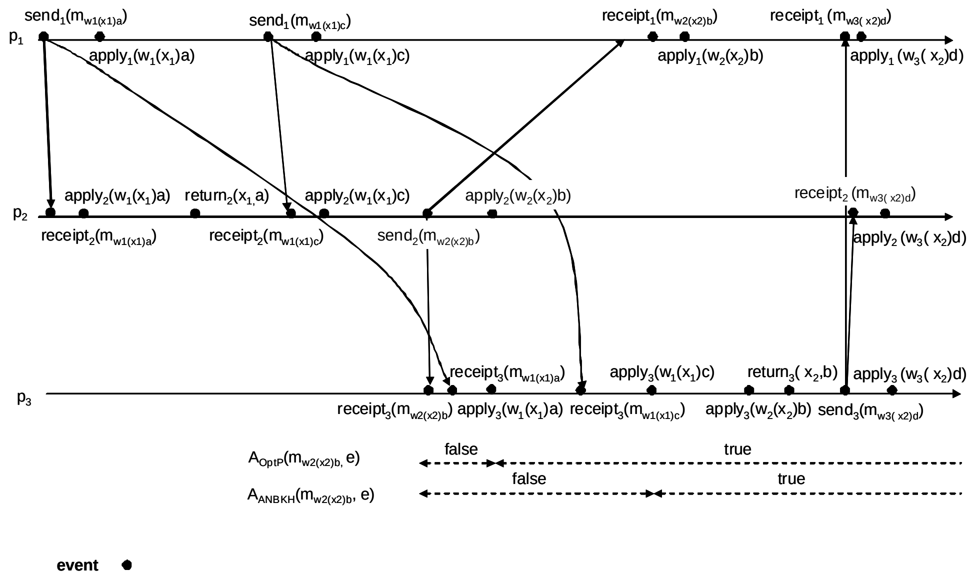


Figure 3.1: A run of $ANBKH$ compliant with the history of Example 1

$$(A_{ANBKH}(m,e) = false \wedge A_{Opt}(m,e) = true)$$

$ANBKH$ 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" [2].

3.4.2 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 $\mapsto_{co}$ [3]. The procedures executed by a MCS process are depicted in Figure 5.1, 5.2 and 3.4. In the following we detail first the data structures and then the protocol behavior.

[2]The false causality notion has been first identified by Lamport [50], then it has received more attention by Cheriton and Skeen [27] and by Tarafdar and Garg [72] in the context of causal message ordering and distributed predicate detection respectively.

[3]The formal notion of system of vector clocks is given in the following where protocol correctness proof is presented.

Data Structures

Each MCS process p_i manages[4] the following local data structures:

$Apply[1...n]$: an array of integers (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 integers (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 k-th write operation invoked by the application process ap_j is the last write operation invoked by ap_j preceding $w_i(x_h)a$ 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 $[0, 0, ..., 0]$.

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 5.1. 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 [5](line 3) and updates the control structures (lines 4,5). In particular, $LastWriteOn[h]$ is set equal to the $w_i(x_h).Write_{co}$.

```
WRITE($x_h$,v)
1  $Write_{co}[i] := Write_{co}[i] + 1$;                % tracking $\mapsto_i$ %
2  RELcast(m($x_h$, v, $Write_{co}$));                  % send event %
3  $x^i_h := v$;                                        % apply event %
4  $Apply[i] := Apply[i] + 1$;
5  $LastWriteOn[h] := Write_{co}$;                      % storing $w_i(x_h)v.Write_{co}$ %
```

Figure 3.2: Write procedure performed by the MCS process p_i

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 5.2. At line

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

[5]When p_i receives a self-sent message, it discards the message.

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.

```
READ($x_h$)
    $\forall k \in [1..n], Write_{co}[k] := max(Write_{co}[k],\ LastWriteOn[h].Write_{co}[k])$;   % tracking $\mapsto_{ro}$ %
    return($x_h^i$);   % return event %
```

Figure 3.3: Read procedure performed by the MCS process p_i

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 3.4.

```
1 when (receipt(m($x_h, v, W_{co}$)) occurs and m was sent by $p_u$ and $(u \neq i)$) do
2     wait until $((\forall\ t \neq u\ W_{co}[t] \leq Apply[t])$ and $(Apply[u] = W_{co}[u] - 1))$   % $A_{opt_P(m,e)}$ %
3     $x_h^i := v$;   % apply event %
4     $Apply[u] := Apply[u] + 1$;
5     $LastWriteOn[h] := W_{co}$;   % storing $w_u(x_h)v.Write_{co}$ %
```

Figure 3.4: p_i's communication thread

If the condition of line 2 in Figure 3.4 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 respect $\mapsto_{co}$. This means that the vector W_{co} in m, i.e. $w_u(x_h)v.Write_{co}$, does 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 3.5 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 3.5 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 broadcasts $m_{w_2(x_2)b}$ 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 keep 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, namely $w_1(x_1)a$, whose update has not arrived at p_3 yet and $w_1(x_1)a \mapsto_{co} w_2(x_2)b$. Therefore the predicate triggering the wait statement at line 2 in Figure 3.4 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. $\mapsto_{co}$.

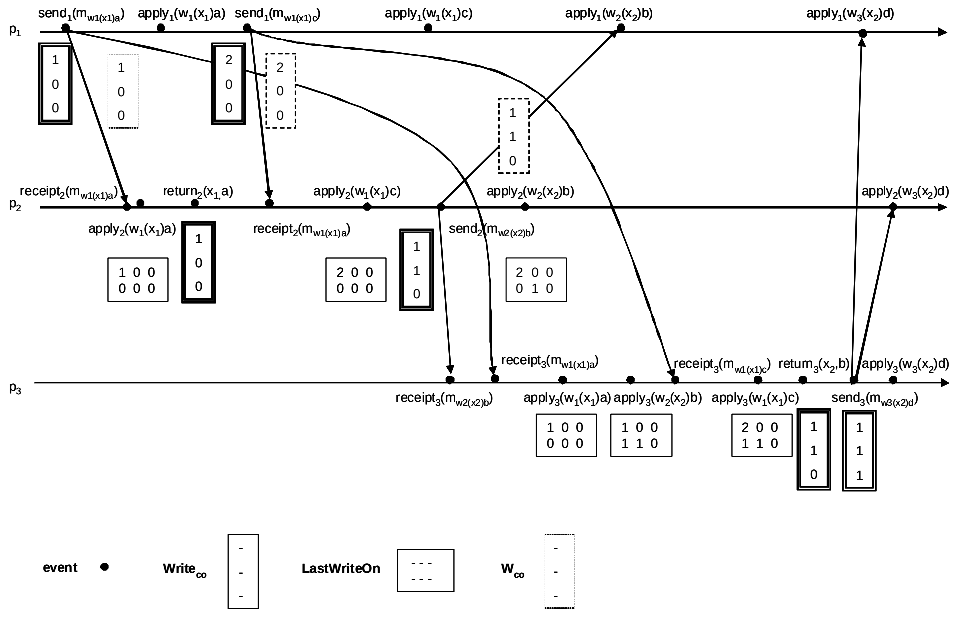


Figure 3.5: A run of *OptP* compliant with the history of Example 1.

A remark on propagation based protocols using partial replication. In a partially replicated environment a variable is replicated only at a subset of processes. These processes are the owners of the variable. Only an owner of a variable can read from and write to it. Then, there are only two main differences between a propagation based protocol using partial replication versus using complete replication: (i) upon a write on a variable x issued by x's owner, an update message is *multicast* to all other owners of x; (ii) let m be the number of shared variables, the control information piggybacked onto an update message is a matrix of size $n \times m$. 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 $\overset{co}{\rightarrow}$. For this reason, the optimality definition applies even in this case, by stating the condition to avoid unnecessary blocking of an applicable update.

Correctness Proof

In this section we first prove that $Write_{co}$ is a system of vector clocks characterizing $\mapsto_{co}$ and $\overset{co}{\rightarrow}$ (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 5.5.2.

Notation $w \mapsto^k_{co} w'$ with $k \geq 1$ 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. 5.1, 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 $w||_{co}w'$ by comparing $w.Write_{co}$ and $w'.Write_{co}$.

Let $Write_{co} = (w.Write_{co}|w \in O_H)$ denote the set of vector clocks values associated to each write by the protocol of Section 5.5.2. 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 3.4.1 $\forall\ w_i, w_j \ \in O_H : \ w_i \neq w_j, (w_i \mapsto_{co} w_j \Rightarrow w_i.Write_{co} < w_j.Write_{co})$

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. $w_i \mapsto_{co}^1 w_j \Rightarrow w_i.Write_{co} < w_j.Write_{co}$
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 5.1. According to line 1 of Figure 5.1, each time p_i executes a write operation, it increments $Write_{co}[i]$. Due to Observation 3, 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 in Fig. 3.4). Then, when p_j executes line 1 of the read procedure (Figure 5.2) 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 3, 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. $w_i \mapsto_{co}^{k>1} w_j$ *then: (i)* $\exists\ w^{'} : w_i \mapsto_{co}^{k-1} w^{'}$. *By the inductive hypothesis we have:* $w_i.Write_{co} < w^{'}.Write_{co}$, *and (ii)* $w^{'} \mapsto_{co}^1 w_j$. *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 3.4.2 $\forall\ w_i, w_j \ \in O_H : \ w_i \neq w_j, (w_i.Write_{co} < w_j.Write_{co} \Rightarrow w_i \mapsto_{co} w_j)$

Proof *The proof is 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 4, $w_j.Write_{co}[i] = k$ 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 4), denoted w', and then it has written w_j. This means that $w' \mapsto_{co} w_j$. Since $h < k$, $w_i \mapsto_i w'$ and then $w_i \mapsto_{co} w_j$ contradicting the initial assumption.

Theorem 3.4.3 $\forall\ w_i, w_j\ \in O_H$:
$w_i \neq w_j, (w_i \mapsto_{co} w_j \Leftrightarrow w_i.Write_{co} < w_j.Write_{co})$

Proof The claim follows from Lemma 3.4.1 and Lemma 3.4.2.

Corollary 3.4.4 $\forall w_i, w_j\ \in O_H$:
$w_i \neq w_j, (w_i \mapsto_{co} w_j \Leftrightarrow w_i.Write_{co}[i] \leq w_j.Write_{co}[i])$

Proof The claim immediately follows from Theorem 3.4.3 and from the code of the protocol of Section 4.2.

Theorem 3.4.5 $\forall w_i, w_j\ \in O_H :\ w_i \neq w_j, (w_i||_{co}w_j \Leftrightarrow w_i.Write_{co}||w_j.Write_{co})$

Proof The claim immediately follows from Theorem 3.4.3 and Definition of concurrency w.r.t. $\mapsto_{co}$.

Corollary 3.4.6 $\forall w_i, w_j\ \in O_H$:
$w_i \neq w_j, (w_i||_{co}w_j \Leftrightarrow w_j.Write_{co}[i] < w_i.Write_{co}[i] \ \wedge\ w_i.Write_{co}[j] < w_j.Write_{co}[j])$

Proof The claim immediately follows from Theorem 3.4.5 and from the code of the protocol of Section 4.2.

Corollary 3.4.7 *$Write_{co}$ is a system of vector clocks characterizing $\overset{co}{\rightarrow}$.*

Proof The claim immediately follows Property 3.2.2, 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 3.4.8 *OptP is safe (see Definition 4).*

Proof Since Property 3.2.2 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. $w_i \mapsto_{co}^{1} w_j$. Let us 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. 5.1)). 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 4 $w_i.Write_{co}[i] = m$ and $w_j.Write_{co}[j] = l$. There are two possible cases:

- ❖ $i = j$. From Corollary 3.4.4 and Observation 3 $w_i.Write_{co}[i] < w_j.Write_{co}[i]$, 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 3.4.4 and Observation 3 $w_i.Write_{co}[i] < w_j.Write_{co}[i]$, 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. $k > 1$. (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) $w' \mapsto_{co}^{1} w_j$. 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 3.4.9 *All write operations invoked by an application process are eventually applied at each MCS process.*

Proof Let us assume by contradiction that 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 $\exists k \neq j \in \{1, \ldots n\} : Apply[k] < w_j.Write_{co}[k]$ 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 3.4.10 *OptP is optimal (see Definition 3.4.1).*

Proof The proof is 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 3.4.7, $Write_{co}$ is a system of vector clock characterizing $\overset{co}{\rightarrow}$. Then, in both cases, there exists an update message $m_{w_k(y)b} \in M_{\widehat{E}}$, respectively with $k \neq u$ or $k = u$, such that $send_k(w_k(y)b)$ precedes $send_u(m_{w_u(x)v})$ w.r.t. $\overset{co}{\rightarrow}$ 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.

3.4.3 Comparison between *ANBKH* and *OptP*

In this section we compare *ANBKH* and *OptP* in order to point out the tradeoff between complexity and performance. In detail, we first formally state complexity and then we analyze performance through simulation results.

Complexity analysis

ANBKH and *OptP* employ a system of vector clocks: respectively $t[1..n]$ and $Write_{co}[1..n]$. *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 vector clock system [17, 25]. Space requirements for local data structure of *ANBKH* is $O(n)$ while the one of *OptP* is $O(n \times m)$ (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 5.2). This fact forces *OptP* to store for each variable x_h, the vector clock associated to the last write operation on x_h.

Simulation

We present simulation results performed to compare *ANBKH* and *OptP* in terms of message buffer overhead. So, let us first point out the relation between the optimality of a protocol and the message buffer overhead at the MCS level.

Message buffer overhead. Message buffer overhead measures the utilization of local buffer at each MCS process used to store update messages. A process buffers a message when this message is not immediately applicable upon its receipt. The message remains in the buffer until it is applied. Upon the receipt of a message m_w, if the activation predicate is evaluated to false, m_w enters the buffer. Ideally, the activation predicate should determine when the message m_w exits the buffer (i.e., as soon as the activation predicate returns true, *apply*(w) immediately occurs). However, this assumption ignores that any operating system could choose to schedule other events, concurrent with *apply*(w), before it[6]. This means m_w is kept in the buffer even though its activation predicate is true. Therefore two factors contribute to the message buffer overhead: (i) the activation predicates and (ii) the scheduler of the operating system. An optimal protocol minimizes the time spent in a buffer by an update message due to its activation predicate. Let us remark that this contribution dominates the one due to the operating system scheduler as soon as the network does not work ideally (where ideally means all messages are delivered to processes in a way compliant to the causality relation).

Causal and FIFO reliable Broadcast [23, 24, 40]. Any MCS protocol relying on a causal reliable broadcast can apply each message upon its arrival. There is therefore no need of an activation predicate at MCS level, the reordering necessary to ensure causal consistency is embedded in the communication primitive. In other words, an activation predicate works at communication level deciding for the delivery of messages. The accuracy of the activation predicate affects the buffer overhead exactly as that of the MCS activation predicate. If a MCS protocol relies on FIFO reliable broadcast, there is the need of an activation predicate associated with each messages at the MCS level to reorder update messages producing causality violation which are not FIFO ones.

As a consequence, the evaluation of an MCS protocol implementing causal consistency in terms of message buffer overhead at the MCS level makes sense only if the protocol is built on the top of a communication system providing either a reliable or a FIFO reliable broadcast. Moreover, the comparison between different MCS protocols in terms of message buffer overhead can be done in a fair way only if these protocols employ the same broadcast primitive. This is why in the rest of this section we compare $ANBKH$ and $OptP$ both using a reliable broadcast primitive [7].

Simulation Description We used the discrete event simulator OMNeTpp 2.3b1 [74]. 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

[6]These events could be *send*, *receive* or either *apply* events of write operations concurrent with w.

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

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, so variance is not reported.

Simulation Results

Figure 3.6 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 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 O_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 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 may be applied at each process in arrival order as their corresponding send events are concurrent w.r.t. $\xrightarrow{co}$. Therefore $OptP$ only buffers those messages which are out of FIFO order.

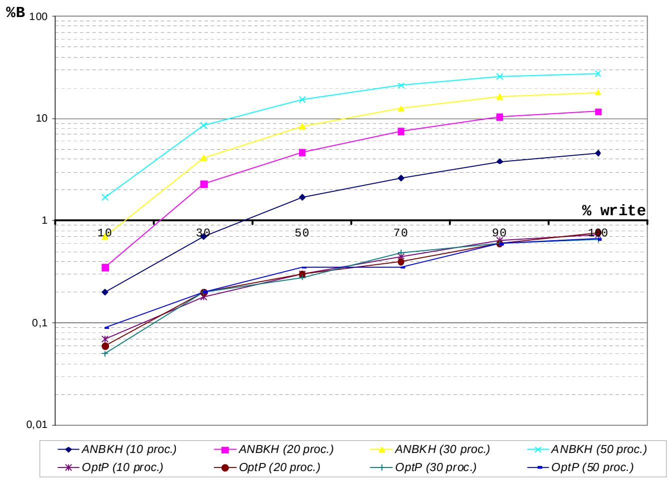


Figure 3.6: %B versus %write

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 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.

3.5 Related Work

Several other protocols implementing causal consistency have appeared in the literature [9, 19, 46, 63].

The protocols presented in [19] and [63] are propagation-based protocols assuming partial replication. They both address the *writing semantics* notion introduced by Raynal and Ahamad [63]. 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 only if there is no 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. However, writing semantics and optimality are orthogonal notions. Then, writing semantics could also be applied to any optimal protocols.

The protocol presented in [46] is a propagation-based protocol using complete replication. However, differently from protocols in [7, 19, 63], the propagation is not immediate but it is token-based. Thus, each process executes its write operations locally and when it obtains the token, it broadcasts a message containing the last written values (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 [9] 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 [48] 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 3.4. The latter aims 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 [34]. The authors introduce a new relation $\xrightarrow{s}$ that tracks only true causality given an application semantics. From an implementation point of view, they approximate $\xrightarrow{s}$ 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 $\mapsto^{co}$. The introduction of $\xrightarrow{co}$ allows to track the application semantics among the events of distributed computations generated by any CRP protocol. That leads to a complete removal of false causality created by the "happened-before" relation at the MCS level.

Chapter 4

Partially Replicated Causal Memory Implementations

To exploit concurrency allowed by causal consistency most protocols implementing distributed causal memories support replication of variables. Replication allows different processes to contemporary read or write a same shared variable by accessing their local copies. On the other hand, replication imposes the cost of maintaining replica consistent. According to this, in this chapter we exploit the limit of causal memory implementations that support partial replication: each process implementing the memory stores a local copy of each variable the corresponding application process aims at reading or writing.

The interest for partial replication relies on the fact that there exist several environments where supporting complete replication may be too expensive in terms of consistency maintenance. This is the case, for example, of large-scale systems spanning geographically distant sites. Let us notice that these last are recognized to be naturally appropriate for distributed applications supporting collaboration [14]. Causal consistency well captures collaborative applications semantics. Moreover, as the number of application processes and variables grow, it is reasonable to assume that each application process may be interested in accessing only a subset of complete variables space. In this sense, implementations based on partial replication are intended to avoid processes to manage information about variables they are not interested in. This makes partial replication promising w.r.t. scalability.

In this chapter we point out the limit of partial replication when implementing causal memory in traditional distributed systems (see section 2.7.1). In detail, we formally characterize which MCS processes are concerned by information on the occurrence of operations performed on the variable x in order to maintain causal consistency. Ideally, we would like that only MCS processes having a local copy of x have to manage information about x. But unfortunately, as we will prove in this chapter, if the variable distribution is not a priori known, it is not possible for the MCS to ensure a causally consistent shared memory, if each MCS process p_i only manages information about variables it locally stores. Let us notice that in the literature some causal memories implementations based on partial replication [19, 63] have been proposed, but they suffer this drawback.

Part of the results presented in this chapter have been published in [42] and [41].

4.1 Problem Statement

We consider traditional distributed systems implementations 2.7.1. We assume partial replication of variables at each MCS process p_i. This means that each process p_i endows a copy of a subset of the shared variables $\mathcal{X}$, that we denote $\mathcal{X}_i$.

Our aim is to determine which MCS processes are concerned by information on the occurrence of operations performed on the variable x in the system. More precisely, given a variable x, we will say that a MCS process p_i is x-*relevant* if, in at least one computation, it has to transmit some information on the occurrence of operations performed on variable x in the corresponding history, to ensure a causally consistent shared memory. Of course, each process managing a replica of x is x-relevant. Ideally, we would like that only those processes are x-relevant. But unfortunately, as will be proved in the next section, if the variable distribution is not a priori known, it is not possible for the MCS to ensure a causally consistent shared memory, if each MCS process p_i only manages information about $\mathcal{X}_i$. The main result of the chapter is a characterization of x-relevant processes.

4.2 A characterization of x-relevant processes

In this section, we characterize x-relevant processes for any given variable x. In order to prove our result, we use notions and notation introduced respectively in section 2.3 and section 2.6.

Theorem 4.2.1 *Given a variable x, an MCS process p_i is x-relevant w.r.t. $\mapsto_{co}$ $\Leftrightarrow$ the corresponding application process ap_i belongs to $C(x)$ or to a minimal x-hoop.*

Proof $\Leftarrow$ If $ap_i \in C(x)$, then p_i is obviously x-relevant. Consider now a process $ap_i \notin C(x)$, but belonging to a minimal x-hoop between two processes in $C(x)$, namely ap_a and ap_b. Let $[ap_a = ap_0, ap_1, \ldots, ap_k = ap_b]$ be such minimal x-hoop, then the history H depicted in Figure 2.4 can be generated. H includes an x-dependency chain along this hoop from $w_a(x)v$ to $o_b(x)v$ and, by definition 2.6.5, it follows that $w_a(x)v \mapsto_{co} o_b(x)$. Thus if $o_b(x)$ is a read operation, the value that can be returned is constrained by the operation $w_a(x)v$, i.e., to ensure causal consistency, it cannot return neither $\perp$ nor any value written by a write operation belonging to the causal past of $w_a(x)v$. Similarly, if $o_b(x)$ is a write operation, namely $o_b(x) = w_b(x)v'$, the relation $w_a(x)v \mapsto_{co} w_b(x)v'$ implies that, to ensure causal consistency, if a process $ap_c \in C(x)$ reads both values v and v' then it reads them in such a order.

In both cases, to ensure causal consistency, process p_b has to be aware that $w_a(x)v$ is in the causal past (see definition 2.3.4) of $w_{k-1}(x_k)v_k$. Since p_a cannot be aware of $w_a(x)v$ causal future, this information has to arrive from p_{k-1}.

Let us remark that $w_a(x)v \mapsto_{co} o_b(x)$ since $w_a(x)v \to_a w_a(x_1)v_1$, and, for each h, $1 \leq h \leq k-1$, $r_h(x_h)v_h \to_h w_h(x_{h+1})v_{h+1}$, and $r_b(x_k)v_k \to_b o_b(x)$. This means that $w_a(x)v$ is in the causal past of $w_h(x_{h+1})v_{h+1}$ because $w_a(x)v$ is in the causal past of $w_{h-1}(x_h)v_h$, for each h such that $1 \leq h \leq k-1$. Moreover, since the hoop is minimal, the only way for process p_h to be aware that $w_a(x)v$ is in the causal past of $w_{h-1}(x_h)v_h$ is through p_{h-1}, for each h s.t. $1 \leq h \leq k-1$. Then for each h such that $1 \leq h \leq k-1$, p_h is x-relevant. In particular p_i is x-relevant.

$\Rightarrow$ The purpose of MCS processes when transmitting control information concerning the variable x is to ensure causal consistency. In particular, if an operation o_1=$\mathrm{w}_a(x)v$ is performed by a process $ap_a \in C(x)$, then any operation $\mathrm{o}_2 = o_b(x)$ performed by another process $ap_b \in C(x)$ is constrained by o_1 only if $\mathrm{o}_1 \mapsto_{co} \mathrm{o}_2$. According to this, to maintain consistency an MCS process has to be aware that o_1 is in the causal past of o_2. We have that $\mathrm{o}_1 \mapsto_{co} \mathrm{o}_2$ only if one of the two following cases holds:

1. A "direct" relation: $\mathrm{o}_1 \mapsto_{ro} \mathrm{o}_2$. In this case, no third part process is involved in the transmission of information concerning the occurrence of the operation o_1.
2. An "indirect" relation: there exists at least one o_h such that $\mathrm{o}_1 \mapsto_{co} \mathrm{o}_h$ and $\mathrm{o}_h \mapsto_{co} \mathrm{o}_2$. Such an indirect relation involve a sequence σ of

processes $p_0 = p_a, \ldots, p_h, \ldots, p_k = p_b$ $(k \geq 2)$ such that two consecutive processes p_{h-1} and p_h $(1 \leq h \leq k)$ respectively perform operations o_{h-1} and o_h with $\mathrm{o}_{h-1} \mapsto_{ro} \mathrm{o}_h$. This implies that there exists a variable x_h such that $\mathrm{o}_{h-1}=\mathrm{w}_{h-1}(x_h)v_h$ and $\mathrm{o}_h=\mathrm{r}_h(x_h)v_h$. Consequently, x_h is shared by p_{h-1} and p_h, i.e., p_{h-1} and p_h are linked by an edge in the graph SG, meaning that the sequence σ is a path between p_a and p_b in the share graph SG. Such a path is either completely included in $C(x)$, or is a succession of x-hoops, and along each of them there is a x-dependency chain. Thus, a process $p_i \notin C(x)$ and not belonging to any x-hoop cannot be involved in these dependency chains.

Let us notice that as we consider one of the above said x-hoop or it is a minimal hoop or there exists at least a variable y that is shared by more than two processes, i.e. at least two edges of the hoop are labelled with y. Let ap_i, ap_j and ap_k be such processes. Without loss of generality, we can assume that ap_i precedes ap_j that precedes ap_k in the hoop. This means that an x-dependency chain along this hoop will always have at least two write operations on y executed by ap_i and ap_j and a read operation performed by ap_k on y. Let $w_a(x)v$, $w_i(y)v'$ and $w_j(y)v''$ respectively be the initial operation of the x-dependency chain and the write operation performed by ap_i and ap_j. Then, we have that $w_a(x)v \mapsto_{co} w_i(y)v' \mapsto_{co} w_j(y)v''$.

So independently by the value return by the read operation performed by ap_k, i.e. v' or v'', p_k has to be aware that $w_a(x)v$ is in the causal past of $w_i(y)v'$ and then of $w_j(y)v''$. So p_k may be advised that $w_a(x)v$ is in the causal past of $w_i(y)v'$ by ap_i. Notice that since ap_i and ap_k share variable y, there is a minimal x-hoop obtained by considering the previous hoop and eliminating all processes between ap_j and ap_k.

So if an application process does not belong to an x-minimal hoop or to $C(x)$, the corresponding MCS has not to manage information about x to maintain consistency.

The result follows from the fact that this reasoning can be applied to any variable x, then to any pair of processes p_a and p_b in $C(x)$, and finally to any x-dependency chain along any x-hoop between p_a and p_b. $\square$

4.3 About the Efficiency of Partial Replication Implementations

From theoretical results obtained in Section 4.2, several observations can be made, depending on which is the *a priori* knowledge on variables distribution.

If a particular distribution of variables is assumed, it could be possible to build the share graph and analyze it off-line in order to enumerate, for each variable x not totally replicated, all the x-hoops. It results from theorem 4.2.1 that only processes belonging to one of these minimal x-hoops will be concerned by the variable x. Thus, an ad-hoc implementation of causal DSM can be optimally designed. However, even under this assumption on variable distribution, enumerating all the hoops can be very long because it amounts to enumerate a set of paths in a graph that can be very big dependently on the number of processes.

In a more general setting, implementations of DSM cannot rely on a particular and static variable distribution, and, in that case, any process is likely to belong to any hoop. It results from theorem 4.2.1 that each process in the system has to transmit control information regarding all the shared variables, contradicting scalability. This lead to inefficient partial replication implementations in terms of information to be managed in order to maintain consistency. Especially, this lack of variables distribution knowledge imposes processes to manage some information that is not necessary to ensure causal consistency.

Our reasoning also applies to sequential consistency (see section 2.3). On the contrary, in the case of atomic consistent memory, partial replication implementations do not suffer the above said draw back. This due to the locality property, i.e. the memory is atomic whenever each single variable is atomic [43]. But as already stated, to provide strong consistency criteria (i.e.atomic and sequential) global synchronization is requested among processes implementing the memory. This makes strong consistency solutions not efficient and not scalable in terms of processes.

In the next section, we weaken causal consistency with the aim at finding a memory semantics strong enough to be somewhat useful and weak enough to be efficiently implementable in a partial replicated environment. We explore the gap between causal consistency and Pram.

4.4 Weakening the Causal Consistency Criterion

In the proof of theorem 4.2.1, we point out that information propagation in order to maintain consistency are due to dependency chains that can be created along the hoops. In the next sections we investigate new order relations obtained by weakening the causality order relation such that, for any variable x, x-hoops cannot lead to the creation of x-dependency chains. The notion of dependency chain has been defined with respect to the particular order relation introduced. This definition holds for any relation defined on the sets O_H, just by replacing the $\mapsto_{co}$ relation by the appropriate relation, and Theorem 4.2.1 still hold in this new setting.

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

4.4.1 Lazy Causal Consistency

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

Definition 4.4.1 (Lazy program order) *Given two operations o_1 and o_2 in h_i, $o_1 \rightarrow_{li} o_2$ iff o_1 is invoked before o_2 and one of the following condition holds:*

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

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

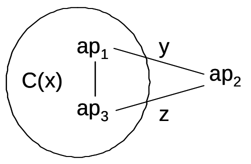
Definition 4.4.2 (Lazy causal order) *Given two operations o_1 and o_2 in O_H, $o_1 \mapsto_{lco} o_2$ if and only if one of the following cases holds:*

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

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

Definition 4.4.3 (Lazy Causally Consistent History) *A* history H *is lazy causally consistent if for each application process* ap_i *there is a serialization* S_i *of* H_{i+w} *that respects* $\mapsto_{lco}$.

A memory is lazy causal if it admits only lazy causally consistent histories. Figure 4.1 depicts a history which is lazy causal but not causal.



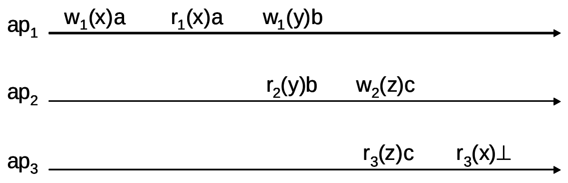


Figure 4.1: A lazy causal but not causal history

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

In this history, no x-dependency chain is created along the x-hoop $[p_1, p_2, p_3]$. In fact, even though $w_1(x)a \mapsto_{lco} r_3(z)c$, we have, according to definition 4.4.3, $r_3(z)c||_{lco}r_3(x)\bot$ and thus $w_1(x)a \not\mapsto_{lco} r_3(x)\bot$. In particular, the value returned by the last read operation is consistent.

However, the situation is different if we consider the history depicted in Figure 4.2. In that case, an x-dependency chain along the x-hoop $[p_1, p_2, p_3]$ is created since, according to definition 4.4.3, $r_3(z)c \rightarrow_{l3} w_3(x)d$ and thus $w_1(x)a \mapsto_{lco} w_3(x)d$.

In particular, if process ap_4 reads both values a and d, it has to read them in this order (it is not the case in the history depicted Figure 4.2, which is *not*

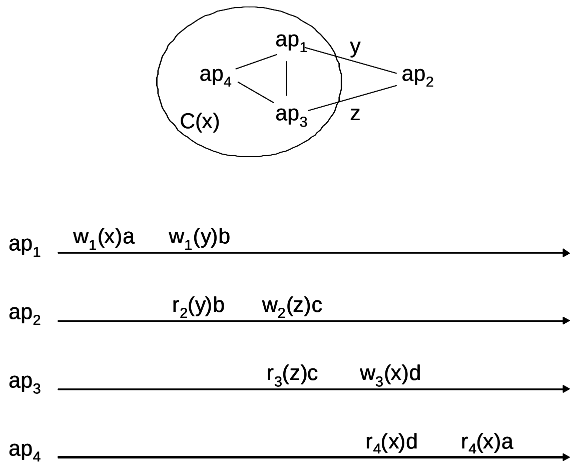


Figure 4.2: A not lazy causal history

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

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

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

4.4.2 Lazy Semi-Causal Consistency

Ahamad et al. [5] have introduced a weakened form of read-from relation, called *weak writes-before*. Their definition is based on a weakened program order, called *weak program order*, that is stronger than the *lazy program order* introduced in the previous section. In this section, we introduce the *lazy writes-before* relation, obtained from the weak writes-before by substituting lazy program order to weak program order. This relation, denoted $\rightarrow_{lwb}$, is formally defined as follows.

Definition 4.4.4 (Lazy write-before order) *Given two operations o_1 and o_2 in O_H, the* lazy writes-before *order relation, $\rightarrow_{lwb}$, on some history H is any relation with the following properties:*

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

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

Formally:

Definition 4.4.5 (Lazy semi-causal order) *Given two operations* o_1 *and* o_2 *in* O_h*,* $o_1 \mapsto_{lsc} o_2$ *if and only if one of the following cases holds:*

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

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

Definition 4.4.6 (Lazy Semi-Causally Consistent History) *A* history H *is lazy semi-causally consistent if for each application process* ap_i *there is a serialization* S_i *of* H_{i+w} *that respects* $\mapsto_{lsc}$*.*

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

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

More precisely, in Figure 4.3 we have $w_1(x)a \mapsto_{lsc} w_3(x)d$. In fact, $w_1(x)a \mapsto_{lwb} r_2(y)b$ (because of $w_1(y)b$) and $w_2(y)e \mapsto_{lwb} r_2(z)c$ (because of $w_2(z)c$). Then, since $r_2(y)b \mapsto_{li} w_2(y)e$ and $r_3(z)c \mapsto_{li} w_3(x)d$, due to transitivity we have $w_1(x)a \mapsto_{lsc} w_3(x)d$.

In particular, if process p_4 reads both values a and d, it has to read them in this order (it is not the case in the history depicted Figure 4.3, which is *not* lazy semi-causal consistent). In particular, p_2 is x-relevant, although $p_2 \notin C(x)$. In this sense, the new order relation is still too strong to allow efficient partial replication.

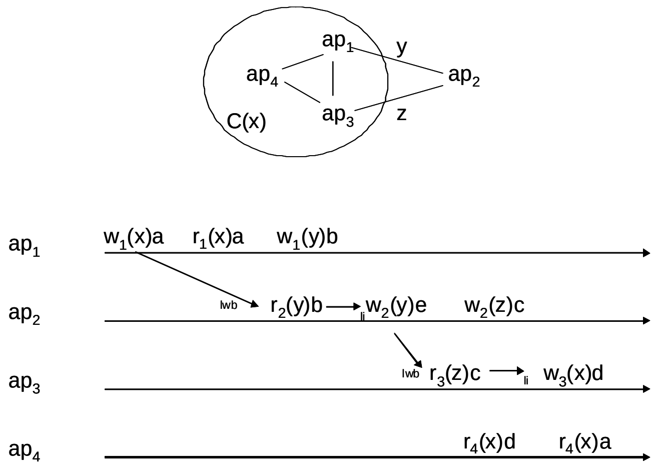


Figure 4.3: A not lazy semi-causally consistent history

It must be noticed that, since the semi-causality order relation, introduced by Ahamad et al. in [5], is stronger than the lazy-semi-causality introduced here, the semi-causality order relation does not allow efficient partial replication either.

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

4.4.3 The case of PRAM

As stated in section 2.3, the PRAM consistency criterion is weaker than the causal consistency criterion. The PRAM consistency is based on a relation, denoted $\mapsto_{pram}$, weaker than $\mapsto_{co}$. Formally [64][1]:

Definition 4.4.7 (PRAM relation) *Given two operations o_1 and o_2 in O_H, $o_1 \mapsto_{pram} o_2$ if, and only if, one of the following conditions holds:*

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

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

[1]in [64] this relation is denoted $\mapsto_{H'}$.

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

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

Theorem 4.4.1 *In a PRAM consistent history, no dependency chain can be created along hoops.*

Proof Let x be a variable and $[p_a, \ldots, p_b]$ be a x-hoop. A x-dependency chain along this hoop is created if H includes $w_a(x)v$, $o_b(x)$ and a pattern of operations, at least one for each process of the x-hoop, implying $w_a(x)v \mapsto_{pram} o_b(x)$. But the latter dependency can occur only if point 1 or point 2 of Definition 4.4.7 holds. Point 1 is excluded because $a \neq b$. Point 2 is possible only if $o_b(x) = r_b(x)v$ and the dependency $w_a(x)v \mapsto_{pram} r_b(x)v$ is $w_a(x)v \mapsto_{ro} r_b(x)v$, i.e., does not result from the operations performed by the intermediary processes of the hoop.

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

Although being weaker than causal memories, Lipton and Sandberg show in [53] that PRAM memories are strong enough to solve a large number of applications like FFT, matrix product, dynamic programming and more generally the class of oblivious computations[2]. In his PhD, Sinha [70] shows that totally asynchronous iterative methods to find fixed points can converge in Slow memories, which are still weaker than PRAM. In the next section we illustrate this power, together with the usefulness of partial replication, by showing how the Bellman-Ford shortest path algorithm can be solved by using PRAM memory.

4.5 A Case Study: Bellmann-Ford algorithm

A packet-switching network can be seen as a directed graph, $G(V, \Gamma)$, where each packet-switching node is a vertex in V and each communication link between node is a pair of parallel edges in Γ, each carrying data in one direction. In such a network, a routing decision is necessary to transmit a packet from a source node to a destination node traversing several links and packet switches. This can be modelled as the problem of finding a path through the graph.

[2]"A computation is oblivious if its data motion and the operations it executes at a given step are independent of the actual values of data."[53]

Analogously for an Internet or an intranet network. In general, all packet-switching networks and all internets base their routing decision on some form of least-cost criterion, i.e minimize the number of hope that correspond in graph theory to finding the minimum path distance. Most least-cost routing algorithms widespread are a variations of one of the two common algorithms, Dijkstra's algorithm and the Bellman-Ford algorithm[20].

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

In the following we propose a distributed implementation of the Bellman-Ford algorithm to compute the minimum path from a source node to every other nodes in a system, pointing out the usefulness of partial replication to efficiently distribute the computation. Another distributed implementation of Bellman-Ford algorithm has been provided by Bertsekas and Tsitsiklis in [22]. They also discuss conditions under which asynchronous and synchronous computations of fixed point algorithms converge. In the following we refer to nodes as processes.

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

Let us use the following notation:

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

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

1. **[Initialization]**
   $x_i^0 = \infty$, $\forall\ n \neq s$
   $x_s^k = 0$, for all k

2. **[Update]** for each successive $k \geq 0$:

   $\forall\ i \neq s$, compute $x_i^{k+1} = \underset{j \in \Gamma^{-1}(i)}{min} [x_j^k + w(j,i)]$

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

The algorithm is distributively implemented as follows . Without loss of generality, we assume that process ap_1 is the source node. We denote as x_i the current minimum value from node 1 to node i. Then, to compute all the minimum path from process ap_1 to every other process in the system, processes cooperate reading and writing the following set of shared variables $\mathcal{X} = \{x_1,\ x_2, \ldots,\ x_N\}$. Moreover, since the algorithm is iterative, in order to ensure liveness we need to introduce synchronization points in the computation. In particular, we want to ensure that at the beginning of each iteration each process ap_i reads the new values written by his predecessors $\Gamma^{-1}(i)$. Thus each process knows that at most after N iterations, it has computed the shortest path. With this aim, we introduce the following set of shared variables $\mathcal{S} = \{k_1,\ k_2,\ \ldots,\ k_N\}$.

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

```
Minimum Path
1  $k_i := 0;$
2  if $(i == 1)$
3     $x_i := 0;$
4     else  $x_i := \infty;$
5  while $(k_i < N)$ do
6          while $(\bigwedge_{h \in \Gamma^{-1}(i)} (k_h < k_i))$ do;
7          $x_i := min([x_j + w(j,i)]\ \forall j \in \Gamma^{-1}(i));$
8          $k_i := k_i + 1$
```

Figure 4.4: pseudocode executed by process ap_i

Since each variable x_i and k_i is written only by one process, namely ap_i, it is simple to notice that the algorithm in Figure 4.4, correctly runs on a PRAM shared memory. Moreover, since each process has to access only a subset of the shared memory, we can assume a partial replication implementation of such memory. In particular, at each node where the process ap_i is running to compute the shortest path, there is also a MCS process that ensure Pram consistency in the access to the shared variables.

The algorithm proposed is deadlock-free. In fact, given two processes ap_i and ap_j such that ap_i is a predecessor of ap_j and viceversa, the corresponding barrier conditions (line 6 of Figure 4.4) cannot be satisfied at the same time: $k_i < k_j$ and $k_j < k_i$.

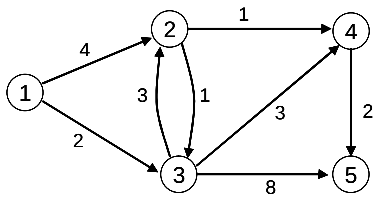


Figure 4.5: An example

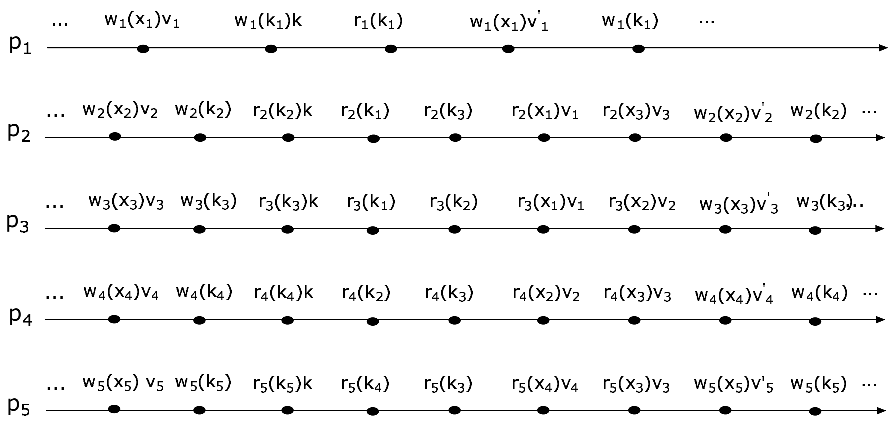
As an example, let us consider the network depicted in Figure 4.5. We have the following set of processes $\Pi = \{ap_1,\ ap_2,\ ap_3,\ ap_4,\ ap_5\}$ and the corresponding variable distribution:

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

In Figure 4.6 we show the pattern of operations generated by each process at the k-th step of iteration, we only explicit value returned by operations of interest. In reality, in order to point out the sufficiency of PRAM shared memory to ensure the safety and the liveness of the algorithm, we start the scenario showing the two last write operations made by each process at $(k-1)$-th step. In this sense, it must be noticed that the protocol correctly runs if each process reads the values written by each of its neighbors according to their program order.

4.6 Related Work

In this chapter we have formally defined necessary and sufficient conditions on processes intended to manage information in order to guarantee causal consistency. This theoretical result leads to several considerations regarding efficient distributed shared memory implementations. One of the key problems in building efficient software DSM systems is to minimize the amount of communication and information needed to keep the distributed memories



p1
... w1(x1)v1 w1(k1)k r1(k1) w1(x1)v'1 w1(k1) ...
p2
... w2(x2)v2 w2(k2) r2(k2)k r2(k1) r2(k3) r2(x1)v1 r2(x3)v3 w2(x2)v'2 w2(k2) ...
p3
... w3(x3)v3 w3(k3) r3(k3)k r3(k1) r3(k2) r3(x1)v1 r3(x2)v2 w3(x3)v'3 w3(k3)...
p4
... w4(x4)v4 w4(k4) r4(k4)k r4(k2) r4(k3) r4(x2)v2 r4(x3)v3 w4(x4)v'4 w4(k4) ...
p5
... w5(x5) v5 w5(k5) r5(k5)k r5(k4) r5(k3) r5(x4)v4 r5(x3)v3 w5(x5)v'5 w5(k5) ...

Figure 4.6: A step of the protocol in Figure 4.4 for the network in Figure 4.5

consistent. According to this, we have shown that partial replication implementations generally require each process to manage information about the entire variables space in order to keep the memory consistent.

As a practical example, let us consider the propagation-based protocol proposed by Ahamad et et. in [8] to implement a causal memory supporting partial replication. They consider a static replication scheme and they assume that only processes having a local copy of a variable may access such variable. On the other hand, in order to ensure causal consistency each process manages data structures that track causality relation among all shared variables. To maintain causal consistency, each time a write operation on a variable x is issued, a corresponding update message is sent to all processes having a local copy of x. Such messages have to piggyback some control information whose size is strictly related to the number of processes in the system and to the number of shared variables. So while reducing the number of messages to be sent, partial replication increments messages size with respect to similar complete replication implementation [7].

Similar studies have been done in several contexts related to causality in distributed systems.

Vector clock systems are the main mechanism to track causality among events in a distributed computation, [50]. Simultaneously and independently introduced by Fidge [30] and Mattern [58], vector clock systems associate each event with a vector of size n, where n is the number of processes in the system. Moreover, each time a message is sent, it is timestamped with the current value of the vector clock at the sender. To track causality Charron-Bost proved that in the worst case the size of the vector clock to be piggybacked by a message has to be equal to the number of processes in the system, [25]. In [69] Kshemkalyani et al. have proposed a technique to reduce the average size of vector to be piggybacked. This last is based on the observation that in general few processes interact frequently and that few entries of the vector change between different send events. On the other hand, Ahamad et al. explore the different way of plausible clocks [10], trying to reduce the size of control information to be used by paying a decrease in accuracy tracking. Finally, in [2] Agarwal and Garg proposed the chain clocks, which overcome the limits of vector clock for particular applications, e.g. predicate detections. Chain clocks are based on the idea to divide the computation poset into a set of chains and then associate each chain with a vector entry. Notice that vector clocks are a particular implementation of chain clocks when considering events at each process as a chain.

In the context of causal message ordering, Kshemkalyani et al. in [48] have presented the necessary and sufficient conditions on information required to enforce causal message delivering. They also propose an optimal implementation with respect to message complexity space. Introduced by Birman et al. in [23] causal message ordering states that given two send events related by the happened before relation, if such messages have a common destination they have to be delivered according to the happened before order. From the first implementation, ISIS [23], several solutions have been proposed in order to reduce the control information to be piggybacked on messages to ensure causal delivery, [66], [68].

Among the existing software distributed shared memory systems (see [28] for a comprehensive bibliography), Munin proposed by Bennet et al. in [21] addresses the problem of reducing synchronization required to implement strong consistency exploring program's semantics. Munin incorporates several techniques investigated in order to reduce consistency-related communication [45]. Bennett et al. [21] made some interesting observations regarding replication or single copy approach to implement distributed shared memory. The right degree of replication may be different under different circumstances and often depends on program's semantics.

Chapter 5

Weakly-Persistent Causal Memory

Intuitively, an intended correct behavior for a shared memory has to ensure that any read operation will return the *last written* value and that any written value may be returned by a *successive read* operation. Unfortunately to enforce this last property in a dynamic distributed system (e.g. P2P system) is far from being trivial. The difficulty arises from the fact that processes implementing such memory may fail or leave the computation at any time, without notifying any other process.

According to this, in this chapter, we introduce the concept of *weakly-persistent causal memory.* While ensuring that every read operation is legal w.r.t. $\mapsto_{co}$, such memory guarantees persistency (see definition 2.5.2) only for write operations written in quiescent periods(arrivals and departures subside). For sake of clarity, an implementation of a weakly-persistent causal object is first provided along with its correctness proof. Then, a weakly persistent causal memory is proposed by extending previous algorithms.

Part of the results presented in this chapter have been published in [16].

5.1 Weakly-Persistent Causal Consistency

Dynamic distributed systems are characterized by churn, i.e. the change in the set of participating nodes due to joins, leaves and failures. While generally imposing a tradeoff between weakening guarantees and increasing costs [38], dealing with high churn leads to ensure some properties only in quiescent periods, i.e. when processes joins and leaves subside. This is the case of object persistency, i.e. every value written on such an object as to be persistent.

In that sense, let us remark as in a failure free environment, e.g. the one presented in section 2.7.2, an object implemented by a set of processes is trivially persistent if a write operation is applied to all processes and a read operation waits for one reply to return the value.

On the contrary, when considering the failure model presented in section 2.7.2, persistency can only be guaranteed for values written in quiescent periods of the system. In other words, it is necessary that a subset of object entities do not suffer memory losses (i.e. processes incarnating the object do not leave the system deliberately or by crashing). But, in non-quiescent periods of the system, potentially all object entities could suffer a memory loss. This makes non persistent the object described by the previous trivial implementation. This is why we need to introduce the notion of a weak form of object persistency, namely weakly-persistent object.

To this aim let us first introduce the notion of complete operation. An operation may be seen as a pair of invocation and response events. We say that an operation is *complete* if its invocation is followed by the corresponding response event in the computation.

Our definition of weakly-persistent object aims at capturing the fact that if the system experiences some quiescent periods, then every complete write operation issued during such periods has to be persistent. Let us notice that a complete write operation may have been executed by a client process that successively leaves the system deliberately or by crashing.

Formally,

Definition 5.1.1 (Weakly-Persistent Object) *An object x is weakly-persistent if there exists a time t such that every complete write operation issued after t is persistent.*

Roughly speaking, in a weakly-persistent object to ensure that a value could be eventually read by a client that reads infinitely many times, this value has to be written infinitely many times. Therefore, a value written a finite number of times in a weakly-persistent object may never be read due to the fact that the write operations could be issued during the non-quiescent period of the system.

Finally, let us remark that if some process read a value written by a write operation that does not complete, such a write has to appear in the history to ensure read legality. On the other hand, we do not request such a write to be persistent. So it may be never read by any other process in the system even though not overwritten.

Weakly-Persistent Causal Consistency Roughly speaking, a Weakly-Persistent Causal Object is a read/write object that always ensures legal reads w.r.t. $\mapsto_{co}$ but that guarantees persistency only of write operation written in quiescent periods.

Formally,

Definition 5.1.2 (Weakly-Persistent Causal Object) *An object x is* weakly-persistent causal *if both the following conditions hold:*

1. *x admits only histories H such that all read operation in $\widehat{H} = (O_H, \mapsto_{co})$ are legal;*
2. *x is weakly-persistent.*

Finally,

Definition 5.1.3 (Weakly-Persistent Causal Memory) *A memory $\mathcal{X} = \{x_1, x_2, ...x_m\}$ is* weakly persistent causal *if both the following conditions hold:*

1. *$\mathcal{X}$ admits only histories H such that all read operation in $\widehat{H} = (O_H, \mapsto_{co})$ are legal;*
2. *each $x_i \in \mathcal{X}$ is weakly-persistent.*

For sake of clarity, in next sections we first present an implementation of a weakly-persistent causal object x along with its correctness proofs. Then, we extend such implementation in order to provide a weakly persistent causal memory.

5.2 Weakly-Persistent Causal Object Implementation

We consider the system model presented in section 2.7.2. Client processes invoke operations (read or write) by sending request messages to the set of n object entities $\{o_1, o_2, \ldots, o_n\}$ implementing the object x. We denote as x^i the local copy of x at o_i. Since each object entity o_i is incarnated by an object manager process that may change during time, we assume the existence of an underlying routing system that is able to route request messages to the object manager process that currently incarnates o_i. When an object manager process receives a request of a client process c_i, it processes that request and then it sends the corresponding response to c_i. There is neither communication among object manager processes nor among clients.

The main idea to ensure causal ordering among operations, is to exploit the write semantics of write operations as done by Ahamad et al. in the context of traditional distributed systems [63]. Roughly speaking, when an object manager process receives a message corresponding to a write operation $w(x)a$, if not obsolete, it is immediately applied even though some write operations in $w(x)a$ causal past (see def. 2.3.4) have not been yet received. When these last arrived, they are simply discarded because overwritten. Notice that the approach considered in Chapter 3 is not reasonable in such environment, since messages may never arrive and then waiting for them will block the computation.

Moreover, in order to guarantee persistency of value written in quiescent periods, an operation invoked by a client process c_i, finishes when c_i has received a response from f distinct object entities. We assume that h object entities are eventually incarnated by a correct object manager process.

Formally,

Assumption 5.2.1 *There are h object entities o_i, whose corresponding $\widehat{o_i}$ is finite and such that both the following conditions are satisfied:*

i) $2n - h < 2f$

ii) $f \leq h$.

where $\widehat{o_i}$ is the sequence of object manager processes incarnating object entity o_i.

Finally, a correct implementation has to satisfy the following properties:

Definition 5.2.1 (Termination) *If a correct client process c_i invokes an operation, then c_i eventually returns from the invocation.*

Definition 5.2.2 (Validity) *If a read operation invoked by a client process c_i returns a value v, then there exists a client process c_j that invoked the write of v.*

The validity property points out that if a read operation returns a value v then the write operation $w(x)v$ has been invoked by some process in the system[1].

[1] Any write operation whose value has been read by some process is supposed to appear in the corresponding history.

5.2.1 Data Structures

Each client process c_i manages:

• `ack[1..n]`: a vector of boolean, one for each object entity. Each entry is initially set to false. It is used to track when f object entities have answered to a read request made by c_i. $ack[k] = true$ means that c_i has received from o_k a response to its current read request;

• `ack`: an integer initially set to 0. It stores the number of acks received by c_i from object entities in order to track when an ack is received by f object entities.

Each object manager o_i has to manage a variable `last`, to track the identity of the client process that issued the last write operation executed at o_i. This information is used to ensure read legality w.r.t. the causality order relation. In addition, processes in the system, both clients and object managers, have to manage a timestamp system to implement a plausible clock t,[10]. The plausible clock system we propose is an adaptation of *R-Entries vector clock system (REV)* proposed by Ahamad et al. in [10]. Each process stores a vector of integers of fixed size l, initially set to $[0, \dots, 0]$. This vector is denoted $t_i[1..l]$ for a client process c_i and $to_i[1..l]$ for an object manager o_i. Each client process c_i is associated to the (i *modulo* l) entry of the plausible clock t. According to this and due to the fact that the number of client processes in the system may be more than l at a given point in time, several clients may share the same plausible clock entry.

Rules to manage t_i/ to_i:

R1 Each time a process sends a message, it timestamps the message m with the current value of its plausible clock, denoted $m.t$.

R2 Each time a client c_i writes, it increments its plausible clock entry $t_i[i \textit{ modulo } l]$, i.e. $t_i[i \textit{ modulo } l] := t_i[i \textit{ modulo } l] + 1$.

R3 Each time a client c_i receives a response message m to a read request, it updates its plausible clock with the timestamp piggybacked by m, i.e. $\forall\ k\ t_i[k] := max(t_i[k], m.t[k])$.

R4 Each time an object manager receives a write request message m from c_j, if $to_i[j \textit{ modulo } l] < m.t[j \textit{ modulo } l]$ then it updates its plausible clock to_i, i.e. $\forall\ k\ to_i[k] := max(to_i[k], m.t[k])$.

Plausible clocks accuracy Let us point out that the size of the plausible clock is independent of the number of clients and of object entities in the system. On the other hand, the ratio between plausible clock size and the number of client processes impacts on the ability of t to precisely track causality order relations[2]. Ahamad et al. in [10] have presented simulation results relating the accuracy and the size of plausible clocks. They have shown that a small number of plausible clocks entries are sufficient to correctly capture a large number of causality relations.

5.2.2 Protocol Behavior

When a client c_i wants to execute a write operation $w_i(x)v$, it increments its entry of the plausible clock t_i and sends an update message corresponding to $w_i(x)v$, namely $m_{write}(v,t)$, to all object entities. An update message $m_{write}(v,t)$, later sometimes referred as *write message*, contains the value v to be written and the value t of the plausible clock at c_i at the time the message was sent.

```
WRITE($v$)
1 $t_i[i \texttt{ modulo } l] := t_i[i \texttt{ modulo } l] + 1$;
2 repeat
3    for $(1 \leq j \leq l)$ send $[m_{write}(v,t)]$ to $o_j$
4 until [receipt($ack_{m_{write}(v,t)}$) from $f$ $o_j$];
5 $cache := v$
```

Figure 5.1: Write procedure performed by client process c_i

Moreover, because of fair-loss links, client process c_i repeatedly sends $m_{write}(v,t)$ to all object entities until an ack is received from f object entities, lines 2, 3 and 4 of write procedure in Figure 5.1. In this way, when $w_i(x)v$ completes, at least f object entities have received $m_{write}(v,t)$. The value written is then stored in c_i's cache, line 5 of write procedure in Figure 5.1.

When a client c_i wants to read, it repeatedly sends its read request to all object entities until responses are collected from f different object entities, lines 2-13 of read procedure in Figure 5.2. A message $m_{read}(num_{seq},\ t, cache)$, corresponding to a read, later sometimes referred as *read message*, contains the sequence number of the request, num_{seq}, the current value of the plausible clock at c_i, namely t, and the current value of c_i's cache. Due to network

[2]Given two operations o_1 and o_2 and a plausible clock system t, then $o_1 \mapsto_{co} o_2$ implies $o_1.t < o_2.t$. On the other hand, if $o_1.t < o_2.t$, one of the following cases arises $o_1 \mapsto_{co} o_2$ or $o_1||_{co}o_2$.

delays and retransmission, num_{seq} is necessary to allow a client to discard old responses when received.

```
READ(x)
1  $num_{seq} := num_{seq} + 1$;
2  while $(ack < f)$
3    repeat
4      for $(1 \leq j \leq n)$   send $[m_{read}(num_{seq}, t, cache)]$ to  $o_j$
5    until [receipt($m_{res}(num_{seq},\ to_j,\ x^j)$) from $o_j$ with $j \in [1..n]$];
6    if $(ack[j] = false)$ then
7                          $ack[h] := true$;
8                          $ack := ack + 1$;
9                          if $(to_j \neq t)$ then
10                                   $cache := x^j$;
11                                   $\forall\ k\ t_i[k] := max(t_i[k], to_j[k])$;
12                                   $ack := f$;
13                         end  if
14   end  if
15 end  while
16 $ack := 0$;
17 for $(1 \leq j \leq n)$   $ack[j] := false$;
18 return($cache$)
```

Figure 5.2: Read procedure performed by client process c_i

In detail, due to fair-loss links client c_i repeatedly sends a read request to all object entities until a response is received, lines 3, 4, 5 of read procedure in Figure 5.2. When a response is received from o_j, c_i checks if it already received a response concerning the current request from o_j, line 6 of read procedure in Figure 5.2. If not, the message is processed by c_i: it tracks an ack more, line 8 of read procedure in Figure 5.2; it checks if the value piggybacked is a new one w.r.t. the one in c_i's cache and if so c_i updates its cache and its control structures, lines 9-12 in Figure 5.2. c_i stops to send such a request when one of the following conditions holds: it has received a response from f distinct object entities or it has received a value different from the one stored in its cache. Finally, the value is returned, line 18 of read procedure in Figure 5.2.

When an object manager o_i receives a write request from a client c_j, o_i checks if the write operation has to be considered obsolete w.r.t. $\mapsto_{co}$, line 2 of write thread in Figure 5.3. If so, o_i discards the message otherwise it applies the value to its local copy of x, namely x^i, and it synchronizes its plausible clock with the one piggybacked by the *write message*, lines 3, 4 and 5 of write thread in Figure 5.3. The variable *last* stores the identifier of the plausible clock entry that was last updated. In any case, it sends back an ack to client process c_j, line 7 of write thread in Figure 5.3.

When an object manager o_i receives a read request by client process c_j, it has to check causal consistency. If the value of x^i causally precedes the one in

```
1  when (receipt($m_{write}(v, t)$) from $c_j$) do
2    if (($t$[$j$ modulo $l$] > $to_i$[$j$ modulo $l$]))
3      then $x^i := v$;
4           $\forall\ k \quad to_i[k] := max(to_i[k], t[k])$;
5           $last := j$ modulo $l$;
6     end if
7     send [$ack_{m_{write}(v,t)}$] to $c_j$
```

Figure 5.3: o_i's write thread

c_j read request, o_i simply sends back a response with the content of c_j current read request, line 4 of read thread in Figure 5.4. [3] Otherwise, it replies with its local value of x and the value of its plausible clock, line 3 of read thread in Figure 5.4. It is important to remark that, an object entity o_i always answer to a read request of a client process c_j.

```
1  when (receipt($m_{read}(num_{seq}, t, val)$) from $c_j$) do
2    if ($tx_i[last] > t[last]$)
3      then send [$m_{res}(num_{seq}, tx_i, x^i)$] to $c_j$
4      else send [$m_{res}(num_{seq}, t, val)$] to $c_j$
```

Figure 5.4: o_i's read thread

A response message $m_{res}(num_{seq}, tx_i, x^i)$ for a read contains: i) the sequence number of the read request ii) the value of o_i's plausible clock at response time, tx_i, and iii) the value x^i to be returned by the read operation.

Figure 5.5 depicts a simple scenario, where client process c_2 writes the value a subsequently read by another client c_1. In Figure 5.5 object entities values are depicted every time they change and since in this scenario we do not consider memory losses, a value stored is not lost. Notice that, the scenario in Figure 5.5 also shows that some messages may be lost because of fair-loss links.

The proposed protocol is quiescent, i.e. eventually no process sends or receives messages [4], due to assumption 5.2.1 and the fact that after f responses a process stops to send the corresponding request message.

Let us notice that our implementation is blocking. This is necessary to implement what we have defined a weakly-persistent object. However, the non-blocking property of traditional causal consistency or not uniform one, is also sacrificed to hide the complexity and unreliability of high dynamic systems beside by considering the memory implemented through a fixed number of

[3]It must be noted that in such a case, a refresh purpose might be considered for a read operation, that is the object manager could update its local structure, logical clock and cache, treating the read as a write. This may improve the availability of written values.

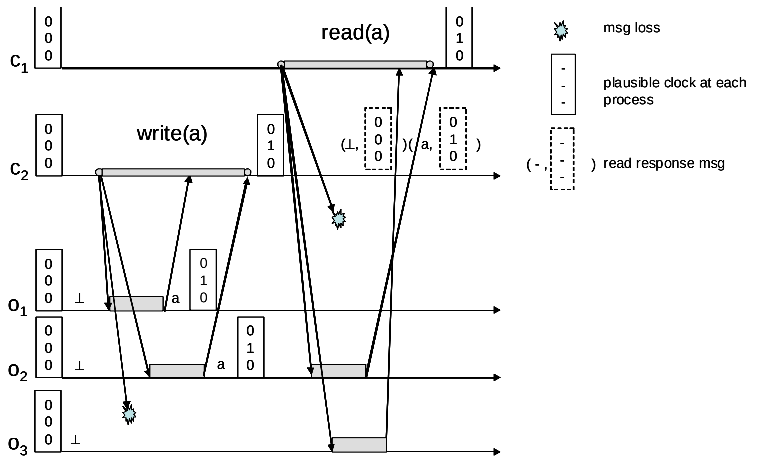


Figure 5.5: A scenario generated by the implementation of the causal object described in Section 4 where n=h=3 and f=2

virtual servers.

5.3 Correctness Proofs

In this section we first prove that t is a plausible clock capturing $\mapsto_{co}$ and then we prove the correctness of the algorithm we present in section 5.5.2 to implement a weakly-persistent causal object.

5.3.1 t is a Plausible Clock capturing $\mapsto_{co}$

Given a write operation $w_i(x)v$, according to line 2 of write procedure in Figure 5.1, such a write operation is associated with a logical clock t, denoted $t.w_i(x)v$. We have to prove that given two write operations $w_i(x)v$ and $w_j(x)v'$ such that $w_i(x)v \mapsto_{co} w_j(x)v'$, then $t.w_i(x)v < t.w_j(x)v'$. On the other hand, according to the properties of plausible clocks, $t.w_i(x)v < t.w_j(x)v'$ means that one of the following case arises: 1) $w_i(x)v||w_j(x)v'$ or 2) $w_i(x)v \mapsto_{co} w_j(x)v'$.

Notation $w \mapsto_{co}^{k} w'$ with $k \geq 1$ means that there exists a sequence of $k \mapsto_{co}$ relations $w \mapsto_{co} w_1 \mapsto_{co} \dots w_h \mapsto_{co} w_{h+1} \mapsto_{co} \dots w_{k-1} \mapsto_{co} w_k \mapsto_{co} w'$ 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 3 *At each client c_i, t_i does not decrease.*

Observation 4 *w is the k^{th} write operation invoked by the client process c_i $\Rightarrow t[i\ modulo\ l].w(x)v \geq k$.*

Proof Let $w(x)v$ be the k^{th} write operation invoked by the client process c_i. Two possible cases arise:

1) at the time of $w(x)v$ invocation, c_i has not yet executed a read operation. Thus $t[i\ modulo\ l].w(x)v$ is equal to k according to line 1 of write procedure in Figure 5.1 and the fact that the initial value of the plausible clock at c_i is $[0, .., 0]$.

2) at the time of $w(x)v$ execution, c_i has executed at least one read operation. There are two possible cases: i) c_i reads a value written by itself, thus it does not update its plausible clock and we are again in case 1); ii) c_i reads a value written by another client process. According to line 11 of read procedure in Figure 5.2, c_i synchronized its clock t with to, the one sent by the object manager in its response to such a read request, lines 2,3 of read thread in Figure 5.4. Moreover, for line 11 of the read procedure in Figure 5.2, the resulting value of t is not minor than the value of t before such a synchronization. Thus, since $w(x)v$ is the k^{th} write executed by c_i and due to line 1 of write procedure in Figure 5.1 and to the fact that when a client reads, its plausible clock does not decrease, we have that $t[i\ modulo\ l].w(x)v \geq k$.

Now to prove that t is a plausible clock capturing $\mapsto_{co}$, we have to prove that: $\forall\ w_i(x)v,\ w_j(x)v' :\ w \neq w',\ w_i(x)v \mapsto_{co} w_j(x)v' \Rightarrow t.w_i(x)v < t.w_j(x)v'$.

Lemma 5.3.1 $\forall\ w_i, w_j\ \in H :\ w_i \neq w_j, (w_i \mapsto_{co} w_j \Rightarrow t.w_i < t.w_j)$

Proof *Let us consider the notation $w_i \mapsto^k_{co} w_j$. The proof is by induction on the value of k.*

Basic step. *Given two write operations w_i and w_j such that $w_i \mapsto^0_{co} w_j \Rightarrow t.w_i < t.w_j$. This means that $w_i \mapsto_{co} w_j$ and $\nexists$ a write w' such that $w_i \mapsto_{co} w'$ and $w' \mapsto_{co} w_j$.*

We distinguish two cases:

(1) $i = j$. This means that w_i and w_j have been executed by the same client process c_i. Each time a client process executes a write operation, it performs the write procedure in Figure 5.1. According to line 1 of Figure 5.1, each time c_i executes a write operation, it increments its corresponding entry of t. Due to Observation 3, if w_i precedes w_j in c_i program order then $t.w_i < t.w_j$. Therefore the claim follows.

(2) $i \neq j$. There exists a read operation invoked by the client process c_j, denoted $r_j(x)$, such that $w_i(x)v \mapsto_{ro} r_j(x)v$ and $r_j(x)v \mapsto_{po} w_j(x)v'$. In detail,

c_j can read the value written by c_i because i) c_i has invoked $w_i(x)v$ and at least a majority of object managers have applied $w_i(x)v$ and ii) one of such object managers has answered to c_j read request. Without loss of generality, let us assume that o_k is the object manager that implements points i) and ii). Then, according to line 4 of the write thread in Figure 5.3, after having applied $w_i(x)v$, tx_k is $\geq$ than $t.w_i(x)v$. Subsequently, c_j reads the value written by $w_i(x)v$. This means that:

- ❖ *when o_k has received the read message m of c_j, its local value of x was v, that is the value written by $w_i(x)v$. Then according to line 4 and 5 of write thread in Figure 5.3 and to lines 2, 3 of read thread in Figure 5.4, o_k sends to c_j a response message $m_{res}(num_{seq}, v, to_k)$ with $to_k \geq t.w_i(x)v$.*

- ❖ *when c_j delivers $m_{res}(v, to_k, num_{seq})$ according to lines 10, 11, 12, 2 and 18 of read procedure in Figure 5.2, c_j updates its t_j and its cache with the corresponding values piggybacked by $m_{res}(num_{seq}, v, to_k)$ and then it returns the value to be read, that is v.*

Then after the read operation $t_j \geq to_k$ that is $t_j \geq t.w_i(x)v$. Moreover, it must be noted that i) for observation 3, t_j never decreases and ii) when c_j writes $w_j(x)v'$, t_j is incremented, (line 1 of write procedure in Figure 5.1). Then since $w_j(x)v'$ is executed by c_j after the execution of $r_j(x)v$ the claim follows, that is $t.w_i(x)v < t.w_j(x)v'$.
Inductive Step. *$w_i \mapsto_{co}^{k>0} w_j$ then: (i) $\exists\ w' : w_i \mapsto_{co}^{k-1} w'$. By the inductive hypothesis we have: $t.w_i < t.w'$, and (ii) $w' \mapsto_{co}^{1} w_j$. Because of* Basic Step *$t.w' < t.w_j$. From (i) and (ii), it follows: $t.w_i < t.w_j$.*

5.3.2 Object Correctness Proofs

Property 5.3.2 (Causal Ordering) *Given two write operations $w(x)v$ and $w(x)v'$ if $w(x)v \mapsto_{co} w(x)v'$, then a client process c_i that reads both values, executes $r_i(x)v$ and then $r_i(x)v'$.*

Proof Roughly speaking, we have to prove that given two write operations $w(x)v$ and $w(x)v'$ if $w(x)v \mapsto_{co} w(x)v'$, then a client process c_i that reads both values, reads v and then v'. Thus, let us assume that a client c_i has executed $r_i(x)v'$. This means that for lines 4-6 of write thread in Figure 5.3 and line 11 of read procedure in Figure 5.2, the logical clock of c_i after the execution of the read is $t_i \geq t.w(x)v'$. Then, when subsequently c_i invokes another read operation, for what said and for observation 3, c_i inserts in the corresponding request message a timestamp $t_i \geq t.w(x)v'$. By contradiction, assume that there is an object manager x_k that responds to that request with

$m_{res}(v, to_k)$, then according to lines 2, 4 and 5 of write thread in Figure 5.3 and line 3 of read thread in Figure 5.4, $to_k[last] = t.w(x)v[last]$. But for lemma 5.3.1 $w(x)v \mapsto_{co} w(x)v'$ implies $t.w(x)v < t.w(x)v'$. This means that $\forall\ k\ t.w(x)v[k] \leq t.w(x)v'[k]$. This contradicts line 2 of read thread in Figure 5.4. Thus when o_k receives the read request of c_i with timestamp $t.w(x)v'$ it sends back the value of c_i's previous request, that is v' line 4 of read thread in Figure 5.4.

Property 5.3.3 (Validity) *If a read operation invoked by a client process c_i returns a value v, then there exists a client process c_j that invoked the write of v.*

Proof The proof follows by lines 1, 3 of write thread in Figure 5.3, lines 3-5, 10 and 18 of read procedure in Figure 5.2, to the read thread in Figure 5.4 and to the property of *no creation* of fair loss channels, [54].

Lemma 5.3.4 (Legal read) *Given a read operation $r(x)v$ belonging to H, then i) there must exist a write operation $w(x)v \in H$ such that $w(x)v \mapsto_{co} r(x)v$ and ii) there must not exist an operation $o(x)v' \in H$ such that $w(x)v \mapsto_{co} o(x)v'$ and $o(x)v' \mapsto_{co} r(x)v$*

Proof In the following we prove points i) and ii).
i) By the validity property and proposition 5.3.3, there exists a write operation $w(x)v$ in the history. Moreover, $w(x)v \mapsto_{co} r(x)v$ by definition of read-from order relation and of causality order relation.
ii) We have to prove that it may not exist an operation $o(x)v' \in H$ such that $w(x)v \mapsto_{co} o(x)v'$ and $o(x)v' \mapsto_{co} r(x)v$. We have two possible cases $o(x)v' = r(x)v'$ or $o(x)v' = w(x)v'$.

- ❖ $[o(x)v' = r(x)v']$. By contradiction let us assume that $w(x)v \mapsto_{co} r(x)v' \mapsto_{co} r(x)v$. Because of i) there exists a write operation $w(x)v' \mapsto_{co} r(x)v'$. So, if $w(x)v \mapsto_{co} w(x)v'$ for proposition 5.3.2 $r(x)v \mapsto_{co} r(x)v'$. Contradiction. If $w(x)v' \mapsto_{co} w(x)v$ for proposition 5.3.2 $r(x)v' \mapsto_{co} r(x)v$. Contradiction.

- ❖ $[o(x)v' = w(x)v']$. So $w(x)v \mapsto_{co} w(x)v' \mapsto_{co} r(x)v$. Let us remember that if $w(x)v \mapsto_{co} w(x)v'$ then for 5.3.1 $t.w(x)v < t.w(x)v'$.

  If $w(x)v' \mapsto_{co} r(x)v$ then 1) $w(x)v'$ and $r(x)v$ are issued by the same process ap_i and $w(x)v'$ is executed before $r(x)v$ or 2) there must exists a read operation $r(x)v'$ such that $w(x)v' \mapsto_{co} r(x)v' \mapsto_{co} r(x)v$.

  Let us consider only case 1) because case 2) has been proved in the first point of such proof. When process ap_i issued its write operation $w(x)v'$

$t.w(x)v < t.w(x)v'$. Then when it later requests to read, for observation 3, its read message piggybacks a timestamp t such that $t.w(x)v' \leq t$. According to line 2 of read thread in Figure 5.4, ap_i may not received a response message with a value v associated to a write operation $t.w(x)v$ such that $t.w(x)v < t$.

Lemma 5.3.5 (Weakly-Persistent Object) *The algorithm implements a weakly-persistency object if the following condition holds:* $2n - h < 2f$.

Proof If a value v is written infinitely many times, than the value v due to assumption 5.2.1, lines 2-4 of write procedure in Figure 5.1, line 1 of write thread in Figure 5.3 and the properties of the plausible clock systems (lemma 5.3.1), v or a value that is causally concurrent with v or a more recent one is permanently stored. This means that a client process that reads infinitely many times, will read one of such values due to lines 2-12 of read procedure in Figure 5.2 and to line 2 and 4 of read thread in Figure 5.4.

We have to prove that given n object entities, if a value is stored by f object entities, provided that h object entities do not suffer memory losses, the value written may be retrieved if not overwritten. Let us consider the case in which there are no concurrent or more recent value written w.r.t. $\mapsto_{co}$.

When the write $w(x)v$ terminates, at least f object entities have stored the value, for line 4 of write procedure in Figure 5.1, line 2 of write thread in Figure 5.3 and for the properties of the plausible clocks and the assumption of no causally concurrent or more recent write operation. Among these, at most n-h may lose its status and thus value v, returning to the initial value $\perp$.

In this sense, let us consider the worst case: n-f object entities do not store the value v, n-h object entities store and subsequently lose value v and the remaining object entities permanently store such a value.

When subsequently a client process c_i invokes a read request, it waits for a response from f object entities. In the worst case, c_i receives a response from the $2n$-f-h object entities that do not have value v. But since it waits for f responses, we are sure that there is at least one response piggybacking value v if $2n - f - h < f$, that is $2n - h < 2f$. In other words, in order to reach f responses, c_i needs a response sent by an object entity that does not belong to the $2n - f - h$ object entities which do not store value v.

Property 5.3.6 (Termination) *Each operation invoked by a correct client eventually completes if* $f \leq h$.

Proof • `Write.` Let c_i be a correct client that issues a write operation $w_i(x)v$. Then according to line 3 of write procedure in Figure 5.1, $w_i(x)v$ completes when c_i receives an ack from f object entities, otherwise it loops into lines 2 and 3 of write procedure in Figure 5.1. Then we have to prove that if a correct client c_i invokes a write $w_i(x)v$ eventually f $ack_{m_{write}(v,t)}$ are received by c_i. This is ensured by line 6 of write thread in Figure 5.3 and by assumption 5.2.1 provided that $f \leq h$, that is the number of responses the process waits for, is at most equal to the number of object entities that after some point in time are incarnated by correct object manager processes.

• `Read.` Let us now consider the case of a read operation. A read operation completes if f response messages are received, lines 2, 4,and 8 of read procedure in Figure 5.2. Then we have to prove that if a correct client c_i invokes a read $r_i(x)v$ eventually an ack from f o_k is received by c_i. Provided that $f \leq h$, this is ensured by assumption 5.2.1 and lines 3, 4 of read thread in Figure 5.3.

Theorem 5.3.7 (Weakly-Persistent Causal Object) *The algorithm implements a weakly-persistency causal object if the following condition holds:* $2n - h < 2f$.

Proof The proof follows by lemma 5.3.4 and the fact that it applies to every read operation in a given history and to every history and by lemma 5.3.5.

Finally, let us point out that, as an example, putting $h = \lceil (2n+1)/3 \rceil$ and $f = \lceil 2n/3 \rceil$, the algorithm presented in section 5.2 implements a weakly persistent causal object.

5.4 Related Work

Read/write objects are building blocks to implement several distributed services, i.e. distributed shared memory, distributed directory lookup services, shared boards and so on.

In a distributed message-passing system where processes may fail by crashing, since requesting object state persistency, atomic objects implementations have to cope with the difficulty of providing object state continuity when processes fail. Attiya et al. in [11] propose an implementation for single-writer/multiple-reader register provided that a majority of processes do not crash. Lynch et al. in [56], extend this last work to multiple-writer/multiple-reader registers adopting a more general quorum-based approach. Their solution also tolerates quorums on-line reconfigurations. Some quorum-based solutions have been also proposed to implement atomic objects in dynamic systems where participants may join, leave and crash during the computation, [37, 57], [33]. Instead of using quorums, in [31] Friedman implements an atomic object on top of a virtually synchronous communication layer. In [31], Friedman also investigates sequential and causal consistent shared objects.

On the other hand, in dynamic systems where processes may join and leave at any time and arbitrarily fast, objects implementations are not persistent by nature. To circumvent this problem, Lynch et al. [55] propose a solution to implement atomic consistency when the system is quiescent. Friedman et al. in the context of peer-to-peer systems propose what they call a *semi-reliable unified storage* abstraction [32]. Interestingly they implement a notion of atomic consistency restricted to uninterrupted partial execution. An uninterrupted partial execution is a collection of sequences of read and write operations, each one by a different process, such that during their execution there are no failure and the set of processes does not change. On the contrary, we guarantee read legality all the time regardless the dynamism of processes, while we guarantee persistency only of value written during quiescent periods. We called such an object weakly-persistent causal object.

To cope with the complexity of dynamic systems, we exploit the idea proposed by Chen et al. in [26] to solve fault-tolerant mutual exclusion problem in dynamic systems. In detail, the object is implemented by a fixed set of virtual servers that may suffer memory losses. A memory loss abstracts the fact that a virtual server is incarnated by a process that may crash and be replaced by a new process that is not able to retrieve any state the crashed process pass through. It is like considering a fixed set of servers that may crash and recover but such that after recovering completely lose their previous state. Guerraoui et al. in [39], point out that atomic registers may by implemented in a crash-recover model provided that i) a majority of processes never crash or eventually recover and never crash again and that ii) given a write operation $w(x)v$, at least a majority of processes log (i.e. store to stable storage) the value v before the write operation returns. Thus, they extend the atomicity consistency criteria defined for multi-writer/multi-reader register in a crash-stop model by providing two new criteria: *persistent atomicity*, to capture the fact that traditional atomicity has to persist through the crashes and *transient atomicity* that does not guarantee atomicity in between crashes.

Finally, in order to track causality order relations between operations, we implement a plausible clock system that is an adaptation of *R-Entries vector clock system(REV)* proposed by Ahamad et al. in [10]. Plausible clocks were also used by Ram et al. in [62] to implement a causal memory in a mobile environment. Their system model, however, differs from our since they consider a fixed set of correct physical master sites and a set of mobile hosts.

Moreover, let us remark that the causal object implemented in section 5.2, is persistent (or weakly-persistent) provided that h object entities are persistent (weakly-persistent), i.e.(eventually) incarnating by correct object manager processes. In this sense, since each object entity may be view as a causal object, we can compare our work with the idea proposed by Afek et al. in [1]. They study how implement a correct object of a specified type through a set of objects of the same type, assuming that at most k of such objects may be k-faulty (where k may be ∞). An object is k-faulty if it may suffer at most k faults. In our case $k = \infty$. Finally, Jayanti et al. in [44] introduce the concept of *graceful degradation*. Roughly speaking, this property states that if base objects only fail by a particular class of failure, then the derived object does not fail more severely than its base objects. The implementation proposed in section 5.2 presents this property, i.e. if more than $n - h$ base objects are not (weakly)persistent than the derived object is not (weakly)persistent. Nothing worst could happen.

5.5 Weakly-Persistent Causal Memory Implementation

In this section, we extend the algorithm proposed in section 5.2 to implement a weakly-persistent causal object, in order to implement a weakly-persistent causal memory. In addition to considerations and assumptions made for a single object, we consider that each object entity incarnates all the shared memory $\mathcal{X}$=$\{x_1, x_2, ...x_m\}$ and each client process caches a copy of the entire memory. We denote as x^i_j the local copy of variable x_j at o_i.

Since write semantics is meaningful only with respect to operations made on a same variable, we need to extend the previous protocol maintaining mutual consistency w.r.t. $\mapsto_{co}$ among the copies of variables cached at each client process (i.e. "it is possible that their values coexisted at some time in a distributed computation" [73]). Moreover, each time a client process c_i issues a write operation on variable x, the corresponding write message has to piggyback the values of all variables cached at c_i, that is a causal consistent view of the entire memory.

5.5.1 Data Structures

In order to guarantee read legality w.r.t. $\mapsto_{co}$, processes in the system, both clients and object managers, have to manage a timestamping system to implement a plausible matrix clock MC, later simply referred as matrix clock. The matrix clock system we propose is an extension of plausible clock system introduced in section 5.2.1. Each process stores a matrix of integers of fixed size $m * l$, where m is the number of shared variables belonging to $\mathcal{X}$ and l is a parameter chosen independently of the number of processes and variables in the system. Each entry of MC is initialized to 0. The above said matrix is denoted MC_i for a client process c_i and MCo_i for an object manager o_i. In particular, $MC[h][*]$ denotes the plausible clock related to variable x_h. Each client process c_i is associated to the $(i\ modulo\ l)$-th column of the matrix clock, that is $MC[*][i\ modulo\ l]$. The same consideration made in section 5.2.1 on the size and the ability to track causality order relations of plausible clocks, may be applied to the above stated matrix clock system.

Rules to manage $MC_i/\ MCo_i$:

R1 Each time a process sends a message, it timestamps the message m with the current value of its matrix clock, denoted $m.MC$.

R2 Each time a client c_i executes a write operation on variable x_h, it increments its matrix clock entry corresponding to x_h, namely $MC_i[h][i\ modulo\ l]$, i.e. $MC_i[h][i\ modulo\ l] := MC_i[h][i\ modulo\ l] + 1$.

R3 Each time a client c_i receives a response message m to a read request, for each $h \in \{1, \dots, m\}$, it updates $MC[h]$ with the corresponding entry of the matrix clock piggybacked by m, i.e. $\forall\ h, k\ MC_i[h][k] := max(MC_i[h][k], m.MC[h][k])$.

R4 Each time an object manager receives from c_j a write request message m on variable x_h, if $MCo_i[h][j\ modulo\ l] < m.MC[h][j\ modulo\ l]$ then for each h such that $MCo_i[h][j\ modulo\ l] < m.MC[h][j\ modulo\ l]$ it updates its plausible clock $MCo_i[h]$, i.e. $\forall\, k\ MCo_i[h][k] := max(MCo_i[h][k], m.MC[h][k])$.

In addition, each client process c_i has to manage a vector of values V_i of fixed size m to locally cache the values of all shared variables. $V_i[j] = v$ means that the value of variable x_j locally cached at c_i is v.

5.5.2 Protocol Behavior

When a client c_i wants to execute a write operation on variable x_h, namely $w_i(x_h)v$, it increments its corresponding entry of its matrix clock MC_i, $MC_i[h][i\ modulo\ l]$ and sends an update message to every object entity. A message $m_{write}(x_h, V, MC)$ corresponding to a write operation, later sometimes referred as *write message*, contains the variable to update, the value $V[h]$ to be written and the value MC of c_i matrix clock at the time the message was sent. Notice that the message also contains the value of all other variables in the memory, i.e. $V[j]\ \forall\ j \in \{1, \dots\ m\}$. This is necessary to maintain consistency among different variables.

In detail, according to the considered system model, it is not reasonable to constraint the delivery of a message to the delivery of another one, since such message may never arrive. Moreover, a process that executes a write operation on variable x_h, relates such written value to the ones it has previously read or written. So, to maintain consistency it is necessary that, when writing each client process sends all the content of the shared memory locally cached.

```
WRITE($x_h, v$)
1   $MC_i[h][i$ modulo $l] := MC_i[h][i$ modulo $l] + 1$;
2   $V[h] := v$;
3   repeat
4      for $(1 \le j \le n)$   send $[m_{write}(x_h, V, MC)]$ to  $o_j$
5   until [receipt($ack_{m_{write}(x_h,V,MC)}$) from  $f$  $o_j$];
```

Figure 5.6: Write procedure performed by client process c_i

When a client c_i wants to execute a read operation on variable x_h, namely $r_i(x_h)v$, it sends a message to all object entities. Analogously to the case of a single object (section 5.2), the read message also contains the values of variables cached at c_i. In such a way, every object entity o_j is able to reply to the request ensuring that eventually an operation will finish, i.e. if the value of the shared variable locally stored at o_j is causally precedent the one at c_i, o_i simply replies sending back to c_i its request message.

When client c_i receives the response message piggybacking a new value with respect to its cache, it has to maintain mutual consistency among the cached variables. In detail, for each variable x_k, c_i has to check if the corresponding value in its cache, namely $V[k]$, causally precedes the one piggybacked by the response message. In this case it has to update its $V[k]$ (lines 11-14 of read procedure in Figure 5.7). Finally, c_i updates each entry of its matrix clock through the component wise maximum operation, line 15 of read procedure in Figure 5.7.

When an object manager o_i receives a read request on variable x_h by a client process c_j, it first checks causal consistency. If the value of x_h at o_i is causally precedent the one piggybacked by the request message of the client, o_i simply sends back to c_j the values in its current read request, line 4 of read thread in Figure 5.8. Otherwise, it replies with its local value of x_h and of every other variable in the shared memory. The response message also contains the current value of o_i matrix clock, line 3 of read thread in Figure 5.8. Let us notice that the value sent by o_i to c_j may be the one sent by c_j itself in the read request or a more recent one w.r.t. $\mapsto_{co}$.

```
READ($x_h$)
$num_{seq} := num_{seq} + 1$;
while $(ack < f)$
  repeat
    for $(1 \leq j \leq n)$   send $[m_{read}(num_{seq}, x_h, MC, V)]$ to $o_j$
  until [receipt($m_{res}(num_{seq},\ MCo_j,\ [x_1^j, x_2^j, \ldots, x_m^j])$) from $o_j$ with $j \in [1..n]$];
  if $(ack[h] = false)$ then
                    $ack[h] := true$;
                    $ack := ack + 1$;
                    if $(MCo_j \neq MC)$ then
                                   $V[h] := x_h^j$;
                                   for each $s \in (1, \ldots, m)$
                                     if $(MCo_i[s] > MC[s])$ $and$ $(s \neq h)$
                                     $V[s] := x_s^j$;
                                     end if
                                        $\forall\ k$   $MC[s][k] := max(MC[s][k], MCo_i[s][k])$;
                                   end for
                                   $ack := f$;
                    end if
  end if
end while
$ack := 0$;
for $(1 \leq j \leq n)$   $ack[j] := false$;
return($V[h]$)
```

Figure 5.7: Read procedure performed by client process c_i

```
when $(receipt(m_{read}(num_{seq}, x_h, MC, V))\ from\ c_j)$ do
  if $(MCo_i[h][last] > MC[h][last])$
    then send $[m_{res}(num_{seq}, MCo_i, [x_1^i, x_2^i, \ldots, x_m^i])]$ to $c_j$
    else send $[m_{res}(num_{seq}, x_h, MC, V)]$ to $c_j$
```

Figure 5.8: o_i's read thread

When receiving an update message $m_{write}(x_h, V, MC)$, an object entity o_i has to check legality. If o_i has not already applied a write operation on x_h^i that is more recent than the one piggybacked by the message with respect to $\mapsto_{co}$ (line 2 of write thread in Figure 5.9), it updates its local structures. In particular, it stores value $V[h]$ in x_h^i (line 3) and then if necessary, o_i updates the rest of the memory to guarantee consistent values among variables, lines 6-8 in Figure 5.9. Let us notice that o_i always updates each entry of its matrix clock in order to track all causality order relations related to the applied write operation.

```
1   when $(receipt(m_{write}(x_h, V, MC))\ from\ c_j)$ do
2     if $((MC[h][j \texttt{ modulo } l] > MCo_i[h][j \texttt{ modulo } l]))$
3       then $x_h^i := V[h]$;
4            $last := j \texttt{ modulo } l$;
5            for each $s \in (1, \ldots, m)$ with $s \neq h$
6                if $(MC[s] > MCo_i[s])$
7                    $x_s^i := V[s]$;
8                end if
9                    $\forall\ k \quad MCo_i[s][k] := max(MCo_i[s][k], MC[s][k])$;
10           end for
11    end if
12    send $[ack_{m_{write}(x_h,V,MC)}]$ to $c_j$
```

Figure 5.9: o_i's write thread

Figure 5.10 depicts a functioning scenario that generates the following history: $h_1 :\ w_1(x)a;\ w_1(y)b \quad h_2 :\ w_2(x)c;\ r_2(y)b \quad h_3 :\ r_3(y)b;\ r_3(x)a$. The shared memory is composed of two variables x and y. It is implemented by 3 object entities, each of which is eventually incarnated by a correct object manager. For sake of clarity, in figure 5.10 we do not depict the response messages necessary to complete the operations.

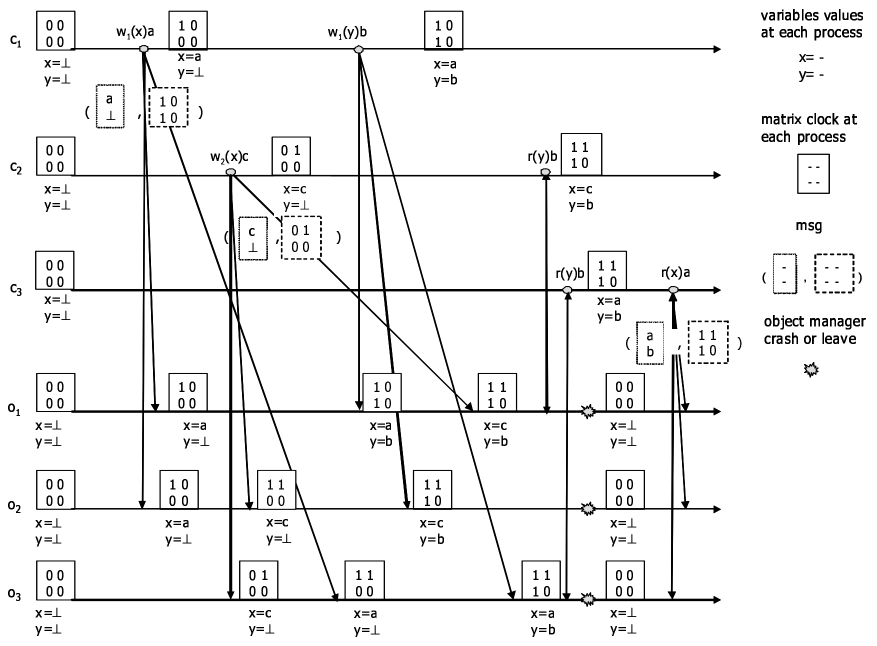

c1
c2
c3
o1
o2
o3
w1(x)a
w1(y)b
w2(x)c
r(y)b
r(x)a
variables values at each process
x= -
y= -
matrix clock at each process
msg
object manager crash or leave

Figure 5.10: A scenario generated by the implementation of the weakly persistent causal consistent memory described in Section 5.5.

5.5.3 Correctness Proofs

Correctness proofs regarding persistency, validity and termination may be trivially derived by the ones presented in section 5.3.2 for the case of a single variable x. So, in this section we simply prove that read operations made on different variables return causal consistent values.

Observation 5 *At each client c_i, MC_i does not decrease.*

Given a write operation w, let us denote as $w.MC$ the matrix clock associated to w.

Lemma 5.5.1 $\forall\ w_i, w_j\ \in H:\ w_i \neq w_j, (w_i \mapsto_{co} w_j \Rightarrow MC.w_i < MC.w_j)$

We omit the proof, since it may be simply deduced making the same reasoning done in lemma 5.3.1.

Finally,

Property 5.5.2 *Let $w_i(x_h)v$ and $w_j(x_k)v'$ be two write operations such that $w_i(x_h)v\ \mapsto_{co}\ w_j(x_k)v'$ with $x_h \neq x_k$. If a process read $x_k = v'$ and then* *it executes a read operation on x_h, this last returns a value $v*$ such that $w(x_h)v^* \not\mapsto_{co} w(x_h)v$.*

Proof Let us consider a client process c_i that executes a read operation $r_i(x_k)v'$ and then it executes a read operation on variable x_h, namely $r_i(x_h)v^*$. Since we assume that each written value is univocally associated to a write operation, we have to prove that $w(x_h)v^*$ does not causally precede $w(x_h)v$. When process c_i invokes a read operation on x_h its matrix clock MC_i is $\geq MC.w(x_h)v$. This may be proved by induction, considering that a read from relation is created if a client process reads the value written by another process, by observation 5 and by line 15 of read procedure in Figure 5.7, lines 2,3 of read thread in Figure 5.8 and lines 7, 9 of write thread in Figure 5.9.

So, when c_i issues is read request, the read message contains $MC_i \geq MC.w(x_h)v$, i.e. $MC_i[r][s] \geq MC[r][s].w(x_h)v$ for all r, s. When an object entity o_j receives such a request, according to line 2 of read thread in Figure 5.8, we have two possible cases: i) o_j sends back its local value of x_h and its MCo_j or ii) o_j sends back to c_i the content of c_i request message. In detail,

i) $MC_{o_i}[h][last] > MC_i[h][last]$ and so $MC_{o_i}[h][last] > MC[h][last].w(x_h)v$. If v^* is the value returned, $MC_{o_i}[h] = MC[h].w(x_h)v^*$. This means that $MC[h][last].w(x_h)v^* > MC[h][last].w(x_h)v$ and so for lemma 5.5.1, we have that $w(x_h)v^*||_{co}w(x_h)v$ or $w(x_h)v \mapsto_{co} w(x_h)v^*$.

ii) c_i reads the value of x_h previously cached. This value could be cached because of a precedent read operation on x_h (again case i) or to maintain mutual consistency when reading another variable. Because of lines 13,14 of read procedure in Figure 5.7 and lines 6,7 of write thread in Figure 5.9 this value is v, a concurrent value or a more recent one w.r.t. $\mapsto_{co}$ because of lemma 5.5.1.

Chapter 6

Conclusions

This thesis has presented a deep study of shared memory, i.e. one of the most interesting interprocess communication models among a set of application processes which are decoupled in time, space and flow. In particular, we focused on causal memories since they offer a good tradeoff between memory access order constraints and the complexity of the programming model as well as of the complexity of the memory model itself.

Differently from strict criteria, causal consistency allows non-blocking operations, i.e. processes may complete read or write operations without waiting for global computation. Moreover, several application semantics are precisely captured by causal consistency, e.g. collaborative tools. So, implementing a stricter consistency criterion not only induces unnecessary complexity to maintain consistency but also reduces the level of possible concurrency.

Traditional distributed systems In the context of traditional distributed shared memory implementations, MCS enforcing causal consistency has been usually implemented by protocols based on (complete/partial)replication of variables at each MCS process. This aimed at exploiting all the concurrency allowed by causal consistency, i.e. application processes may simultaneously access (by reading or writing) a same shared variable. But replication requires consistency maintenance. In this sense, we have proposed an optimality criterion to avoid the introduction of unnecessary constraints when maintaining causal consistency. Moreover, we have pointed out the limits of partial replication implementations in terms of processes intended to manage information to maintain causal consistency.

In detail, we have defined an optimality criterion for *Complete Replication and Propagation* (CRP) based protocols implementing the Memory Consistency System.

Based on complete replication of variables at each MCS process and propagation of the variable updates, these protocols maintain causal consistency through the suspension/reactivation of process threads which are in charge of executing the local update. If an update message arrives at a process and its immediate application violates causal consistency, the update thread is suspended by the CRP protocol, i.e. its application is delayed. Informally, an optimal CRP protocol allows each update to be applied at a MCS process *as soon as* the causal consistency criterion is not violated. In other words, no update thread is kept suspended for a period of time more than strictly necessary. This aims at completely exploiting the concurrency allowed by causal consistency. We have also presented an optimal CRP protocol.

In addition, we have proved that while be promising with respect to system scalability, distributed causal memories implementations supporting partial replication suffer the following drawback: in absence of an a priori knowledge of variable distribution, partial replication implementations require each MCS process to manage information about all shared variables in order to keep the memory causal consistent.

Dynamic distributed systems Finally, we have studied causal memories in the context of emerging dynamic systems (e.g. peer-to-peer). From the dynamics and intrinsic unreliability characterizing such systems the necessity to formalize a persistency property arises. Moreover, because persistency may hard or even not possible to be implemented when considering high dynamic systems, we have introduced a weak form of persistency. This last is strong enough to ensure some computational progress and weak enough to be implemented in high dynamic distributed systems, such as peer-to-peer systems. We have finally provided a weakly persistent causal consistent distributed shared memory implementation along with its correctness proof.

Research directions Applications running on top of dynamic distributed systems may have different persistency requirements. As an example, when considering collaborative applications we need strong persistency to implement a sort of data storage shared by the partners (e.g., [49], [59]). On the contrary, when dealing with mailing services or music file sharing, the lose of some data might be tolerated. This demands for the definition of a hierarchy of persistency properties (best effort towards perfect) for (read/write) shared objects.

Moreover, from emerging system dynamics the necessity for self-maintaining solutions. In other words, while the underling system is not persistent by nature, due to the continuous joins and leaves of nodes, a shared memory may contain a refresh procedure with the aim at maintaining written values persistent despite processes leaves and crashes. Let us notice, that this require a good formalization and understanding of systems dynamics in order to not waste refresh while preserving values. Some important works recently go in such direction, e.g. [38]. Finally, many applications running on the top of a dynamic distributed systems may require stronger consistency criteria than causality. Therefore, new consistency criteria could be investigated especially towards definition and implementation of some form of weak atomicity.

Short Curriculum Vitae I was born in Rome (Italy) in January 1976, and attended in Rome both primary and secondary schools. In 1996 I obtained a Linguistic High School Degree from Liceo Linguistico “Regina Apostolorum”. Then, I started studying at Universitá degli studi di Roma “La Sapienza” - School of Engineering where I obtained a laurea (M.Sc) degree in Computer Engineering. Since 2003, I have been working in the Middleware Laboratory (MidLab) at Dipartimento di Informatica e Sistemistica “A. Ruberti” - Universitá degli studi di Roma “La Sapienza” as a PhD student and teaching assistant under the guidance of Professor Roberto Baldoni (co-advisor). The PhD course has been co-tutored with Université de Rennes 1 (Rennes, France) where I spent 7 months to work with my co-advisor Professor Jean Michel Hélary.

Bibliography

[1] Y. Afek, D.S. Greenberg, M. Merritt, and G. Taubenfeld, *Computing with faulty shared objects*, J. ACM **42** (1995), no. 6, 1231–1274.

[2] A. Agarwal and V. K. Garg, *Efficient dependency tracking for relevant events in shared-memory systems*, PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing (New York, NY, USA), ACM Press, 2005, pp. 19–28.

[3] M. K. Aguilera, *A pleasant stroll through the land of infinitely many creatures*, SIGACT News **35** (2004), no. 2, 36–59.

[4] M. K. Aguilera., W. Chen, and S. Toueg, *On quiescent reliable communication*, SIAM J. Comput. **29** (2000), no. 6, 2040–2073.

[5] M. Ahamad, R.A. Bazzi, R. John, P. Kohli, and G. Neiger, *The power of processor consistency*, SPAA '93: Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures (New York, NY, USA), ACM Press, 1993, pp. 251–260.

[6] M. Ahamad, J. E. Burns, P. W. Hutto, and G. Neiger, *Causal memory*, WDAG '91: Proceedings of the 5th International Workshop on Distributed Algorithms (London, UK), Springer-Verlag, 1992, pp. 9–30.

[7] M. Ahamad, G. Neiger, J.E. Burns, P. Kohli, and P.W. Hutto, *Causal memory: Definitions, implementation and programming*, Distributed Computing **9** (1995), no. 1, 37–49.

[8] M. Ahamad and M. Raynal, *Exploiting write semantics in implementing partially replicated causal objects*, *the Sixth Euromicro Workshop*, 1998, pp. 157–163.

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

[10] M. Ahamad and F.J. Torres-Rojas, *Plausible clocks: costant size logical clocks for distributed systems*, *Distributed Computing* **12** (1999), 179–195.

[11] H. Attiya, A. Bar-Noy, and D. Dolev, *Sharing memory robustly in message-passing systems*, J. ACM **42** (1995), no. 1, 124–142.

[12] H. Attiya and J. Welch, Distributed Computing, second ed., Wiley, 2004.

[13] H. Attiya and J.L. Welch, *Sequential consistency versus linearizability*, ACM Trans. Comput. Syst. **12** (1994), no. 2, 91–122.

[14] O. Babaoglu, A. Bartoli, and G. Dini, *Replicated file management in large-scale distributed systems*, Workshop on Distributed Algorithms, 1994, pp. 1–16.

[15] R. Baldoni, A.Milani, and S. Tucci Piergiovanni, *An optimal protocol for causally consistent distributed shared memory systems*, *International Parallel and Distributed Processing Symposium (IPDPS-04)*, 2004.

[16] R. Baldoni, M. Malek, A. Milani, and S.Tucci Piergiovanni, *Weakly-persistent causal object in dynamic distributed systems*, In proceedings of the 25th IEEE Symposium On Reliable Distributed Systems (SRDS) (Leeds, UK), October 2006.

[17] R. Baldoni and G. Melideo, *On the minimal information to encode timestamps in a distributed computation*, Information Processing Letter **84** (2002), no. 3, 159166.

[18] R. Baldoni, A. Milani, and S. Tucci Piergiovanni, *Optimal propagation-based protocols implementing causal memories*, *Distributed Computing* **18** (2006), no. 6, 461–474.

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

[20] R.E. Bellman, *On a routing problem*, Quart. Appl. Math. **16** (1958), 87–90.

[21] J.K. Bennett, J.B. Carter, and W. Zwaenepoel, *Munin: distributed shared memory based on type-specific memory coherence*, PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming (New York, NY, USA), ACM Press, 1990, pp. 168–176.

[22] D. P. Bertsekas and J.N. Tsitsiklis, Parallel and Distributed Computation: Numerical Methods., Prentice Hall, 1989.

[23] K.P. Birman and T.A. Joseph, *Reliable communication in the presence of failures*, ACM Transactions on Computer Systems **5** (1987), no. 1, 47–76.

[24] K.P. Birman, A. Schiper, and P. Stephenson, *Lightweigt causal and atomic group multicast*, ACM Transactions on Computer Systems **9** (1991), no. 3, 272–314.

[25] B. Charron-Bost, *Concerning the size of logical clocks in distributed systems*, Information Processing Letter **39** (1991), no. 1, 1116.

[26] W. Chen, S. Lin, Q. Lian, and Z. Zhang, *Sigma: A fault-tolerant mutual exclusion algorithm in dynamic distributed systems subject to process crashes and memory losses*, 11th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC), December 2005.

[27] D.R. Cheriton and D. Skeen, *Understanding the limitations of causally and totally ordered communication*, SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles (New York, NY, USA), ACM Press, 1993, pp. 44–57.

[28] M. R. Eskicioglu, *A comprehensive bibliography of distributed shared memory*, SIGOPS Oper. Syst. Rev. **30** (1996), no. 1, 71–96.

[29] A. Fernández, E. Jiménez, and V. Cholvi, *On the interconnection of causal memory systems.*, Journal of Parallel and Distributed Computing **64** (2004), no. 4, 498–506.

[30] C. J. Fidge, *Time stamps in message-passing systems that preserve the partial ordering.*, Australian Computer Science Communications **1** (1988), no. 10, 56–66.

[31] R. Friedman, *Using virtual synchrony to develop efficient fault tolerant distributed shared memories*, Technical Report 95-1506, Department of Computer Science, Cornell University, Ithaca, NY, 1995.

[32] R. Friedman and M. Raynal, *Modularity: A first class concept to address distributed systems*, Tech. Report **PI-1707**, IRISA, Rennes, 2005.

[33] R. Friedman, M. Raynal, and C. Travers, *Two abstractions for implementing atomic objects in dynamic systems*, In proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS) (Pisa, Italy), December 2005.

[34] P. Gambhire and A.D. Kshemkalyani, *Reducing false causality in causal message ordering*, Proceedings of International Conference on High Performance Computing, LNCS 1970, 2000, p. 6172.

[35] V. K. Garg, *Elements of Distributed Computing*, Wiley, 2002.

[36] Vijay K. Garg and Michel Raynal, *Normality: A consistency condition for concurrent objects*, Parallel Processing Letters **9** (1999), no. 1, 123–134.

[37] S. Gilbert, N. Lynch, and A. Shvartsman, *Rambo ii: Rapidly reconfigurable atomic memory for dynamic networks*, In Proc. 17th Intl. Symp. on Distributed Computing (DISC), June 2003, pp. 259–268.

[38] P. B. Goedfrey, S. Shenker, and I. Stoica, *Minimizing churn in distributed systems*, Proc. of ACM SIGCOMM, 2006.

[39] R. Guerraoui and R. Levy, *Robust emulations of shared memory in a crash-recovery model*, Tech. report, EPFL, http://lpdwww.epfl.ch/publications, 2004.

[40] V. Hadzilacos and S. Toueg, *A modular approach to fault-tolerant broadcasts and related problems*, Tech. Report TR 94-1425, Dept. of Computer Science, Cornell University, Ithaca, New York, USA, 1994.

[41] J.M. Hélary and A. Milani., *Efficient partial replication to improve locality in large-scale systems*, International workshop on Locality Preserving Distributed Computing Methods(DISC 2005 Co-located Workshop), 2005.

[42] J.M. Hélary and A. Milani, *About the efficiency of partial replication to implement distributed shared memory*, *International Conference On Parallel Processing(ICPP-06)*, 2006, pp. 263–270.

[43] M. Herlihy and J.M. Wing, *Linearizability: A Correctness Condition for Concurrent Objects*, CACM Transactions on Programming Languages and Systems **12** (1990), no. 3, 463–492.

[44] P. Jayanti, T. D. Chandra, and S. Toueg, *Fault-tolerant wait-free shared objects*, Journal of the Association for Computing Machinery (JACM) **45** (1998), no. 3, 451 – 500.

[45] J.K. Bennett J.B. Carter and W. Zwaenepoel, *Techniques for reducing consistency-related communication in distributed shared-memory systems*, ACM Transactions on Computer Systems (TOCS) **13** (1995), no. 3, 205–243.

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

[47] ———, *On the interconnection of causal memory systems*, Journal of Parallel and Distributed Computing **64** (2004), no. 4, 498–506.

[48] A. Kshemkalyani and M. Singhal, *Necessary and sufficient conditions on information for causal message ordering and their optimal implementation*, Distributed Computing **11** (1998), 91–111.

[49] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, *Oceanstore: An architecture for global-scale persistent storage*, Proceedings of ACM ASPLOS, ACM, November 2000.

[50] L. Lamport, *Time, Clocks and the Ordering of Event in a Distributed System*, Communications of the ACM **21** (1978), no. 7, 558–565.

[51] ———, *How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs*, IEEE Transactions on Computers **28** (1979), no. 9, 690–691.

[52] ———, *On Interprocess Communication. Part I: Basic Formalism*, Distributed Computing **1** (1986), no. 2, 77–85.

[53] R. Lipton and J. Sandberg, *Pram: a scalable shared memory*, Tech. Report **CS-TR-180-88**, Princeton University, 1988.

[54] N. Lynch, Distributed Algorithms., Morgan Kaufmann Publisher, San Mateo,CA, 1996.

[55] N. Lynch, D. Malkhi, and D. Ratajczak, *Atomic data access in content addressable networks*, In proceedings of the First International Workshop on Peer-to-Peer Systems, 2002.

[56] N. Lynch and A. Shvartsman, *Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts*, Symposium on Fault-Tolerant Computing, 1997.

[57] ———, *Rambo: A reconfigurable atomic memory service for dynamic networks*, In Proc. 16th Intl. Symp. on Distributed Computing (DISC), October 2002, pp. 173–190.

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

[59] A. Milani, L. Querzoni, and S. Tucci Piergiovanni, *Global data management*, vol. 8, ch. Data Object Storage in Large Scale Distributed Systems, IOS Press, 2006.

[60] J. Misra, *Axioms for Memory Access in Asynchronous Hardware Systems*, ACM Transactions on Programming Languages and Systems **8** (1986), no. 1, 142–153.

[61] N. Mittal and V. K. Garg, *Consistency conditions for multi-object distributed operations*, International Conference on Distributed Computing Systems, pp. 582–599.

[62] D. Janaki Ram, M. Uma Mahesh, N. S. K. Chandra Sekhar, and C. Babu, *Causal consistency in mobile environment*, Operating Systems Review **35** (2001), no. 1, 34–40.

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

[64] M. Raynal and A. Schiper, *From Causal Consistency to Sequential Consistency in Shared Memory Systems*, Proc. 15th Int. Conf. on Foundations of Software Technology & Theoretical Computer Science, 1995, pp. 180–194.

[65] ——————, *A suite of formal definitions for consistency criteria in distributed shared memories*, Proceedings Int Conf on Parallel and Distributed Computing (PDCS'96) (Dijon, France), 1996, pp. 125–130.

[66] M. Raynal, A. Schiper, and S. Toueg, *The causal ordering abstraction and a simple way to implement it*, Information Processing Letters **39** (1991), no. 6, 343–350.

[67] A. Rowstron and P. Druschel, *Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems*, Proceedings of International Conference on Distributed Systems Platforms (Middleware), 2001.

[68] A. Schiper, J. Eggli, and A. Sandoz, *A new algorithm to implement causal ordering*, Proceedings of the 3rd International Workshop on Distributed Algorithms (London, UK), Springer-Verlag, 1989, pp. 219–232.

[69] M. Singhal and A. Kshemkalyani, *An efficient implementation of vector clocks*, Inf. Process. Lett. **43** (1992), no. 1, 47–52.

[70] H.S. Sinha, *Mermera: Non-coherent distributed shared memory for parallel computing*, Ph.D. thesis, Dept. of Computer Science, Boston University, 1993, TR BU-CS-93-005.

[71] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H.Balakrishnan, *Chord: A scalable peer-to-peer lookup service for internet applications*, In Proceedings of ACM SIGCOMM, 2001.

[72] A. Tarafdar and V.K. Garg, *Addressing False Causality while Detecting Predicates in Distributed Programs*, Proceedings of the 8th International Conference on Distributed Computing Systems, 1998, pp. 94–101.

[73] F. J. Torres-Rojas, M. Ahamad, and M. Raynal, *Lifetime based consistency protocols for distributed objects*, *Proceedings of the 12th International Symposium on Distributed Computing (DISC '98)* (Andros, Greece), September 1998, pp. 378–392.

[74] A. Varga, *Omnet++, http://www.omnetpp.org/.*

[75] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. Katz, and J. Kubiatowicz, *Tapestry: An infrastructure for fault-tolerant wide-area location and routing*, Tech. Report UCB/CSD-01-1141, University of California at Berkeley, Computer Science Division, 2001.

Università *La Sapienza*
Dottorato di Ricerca in Ingegneria Informatica
Collana delle tesi
Collection of Theses

V-93-1 Marco Cadoli. *Two Methods for Tractable Reasoning in Artificial Intelligence: Language Restriction and Theory Approximation.* June 1993.

V-93-2 Fabrizio d'Amore. *Algorithms and Data Structures for Partitioning and Management of Sets of Hyperrectangles.* June 1993.

V-93-3 Miriam Di Ianni. *On the Complexity of Flow Control Problems in Store-and-Forward Networks.* June 1993.

V-93-4 Carla Limongelli. *The Integration of Symbolic and Numeric Computation by p-adic Construction Methods.* June 1993.

V-93-5 Annalisa Massini. *High Efficiency Self-routing Interconnection Networks.* June 1993.

V-93-6 Paola Vocca. *Space-time Trade-offs in Directed Graphs Reachability problem.* June 1993.

VI-94-1 Roberto Baldoni. *Mutual Exclusion in Distributed Systems.* June 1994.

VI-94-2 Andrea Clementi. *On the Complexity of Cellular Automata.* June 1994.

VI-94-3 Paolo Giulio Franciosa. *Adaptive Spatial Data Handling.* June 1994.

VI-94-4 Andrea Schaerf. *Query Answering in Concept-Based Knowledge Representation Systems: Algorithms, Complexity, and Semantic Issues.* June 1994.

VI-94-5 Andrea Sterbini. *2-Thresholdness and its Implications: from the Synchronization with PVchunk to the Ibaraki-Peled Conjecture.* June 1994.

VII-95-1 Piera Barcaccia. *On the Complexity of Some Time Slot Assignment Problems in Switching Systems.* June 1995.

VII-95-2 Michele Boreale. *Process Algebraic Theories for Mobile Systems.* June 1995.