Università degli Studi di Roma "La Sapienza"

Dottorato di Ricerca in Ingegneria Informatica

XIII – 01 – 2

Tracking Causality in Distributed Computations

Giovanna Melideo

Università degli Studi di Roma "La Sapienza"

Dottorato di Ricerca in Ingegneria Informatica

XIII – 01 – 2

Giovanna Melideo

Tracking Causality in Distributed Computations

Thesis Committee

Prof. Giorgio Ausiello and
Alberto Marchetti-Spaccamela (Advisors)
Prof. Roberto Baldoni
Prof. Giuseppe Italiano

Reviewers

Prof. Achour Mostefaoui
Prof. Ravi Prakash

Author's address:
Giovanna Melideo
Dipartimento di Informatica e Sistemistica
Università degli Studi di Roma "La Sapienza"
Via Salaria 113, I-00198 Roma, Italy
e-mail: melideo@dis.uniroma1.it

Contents

Acknowledgements

I would like to thank Giorgio Ausiello and Alberto Marchetti-Spaccamela, my advisors. They are wonderful people and their many suggestions and constant support were very precious for me and made my research possible.

Together with Giorgio and Alberto my gratitude is for Roberto Baldoni who introduced me to the problem of the causality tracking in distributed computations and constantly followed my research. To work with him was a real pleasure.

A special thanks is due to Jean Michel Hélary and Michel Raynal whose hospitality in Rennes was really great. The time I spent there was very precious for me.

My gratitude is for the colleagues I worked with during these years and coauthored my papers: Roberto Baldoni, Jean Michel Hélary, Alberto Marchetti-Spaccamela, Marco Mechelli and Michel Raynal. Among them, I would especially like to thank Roberto, Michel and Jean Michel whose collaboration was of prime importance to accomplish this work.

I wish to thank the Thesis Committee, Giorgio Ausiello, Alberto Marchetti-Spaccamela, Roberto Baldoni and Giuseppe Italiano and for the external reviewers, Achour Mostefaoui and Ravi Prakash. Their advises and suggestions were useful to improve contents and presentation of this thesis.

Many thanks to all the people of the Department of Computer and System Science of the University of Rome "La Sapienza". In particular I would especially like to thank my friends Enver Sangineto, Andrea Vitaletti, Massimo Mecella, Luigi Laura, Camil Demetrescu, Stefano Leonardi and Luca Becchetti, with whom I shared these years in a pleasant atmosphere.

A special thank is due to Stefano Varricchio who introduced me to research for his guidance through the early years of chaos and confusion.

The last words are dedicated to my husband Federico, my parents and to my brother Marco. Without their support, patience and love this work would never have come into existence.

Abstract

An asynchronous distributed computation is usually modeled as a partial order of events produced by n processes. A main characteristic of these computations lies in the fact that processes do not share a common global memory and communicate only by exchanging messages over a communication network.
Causality is a key concept to understand and master the behavior of asynchronous distributed systems. More precisely, given two events of a distributed computation, a crucial problem that has to be solved in a lot of applications is to know whether they are causally related, that is if the occurrence of one of them could be a consequence of the occurrence of the other. Hence, a fundamental distributed computing problem consists in the tracking of causality on events. By associating with each event a vector timestamp whose size n is equal to the number of processes, a timestamping protocol makes possible to safely decide whether two events are causally related or not by only analyzing their timestamps.
The thesis considers the tracking of the causality relation on the events of asynchronous message-passing distributed computations and investigates the problem of reducing the size of the control information piggybacked by application messages (message timestamps).
We propose a formal framework for timestamping protocols which make possible to detect causality relation between events on-the-fly, that is by only comparing their timestamps, and we provide lower bounds on the amount of non-structured information (i.e., number of bits) necessary to encode timestamps and message timestamps when considering such protocols.
In order to face the problem of the size of message timestamps, that could become prohibitive, we present a few protocols which track causality on-the-fly and after a delay on any set of "relevant" events. Namely, timestamping protocols are introduced in a more general context where any subset of events can be considered relevant from the application point of view and communication channels are not required to be FIFO.
We also face the problem of the irremediable loss of causal dependency information, which arises when adopting protocols with a bounded size of message timestamps to track causality between relevant events. With reference to this

we provide a characterization of the minimal size of message timestamps which ensures no loss of dependency information with respect to a given distributed computation and a given set of relevant events.

Chapter 1

Introduction

Distributed and parallel systems are today generally available, and their technology has reached a certain degree of maturity, nevertheless we still lack complete understanding of how to design, realize, and test the software for such systems. Although substantial research effort has been spent on this topic, it seems that understanding the behavior of an asynchronous distributed program remains a challenge. One of the main reasons for this is the nondeterminism that is inherent to such distributed programs; in particular, it is difficult to keep track of the various local activities that happen concurrently and may interact in a way which is difficult to predict.

An asynchronous distributed program is usually modeled as a collection of distinct processes which are spatially separated, and which communicate with another by exchanging messages. The message transmission delay is not negligible compared to the time between events in a single process and execution times and message delays may vary substantially for several repetitions of the same algorithm. Furthermore, a global system clock or perfectly synchronized local clocks are generally not available. A distributed computation describes the execution of a distributed program. Each internal or communication action executed by processes realizing the computation constitutes an event.

For a proper understanding of a distributed program and its execution, it is important to determine the *causality relation* between any pair of events e and e' that occur in its computation, that is to know if the occurrence of event e' could be the consequence of the occurrence of e.

The notion of causality relation between events, which allows us to apprehend the cause/effect relation between events, has been formally defined by Lamport [27], endowing the set of events of a distributed computation with a partially ordered set structure. Since it has been formally defined, this notion has been extensively studied and several protocols to capture it have been proposed [1, 10, 13, 14, 17, 16, 19, 18, 25, 26, 27, 30, 31, 37, 36, 38].

The concept of causality relation is of considerable interest to design of distributed systems and finds applications in several domains such as consistent global state selection [12, 23], detection of obsolete data [28] and replicated data management [32]. Moreover, it is crucial when considering a distributed debugging software tool [18] or a tool for the consistent recovery of a distributed application after a failure of one of its components [9, 12]. These require the capability to detect *causal dependency or concurrency* of events forming a distributed computation "*on-the-fly*", that is concurrently with the execution of the computation, but the characterization and efficient representation of the causality relation is a non-trivial problem.

1.1 Related Work

Since the Lamport's seminal paper, several protocols have been proposed in order to track causality. They require to timestamp events with values in a partially ordered set called timestamps domain and to propagate information (i.e., message timestamps) among processes participating in the computation, by piggybacking additional control data to the computation messages. All these methods have to face the problem of the size of piggybacked information that could become prohibitive.

The *Vector Clocks protocol* has been introduced simultaneously and independently by Fidge [17] and by Mattern [30] to detect causality on-the-fly. This protocol allows an external observer (also called checker) to detect a causality relation between any pair of events by just comparing their timestamps. A major drawback of this method lies in the fact that each message has to piggyback a vector of n integers, where n is the number of processes involved in the computation. This makes it inappropriate to cope with scalability problems. Having a bounded, possibly fixed, control information size, would help protocol designers to develop efficient protocols fro tracking causal dependencies. Nevertheless, Charron-Bost [14] showed that, if timestamps are structured as vectors of integers, the size n is a necessary requirement for detecting causality on-the-fly. Therefore the answer to the scalability issue in a system of vector clocks has to be found by exploiting some tradeoffs.

Up to now, two tradeoffs have been pointed out in the literature: on the one hand, trading message overhead versus local memory overhead, on the other one, trading message overhead and local memory overhead versus missing some concurrencies between events (i.e., concurrent events can be perceived as ordered).

The first class includes the efficient implementation of vector clocks proposed in [37] by Singhal and Kshemkalyani. This implementation tries to move locally as many as possible of the complexity of managing a vector clocks

system while maintaining the on-the-fly detection of causality, in the context of FIFO communication channels. Their efficient technique reduces as possible the size of the message timestamps, however this size could grow up to the number of processes. Moreover, each process maintains two extra vector clocks.

The second tradeoff is exploited by Plausible clocks protocol [38]. This clocks system bounds the size of message overhead and local overhead to some integer k less than n and does not represent causality in an isomorphic way. In fact, the set of timestamps assigned to events is an order extension of the partial order of the computation.

1.2 Contents and Main Results of the Thesis

The thesis considers the tracking of the causality relation on the events of asynchronous message-passing distributed computations by using vector timestamps associated with events by a timestamping protocol[1] and investigates the problem of reducing the size of the control information piggybacked by application messages (message timestamps).

We first discuss an open problem proposed by Schwartz and Mattern [36] about the minimum amount of information managed by protocols which *characterize causality on-the-fly*, i.e., which represent causality in an isomorphic way, by allowing to detect dependencies between two events by only comparing their timestamps. We propose a formal framework for such timestamping protocols and we provide lower bounds on the amount of non-structured information (i.e., number of bits) necessary to encode timestamps and message timestamps. We stress that up to now, the only known result concerning the size of timestamps is the one of Charron-Bost [14] which studies the minimal size of timestamps only in the case where they are structured as vectors of n integers (where n is the number of processes) and states that in such a case the size n is necessary for any timestamping protocol which characterizes causality on-the-fly. Lower bounds prove that any timestamping protocol that characterizes causality on-the-fly has to store locally at each process and to piggyback on each application message an amount of information which is slight smaller than a vector of integers of size n.

In order to answer to the scalability issue we point out a third tradeoff: *trading message overhead versus detection time in causality tracking*. We introduce a property, namely the "strong clock condition with delay", that captures this tradeoff. Operationally, any timestamps system which satisfies this property allows to correctly detect causality relation between any pair of events,

[1] Vector timestamps are not the only way to encode the causality relation. For examining other approaches see for instance [1, 33].

although detection of a set of dependencies might require an extra computation time (detection delay). We derive a scalable causality tracking protocol, namely k-Dependency Vectors, which exploits this new tradeoff by piggybacking on each application message a constant number $k \leq n$ of integers selected from a vector of size n local at each process. It is interesting to remark that if $k = n$, k-dependency vectors become vector clocks. If $k = 1$, k-dependency vectors boil down to a timestamping protocol, namely Direct Dependencies protocol, proposed by Fowler and Zwaenepoel in [18] in the context of distributed debugging.

An important study in this thesis concerns efficient causality tracking timestamping on any subset of events considered relevant from the application point of view. In many applications (distributed debugging and rollback-recovery, just to name a few) detecting causality relations (or concurrencies) on all events is not desirable. More precisely, one is usually interested only in a subset of events called the *relevant* or *observable* events. So, an interesting issue is to consider the causality tracking on relevant events. In this context we propose a suite of simple and efficient implementations of vector clocks that address the reduction of the size of message timestamps at the expense of an extra storage space at each process. From a practical point of view, these protocols form a suite of protocols suited to be embedded in an adaptive timestamping software layer that can be used to reduce as much as possible the control information piggybacked on application messages.

It is important to note that tracking causality on only a subset of events of a computation is a more subtle issue than tracking causality on all the events. This is due to the fact that the non-relevant events can establish causal dependencies on relevant events. This means that a protocol given in a context in which all the events are relevant does not necessarily work when only some events are relevant. This is the case of both the efficient implementation of vector clocks presented by Singhal and Kshemkalyani and the k-dependency vectors protocol introduced first in this thesis, which are reconsidered in the case where only a subset of the events are relevant.

With reference to the protocol presented by Singhal and Kshemkalyani in a context where all the events are relevant and channels are FIFO, we propose an extension to suit the general case where any set of events can be considered as relevant. Then we show that when FIFO channels are available protocols embedded in the software layer are more efficient than the extended version of the protocol of Singhal and Kshemkalyani.

We also face the problem of the irremediable loss of causal dependency information [26], which arises when adopting protocols with a bounded size of message timestamps to track causality between relevant events. With reference to k-dependency vectors protocol we provide a characterization of the

minimal size of message timestamps which ensures no loss of dependency information with respect to a given distributed computation and a given set of relevant events.

Chapter Organization. In the following we overview the structure of the thesis.

In Chapter 2 we describe the formal framework used throughout the thesis and we present a model of an execution of a distributed program as a partially ordered set.

In Chapter 3 we discuss a formal framework for timestamping protocols which characterize causality on-the-fly by storing and piggybacking on messages the bare minimum information. As an example, we analyze the Vector Clocks protocol. Under this framework, we study the size of non-structured information (i.e., the number of bits) necessary to encode timestamps and message timestamps when using any timestamping protocol which characterizes causality on-the fly and we provide lower bounds on the number of bits that any timestamping protocol needs to use in order to encode timestamps and message timestamps in order that the protocol may capture causality.

Chapter 4 provides a comprehensive list of the tradeoffs in a vector clocks system answering to the scalability issue. Mainly, it presents a general scheme for causality tracking, called k-Dependency Vectors, which exploits a tradeoff between the size of the control information piggybacked on application messages and the delay at the checker in detecting causality between events non-on-the-fly detectable. This scheme has direct dependencies [18] and vector clocks [17, 30] as extreme cases when considering k equal to one and to n, respectively. We first propose a formal characterization of this suite of protocols and then we present implementation rules. A discussion of experimental results of a simulation study which carries out a performance comparison between k-dependency vectors protocol using specific strategies (namely, MRR and random ones) and direct dependencies protocol (i.e. 1-Dependency Vectors protocol) concludes the chapter.

In the rest of the thesis we investigate the causality tracking on a subset of events which are considered relevant from the application point of view.

In Chapter 5 we investigate the problem of reducing the size of the control information piggybacked by application messages. This investigation is done in a context where communication channels are not required to be FIFO and where there is no a priori information on the communication graph connectivity or the communication pattern. We first introduce an abstract condition which expresses which part of control information can be omitted from a message timestamp without preventing a correct causality tracking event

timestamping. Then, several protocols are proposed which are based on a concrete approximation of the abstract condition. These protocols provide efficient implementations of vector clocks, in the sense that the size of message timestamps can be less than the number of processes. We show that when FIFO channels are available, these protocols are strictly more efficient than the extended version (also proposed in this chapter) of the Singhal and Kshemkalyani's protocol.

In Chapter 6 we provide a characterization of the sets of relevant events ensuring no loss of dependency information when adopting protocols with a bounded size of message timestamps. Equivalently, the main result is a characterization of the minimal size to be used in order to ensure completeness with respect to a given distributed computation and a given set of relevant events. We point out a tradeoff between the amount of information that must be propagated within computation and the delay in detecting causality relations between events. Although the minimal size can be known only when all the computation is known, it can be used off line to perform a posteriori analysis of a computation.

Most of the results of this thesis has been published in or submitted to International Journals and Conferences [5, 6, 8, 21, 22, 24]. In particular, a part of the results in Chapters 3, 5 and 6 appears in [5, 21, 22, 24].

Chapter 2

Asynchronous Distributed Computations

2.1 Introduction

A *distributed program* consists of a collection of several sequential processes. Processes do not share any memory and communicate and synchronize solely by *asynchronous message passing*, i.e., the message propagation delay is finite but unpredictable, and the computation at a process that sends a message does not wait for an acknowledgement that the message is delivered. More important, there is no common physical clock. A distributed program can be directly written by a programmer or can be the result of the compilation of a parallel or a sequential program for a distributed memory parallel machine.

A *distributed computation* describes the execution of a distributed program. Processes realizing the distributed computation execute actions which are either communication actions, as sending and reception of a message, or internal actions (all the other actions). Each execution of such an action constitutes a *primitive event*.

We point out that in this thesis we only model *complete* computations, that is distributed computation with *point-to-point reliable communications*. A priori, communication in distributed systems is not reliable, that is messages can be lost because of communication failures, duplicated because of retransmissions, or their contents can be garbled or destroyed, nevertheless sophisticated techniques and protocols have been devised which cope with these problems and guarantee reliable message delivery to the application. For this reason, in this thesis will be supposed that *messages communication is reliable*. With point-to-point reliable communications, we can suppose that send and receive events are in one-to-one correspondence, and there is no message in transit at the end of the computation.

2.2 Terminology

Before introducing the distributed computation model in Section 2.3, it is useful to recall some basic notations related to the partial order set theory[1].

A *partial order* $P = (X, \leq_P)$ consists of a finite set X of elements of P and a set $\leq_P \subseteq X \times X$ of relations of P which are pairs (x, y) of elements, denoted $x \leq_P y$. The relation $\leq_P$ is transitive and antisymmetric. When $\leq_P$ is also irreflexive, it will be denoted $<_P$.

Let x and y be two different events in E. We say that x and y are *comparable* in P, denoted $x \sim_P y$, when either $x \leq_P y$ or $y \leq_P x$ holds. On the other hand, x and y are said to be *incomparable* in P, denoted $x \|_P y$, if neither $x \leq_P y$ nor $y \leq_P x$ holds.

A *total order* $P = (X, \leq_P)$ is a partial order in which every pair of distinct elements are comparable.

Definition 2.2.1 (Covering Relation). We say that x is *covered* by y (or y covers x), denoted $x \leq_P^- y$, if

$$(x \leq_P y) \ \wedge \ (\nexists z \in X \ (x \neq y \neq z), \quad x \leq_P z \leq_P y). \tag{2.2.1}$$

Hasse Diagram and Covering Graph. A partial order $P = (X, \leq_P)$ defines the *Hasse diagram* of P, denoted $P^- = (X, \leq_P^-)$, which really represents the transitive reduction of P. The directed graph associated with P^- is called the *covering graph* and is defined as a directed graph whose vertices are elements of X and two vertices x and y are joined by a directed edge if x is covered by y. Usually, the direction of edges is not represented by arrows but must be read bottom-up.

Definition 2.2.2 (Chain/Antichain). A *chain* (resp. *antichain*) in a partial order $P = (X, \leq_P)$ is a subset of X such that every pair of distinct elements are comparable (resp. incomparable).

Definition 2.2.3 (Decomposed Partial Order). A partial order $P = (X, \leq_P)$ is called a *n-decomposed partial order* if X can be decomposed into n sets $X_1, \ldots, X_n$ such that X_i is a chain, for every $i = 1, \ldots, n$.

Definition 2.2.4 (Order Restriction). A partial order $Q = (A, \leq_Q)$ is an *order restriction* of $P = (X, \leq_P)$ on A if $A \subseteq X$, and

$$\forall x, y \in A, \ x \leq_Q y \ \Leftrightarrow \ x \leq_P y. \tag{2.2.2}$$

[1] For an introduction to partial order theory, see for instance [15].

Definition 2.2.5 (Order Extension). An order $Q = (X, \leq_Q)$ is an *order extension* of $P = (X, \leq_P)$ if

$$\forall x, y \in X,\ x \leq_P y \ \Rightarrow\ x \leq_Q y. \tag{2.2.3}$$

Definition 2.2.6 (Linear Extension). A *linear extension* $Q = (X, \leq_Q)$ of $P = (X, \leq_P)$ is an order extension of P which is a total order.

Definition 2.2.7 (Prefix-Closed Subset). Let $P = (X, \leq_P)$ be a partially ordered set. $A \subseteq X$ is a prefix-closed subset of X (under $\leq_P$) if, when $x \in A$, one has: $y \leq_P x \ \Rightarrow y \in A$.

We will denote as $2^X_{\leq_P} \subseteq 2^X$ the set of all the prefix-closed subsets of $(X, \leq_P)$.

Further Notations. Let $f : A \times B \to C$ denote any function. In this thesis we will adopt the following notation:

- $\forall x \in A, f_{A=x} : B \to C$ is the projection of f on x, i.e., $\forall y \in B,\ f_{A=x}(y) = f(x, y)$. When the context will make evident that we are considering the projection on A, we will write f_x instead of $f_{A=x}$;
- $\forall y \in B, f_{B=y} : A \to C$ is the projection of f on y, i.e., $\forall x \in A,\ f_{B=y}(x) = f(x, y)$. As before, sometimes we will write f_y instead of $f_{B=y}$;
- the image of any function f is denoted $Im(f)$.

2.3 The Computation Model

Let $P_1, P_2, \ldots, P_n$ be n sequential processes in a distributed program $\mathcal{P}$. Each process P_i runs a sequential program, whose execution is modeled by a finite set E_i of primitive events, which can be classified in three types of events, namely, *send*, *receive*, and *internal* events.

For every event e, let $P_{e.producer}$ (or, wlog, $e.producer$) be the process producing event e. In particular, if e is a send or a receive event of a message m, we will denote as $P_{e.sender}$ and $P_{e.receiver}$, respectively, processes which produce event e. Moreover, when referring to a message m, processes which produce corresponding send and receive events will be represented by $P_{sender(m)}$ and $P_{receiver(m)}$.

Let E denote the set of events produced by any execution of all the n local programs, that is E is the *disjoint union* of sets E_i: $E = \biguplus_{i=1}^{n} E_i$ ($\uplus$ denotes the disjoint union). Events evolve dynamically during the computation and can be obtained by collecting traces issued by the running processes.

2.3.1 Causality Relation

In an abstract way, events produced by any execution of a distributed program of n processes are related, as Lamport remarked in 1978 [27].

Local Precedence Relation $\prec_l$. As we assume that each P_i is strictly sequential, events occurring at process P_i are *totally ordered* by their local sequence of occurrence. Therefore, if we denote as $e_{i,j}$ the j-th event produced by process P_i, it can be defined on E a relation of *local precedence* $\prec_l$ which asserts that for every i,

$$e_{i,j} \prec_l e_{i,j'} \;\Leftrightarrow\; j < j'.$$

Message Precedence Relation $\prec_m$. Communication actions executed by processes establish relations of potential causality among events in different processes. Let $send(m)$ denote a send event of a message m and $receive(m)$ the corresponding receive event. We can define on E a relation of *message precedence* $\prec_m$ which states that, for every pair of events $e, e' \in E$, $e \prec_m e'$ if e is the sending action of a message m and e' the corresponding reception action, i.e.,

$$e \prec_m e' \;\Leftrightarrow \exists m,\; (e = send(m)) \wedge\; (e' = receive(m)).$$

The cause and effect relation between events of any distributed computation is captured by Lamport's "happened before" or *causality relation*, which defines a partial order $\prec$ on the set of events E describing the computation [27].

Definition 2.3.1. For a given distributed computation, $e \prec e'$ holds, for every pair of events $e, e' \in E$, if one of the following conditions holds:

1. $e \prec_l e'$;
2. $e \prec_m e'$;
3. $\exists e'' \in E$ such that $e \prec e''$ and $e'' \prec e'$.

The causality relation is the smallest relation satisfying these conditions.

We note that we can equivalently define the *causality relation* as the transitive closure of the partial relation $\prec_l \cup \prec_m$, denoted as

$$\prec = (\prec_l \cup \prec_m)^+. \tag{2.3.1}$$

Definition 2.3.2 (Concurrency). Two distinct events e and e' in E are *concurrent*, denoted $e \| e'$, if they are incomparable, i.e., $\neg(e \prec e') \wedge \neg(e' \prec e)$.

We can safely assume that a process does not communicate with itself, i.e., corresponding send and receive events are located on different processes. More generally, as distributed computations in which an event can causally precede itself do not seem to be physically meaningful, we assume $\neg(e \prec e)$ for every event e, i.e., $\prec$ is an *irreflexive* partial order on E.

The causality relation determines the primary characteristic of time, namely that the future cannot influence the past. A way of viewing previous definitions is to say that $e \prec e'$ means that it is possible for event e to causally affect event e'. Two events are concurrent if neither can causally affect the other. Appropriately, $e \prec e'$ can be also read "e can causally affect or precedes e'", "e' potentially depends on e", "e happens before e'", or "e' knows about e".

Immediate Predecessors. Let $\prec^-$ denote the covering relation associated with E. In our context, when an event e' is covered by another event e, we will say that e' is an *immediate predecessor* of e. In complete computations, where send and receive events are in one-to-one correspondence, there are at most two immediate predecessors per event e, that is the *local immediate predecessor* and the *communication immediate predecessor*, when e is a receive event. Namely:

- e' is the local immediate predecessor of e, denoted $e' = pred_l(e)$, if $e' \prec_l^- e$. We recall that the local precedence relation $\prec_l$ is transitively closed and $\prec_l^-$ denotes its transitive reduction, i.e., $e_{i,j} \prec_l^- e_{i,j'}$ only if $j = j' - 1$.
- e' is the communication immediate predecessor of e, denoted $e' = pred_m(e)$, if $e' \prec_m e$. In such a case we note that by definition $\prec_m = \prec_m^-$.

2.3.2 Computations as a Partial Ordered Sets of Events

In an abstract way, each execution of a distributed program $\mathcal{P}$ can be viewed as a n-decomposed partial ordered set $\widehat{E} = (E, \prec)$ where all the events occurring during the computation are the elements, and the ordering between these events is that imposed by the causality relation. We can equivalently consider a distributed computation as a directed graph $(E, \prec)$ (also called the *computation graph*) where events represent the vertices and $\prec$ the set of edges.

Let $e_{i,j}$ denote the *generic* j-th event produced by *any* execution of process P_i. The partial order $\prec$ (i.e., the set of edges in the computation graph) indicates how these events are causally related: an event $e \in E$ is a send event and $e' \in E$ its corresponding receive events if $(e, e') \in \prec$. On the contrary, e is an internal event if it does not appear in any pair of $\prec$.

Assumption: the parameter m. Without loss of generality, for simplicity's sake, we suppose that E_i is a chain of size m for each $i \in \{1, \ldots, n\}$, that is each process realizing the distributed computation executes m actions. When an execution of process P_i produces less than m events, we can safely assume that the chain is completed with "dummy" internal events. Under this assumption, different executions of the same distributed program $\mathcal{P}$ of n processes can be modeled as the set of n-decomposed partially ordered sets $(E, \prec)$ (E_i being a chain of size m) which are able to describe distributed computations.

Remark 2.3.1. By definition of causality relation (see Definition 2.3.1), a relation $\prec \in E \times E$ can be a causality relation on E only if $\prec_l \subseteq \prec$, where

$$\prec_l = \{(e_{i,j}, e_{i,j'}) \mid \ i = 1, \ldots, n, \ j = 1, \ldots, m-1, \ j' > j\}.$$

Therefore, a partial order $\widehat{E} = (E, \prec)$ which models a computation is necessarily an opportune order extension of the partial order $(E, \prec_l)$. In particular, the partial order $(E, \prec_l)$ models a computation in which all the sequential processes execute internal actions, i.e., $e_{i,j}$ is an internal event, for every $e_{i,j} \in E$. Then, distributed computations can be obtained by defining on $(E, \prec_l)$ appropriate relations $\prec_m$. That is, every computation graph is based on a "basic" n-decomposed computation graph $(E, \prec_l)$ and it can be obtained by adding edges between vertices in different processes, without forming cycles and in such a way that pairs of connected vertices are in a one-to-one correspondence.

By following previous considerations and by considering the Charron-Bost's definition of causality relation [14], we can state that:

Definition 2.3.3. A binary relation $\prec \subseteq E \times E$ is a causality relation if:

- $\prec_l \subseteq \prec$, denoting that $(E, \prec)$ is a n-decomposed partial order;
- $\prec$ is *irreflexive*, denoting that the directed computation graph is acyclic;
- $\forall e \in E, \quad |\{(e, e') \in \prec_m | \ e' \in E\} \cup \{(e', e) \in \prec_m | \ e' \in E\}| \leq 1$, this restriction imposing that for every receipt of a message m, there exists a single sending of m.

Definition 2.3.4 (Causality Relations Set). The set of relations $\prec \subseteq E \times E$ which are causality relations of E is denoted as $\mathcal{C}[E] \subseteq 2^{(E \times E)}$.

Consequently, we can give a formal model for the *computations set* $\widehat{\mathcal{E}}$.

Definition 2.3.5 (Computations Set). The set $\widehat{\mathcal{E}}$ modeling all the executions of a distributed program $\mathcal{P}$ is defined as the set

$$\widehat{\mathcal{E}} = \{\widehat{E} = (E, \prec) \mid \prec \in \mathcal{C}[E]\}. \tag{2.3.2}$$

Definition 2.3.6 (Causal Past). Let $\widehat{E} = (E, \prec)$ be a distributed computation in $\widehat{\mathcal{E}}$. The *causal past* of an event $e \in E$ produced during the execution of $\widehat{E}$ is the *prefix-closed set* of events which might have affect e. Namely:

$$\begin{gathered} \downarrow : E \times \mathcal{C}[E] \rightarrow 2^E \\ \downarrow (e, \prec) = \{e' \in E \mid \ e' \prec e\} \cup \{e\}. \end{gathered} \tag{2.3.3}$$

Causal Pasts Properties

Each causal past $\downarrow (e, \prec)$ can be decomposed in n disjoint subsets $\downarrow_i (e, \prec)$ which represent each one the projection of $\downarrow (e, \prec)$ on E_i, i.e., $\downarrow_i (e, \prec) = \downarrow (e, \prec) \cap E_i$.

Remark 2.3.2. Function $\downarrow$ is not injective, that is there exist events in different computations with the same causal past, i.e.,

$$\exists \ \prec, \prec' \in \mathcal{C}[E], \ \exists \ e, e' \in E, \ \ \downarrow (e, \prec) = \downarrow (e', \prec'). \tag{2.3.4}$$

So, when $\downarrow (e, \prec) = \downarrow (e', \prec')$, either $(\prec \neq \prec')$ or $(\prec = \prec' \wedge e = e')$ must hold. On the contrary, for every causality relation $\prec \in \mathcal{C}[E]$, the function $\downarrow_\prec$ (i.e., the restriction of $\downarrow$ to the causality relation $\prec$) is *injective*, as, by definition of causal past, $\forall e, e' \in E$, $e \neq e' \ \Rightarrow \downarrow_\prec (e) \neq \downarrow_\prec (e')$.

As stated by Schwartz and Mattern in [36], by Definition 2.3.6 of causal past directly follows that causality and causal past notions are related.

Lemma 2.3.1. *Causality and causal past are related as follows:*
$\forall \prec \in \mathcal{C}[E]$ *and* $\forall e, e' \in E, \ (e \neq e')$ *one has:*

1. $e \prec e' \Leftrightarrow e \in \downarrow (e', \prec)$;
2. $e \| e' \Leftrightarrow e \notin \downarrow (e', \prec) \ \wedge \ e' \notin \downarrow (e, \prec)$.

That is, for each given computation $\widehat{E} = (E, \prec) \in \widehat{\mathcal{E}}$, the partial order $(Im(\downarrow_\prec), \subset)$ of all the causal pasts which can be observed during the execution of $\widehat{E}$ is an isomorphic embedding of this computation. In fact, $\downarrow_\prec$ is injective. Moreover, for every pair of events e, e' and for every causality relation $\prec \in \mathcal{C}[E]$, one has the following property:

$$e \prec e' \Leftrightarrow \downarrow_\prec (e) \subset \downarrow_\prec (e'). \tag{2.3.5}$$

Notations. Let $\widehat{E} = (E, \prec)$ be a computation in $\widehat{\mathcal{E}}$. For simplicity's sake, we will denote as:

- $\downarrow[\widehat{E}]$, instead of $Im(\downarrow_{\prec})$, the set of causal pasts of events in the computation $\widehat{E}$, i.e., $\downarrow[\widehat{E}] = \bigcup_{e \in E}\{\downarrow(e, \prec)\}$;
- $\downarrow[\widehat{\mathcal{E}}]$, instead of $Im(\downarrow)$, the set of causal pasts which could be observed during different executions of a distributed program $\mathcal{P}$, i.e., $\downarrow[\widehat{\mathcal{E}}] = \bigcup_{\widehat{E} \in \widehat{\mathcal{E}}} \downarrow[\widehat{E}]$.

2.3.3 Graphical Representations

Space-Time Diagram

It is helpful to view a distributed computation $\widehat{E}$ in terms of a "space-time diagram", of which Figure 2.1(a) shows an example. The vertical direction represents space, and the horizontal direction represents time, which is assumed to move from left to right. The horizontal lines denote processes, each dot on process lines denotes an event $e \in E$. All the pairs in the causality relation $\prec$ (or, equivalently, messages) are depicted as arrows connecting send events with their corresponding receive events.

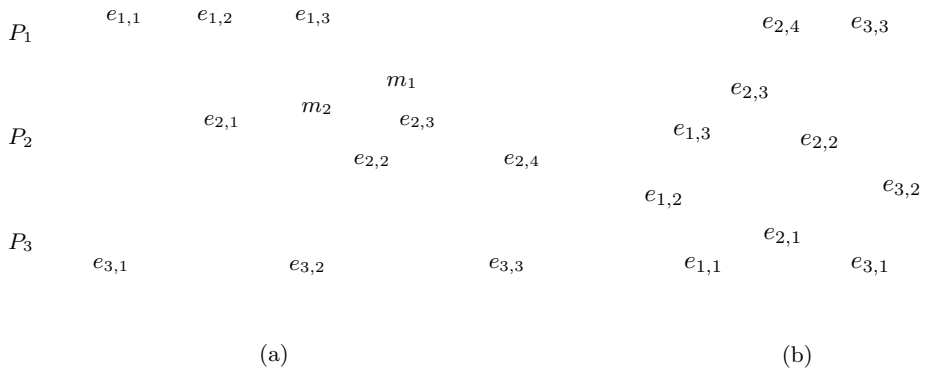


Figure 2.1: A Space-Time diagram of a distributed computation (a) and its corresponding Covering Graph (b)

The crucial point in the definition of asynchronous computations is that the transitive relation $\prec$ is required to be a partial order. This guarantees that it is cycle-free and that space-time diagrams can be drawn in such a way that the process lines and all messages arrows go from left to right.

It is easy to see that in asynchronous computations an event e causally precedes another event e' if and only if there is a directed left-to-right path in the diagram starting at e and ending at e'.

Covering Graph

The poset $\widehat{E}^- = (E, \prec^-)$ defines the Hasse diagram associated with the distributed computation $\widehat{E} = (E, \prec)$, which considers only non-transitive causal dependencies between events. Figure 2.1(b) shows the Covering Graph associated with the Hasse diagram $\widehat{E}^-$ of the computation of Figure 2.1(a).

Note that Figure 2.1(a) seems to make implicit allusions to global time depicting a specific computation, whereas Figure 2.1(b) only shows the logical relation of events, i.e., the causal structure of the computation.

2.3.4 FIFO-Computations

Messages and their transmission modes play an important role in distributed systems since messages are the only means by which processes can exchange data and can synchronize their actions. Even if one supposes that messages communication is reliable, as it is done in this thesis, there exist several paradigms for ordered delivery of messages in a distributed system. For example, messages may be transmitted in a "blocking" way, that is the sender is forced to wait until the message is delivered at its destination, and messages might be received in the order in which they were sent or received out of sequence.

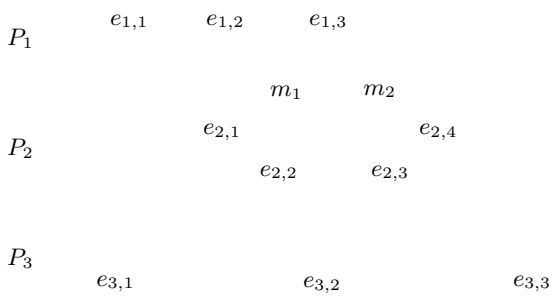


Figure 2.2: A FIFO-computation

In the design of distributed algorithms it is often assumed that any two messages exchanged between the same two processes are received in the order in which they have been sent (see Figure 2.2), while the order in which the messages are received may be arbitrary in case the send events of the messages are

concurrent. This could arise at different levels: it could result from a particular implementation of a communication protocol, or they could be imposed by the semantics of a distributed programing language in order to simplify correctness proofs of distributed programs.

This constraint, referred to as the FIFO (i.e., "First-In First-Out"), can be modeled by the following condition:

Definition 2.3.7 (FIFO). An asynchronous distributed computation $\widehat{E}$ is called a FIFO-computation if for all pairs of events (e, e') and (f, f') such that $e \prec_m e'$ and $f \prec_m f'$

$$e \prec_l f \ \wedge \ (\exists i, \ e', f' \in E_i) \ \Rightarrow \ e' \prec_l f'. \tag{2.3.6}$$

For instance, the distributed computation depicted in Figure 2.1(a) is not a FIFO-computation. In fact, $send(m_1) \prec_l send(m_2)$, $receive(m_1)$ and $receive(m_2)$ are events on the same process but $receive(m_2) \prec_l receive(m_1)$. On the contrary, in Figure 2.2 the same computation is forced to be a FIFO-computation.

Chapter 3

Causality Tracking between Events

Timestamping protocols are used to capture the causality relation or the concurrency between events in distributed computations.

In this chapter we are interested in analyzing the size of non-structured information (i.e., the number of bits) necessary to encode timestamps and message timestamps when timestamping protocol characterizes causality on-the fly, i.e., represents causality in an isomorphic way. To reach this purpose we introduce a formal framework for such timestamping protocols and by taking into account some operational aspects, we exhibit a few properties which they have to satisfy in order to characterize causality on-the-fly. Under the proposed formal framework we derive important lower bounds on the amount of information managed by protocols which represent causality in an isomorphic way.

Results presented in this chapter affirm the necessity of using at least:

- $\lceil \log_2 (m+1)^n - \sum_{k=3}^{n} \binom{n}{k} \binom{2k-3}{k} \rceil$ bits to encode timestamps;
- $\lceil \log_2 (m+1)^{n-1} - 2^{n-1} + n - 1 \rceil$ bits to encode message timestamps.

These statements answer the open problem of Schwartz and Mattern [36] about the minimum amount of information managed by protocols which represent causality in an isomorphic way. We remark that they are more general than well-known Charron-Bost's result [13] who only analyzed the minimal size of "structured" information, by affirming the necessity of using at least n integers to characterize causality on-the-fly, when the protocol structures timestamps as vectors of integers.

The chapter is structured in 4 sections. Section 3.1 presents a formal framework for timestamping protocols. Vector Clocks protocol is introduced in Section 3.2. Main results are contained in Section 3.3, where we first prove a

bijection between causal pasts and timestamps which could be observed when considering several executions of a distributed program $\mathcal{P}$ (Subsection 3.3.1). Next, in Section 3.3.2 a characterization of prefix-closed subsets of events which are causal pasts is provided, and the cardinality of this set is computed. In Section 3.3.3 we point out that the timestamps domain must be a partially ordered set equipped with an operator which has to satisfy suitable properties. This is the baseline for a formal framework for "efficient" timestamping protocol (Subsection 3.3.4). In Section 3.3.5 we propose a formal characterization of prefix-closed subsets of events of $(E, \prec_l)$ which are message timestamps, and the cardinality of this set is computed. Lower bounds on the amount of information managed by protocols which characterize causality on-the-fly are provided in the last Section 3.3.6. Section 3.4 concludes the chapter.

3.1 Timestamping Protocols

Methods used to track causality between events are based on timestamps associated with events and on the piggybacking of information on messages (also called *message timestamps*) used to update timestamps. A timestamping protocol assigns on-the-fly, that is during the evolution of a computation $\widehat{E} = (E, \prec)$, to each event $e \in E$, a value of a suitable partially ordered set $(T, <)$.

3.1.1 A Formal Approach

A *timestamping protocol* $\mathcal{A}$ is usually characterized by:

- a partial ordered set $\widehat{T} = (T, <)$ called *timestamps domain*;
- a *timestamping function* $\phi : E \times \mathcal{C}[E] \rightarrow T$ which establishes a correspondence between events in a computation and values in T, called timestamps;
- a set of rules implementing the algorithm which decide control information to piggyback on outgoing messages, called *message timestamps*, in order to update timestamps associated with events.

The Checker Process

We consider an additional *process observer* (i.e. *the checker*), whose role is to detect if a pair of events is causally related or concurrent by only comparing timestamps associated with events. Namely, each time an event e is generated by a process P_i during an execution of a computation $\widehat{E} = (E, \prec)$, the checker receives a pair $(e, \phi(e, \prec))$, $\phi(e, \prec)$ being the current timestamp associated

with e. By using these information, it has to detect causal dependencies and concurrencies between events.

Notations. Let $\widehat{E} = (E, \prec)$ be a computation in $\widehat{\mathcal{E}}$. For simplicity's sake, we will denote as:

- $\phi[\widehat{E}]$, instead of $Im(\phi_{\prec})$, the set of timestamps associated with events during the evolution of the computation $\widehat{E}$, i.e., $\phi[\widehat{E}] = \bigcup_{e \in E}\{\phi(e, \prec)\}$.

  The pair $(\phi[\widehat{E}], <)$ denotes the order restriction of $(T, <)$ on $\phi[\widehat{E}]$, called *timestamps system.*

- $\phi[\widehat{\mathcal{E}}]$, instead of $Im(\phi)$, the set of all the timestamps which could be associated with events during several executions of the distributed program $\mathcal{P}$, i.e., $\phi[\widehat{\mathcal{E}}] = \bigcup_{\widehat{E} \in \widehat{\mathcal{E}}} \phi[\widehat{E}]$.

3.1.2 Strong Completeness Property

The aim of a timestamping protocol is to associate on-the-fly values in T with events, in order that the resultant timestamps system $(\phi[\widehat{E}], <)$ may be an isomorphic embedding of the computation $\widehat{E} = (E, \prec)$, for *any* $\widehat{E} \in \widehat{\mathcal{E}}$, i.e., for any execution of a distributed program $\mathcal{P}$. In this case we say the timestamps system and the protocol *characterize causality (and concurrency) on-the-fly*, or, equivalently, they are *strongly complete.*

This is usually formalized [14, 17, 36] by requiring that the following *Strong Completeness Property* (SC-P) is satisfied:

Property 3.1.1 (SC-P).

$$\forall \prec \in \mathcal{C}[E],\ \forall e, e' \in E,\quad e \prec e' \Leftrightarrow \phi(e, \prec) < \phi(e', \prec). \tag{3.1.1}$$

If the protocol characterizes causality on-the-fly, the checker can instantaneously detect the causal dependency or the concurrency between events by only comparing their timestamps (*on-the-fly detection*), as the structure of causality is represented in an isomorphic way.

3.2 Vector clocks

A Vector Clocks system has been empirically used as an ad hoc device to solve specific problems [28, 32, 34] before being defined as a concept, with the associate theory, in 1988, simultaneously and independently by Fidge [17] and by Mattern [30].

Let $(\mathbb{N}^n, <)$ be the partially ordered set of n-dimensional vectors of integers, $<$ being defined as follows:

Definition 3.2.1. For every pair of vectors $U, V \in \mathbb{N}^n$,

$$U < V \;\Leftrightarrow\; (\forall i = 1, \ldots, n,\; U[i] \leq V[i]) \;\wedge\; (\exists j \in \{1, \ldots, n\},\; U[j] \neq V[j]).$$

Moreover, let the function $max : \mathbb{N}^n \times \mathbb{N}^n \to \mathbb{N}^n$ be defined in such a way:

Definition 3.2.2. For every pair of vectors $U, V \in \mathbb{N}^n$:

$$max(U, V)[i] = max(U[i], V[i]),\; \forall i = 1, \ldots, n.$$

3.2.1 The Protocol

Vector clocks protocol associates a n-dimensional vector of integers, denoted $e.VC$, as timestamp with each event e produced by the computation in order that the i-th entry of this vector $e.VC[i]$ may represent the number of events on process P_i in the causal past of e. During the computation, each application message carries a vector clock as control information.

Notation. Later on, when it is clear that we are referring to the current computation, we will drop the causality relation $\prec$ in functions ϕ and $\downarrow$.

From an operational point of view the protocol works as follows: each process P_i endows a vector VC_i of n integers, where $VC_i[j]$ represents the index of the most recent event of process P_j known by P_i.

The value of VC_i actually represents the timestamp $\phi(e)$ of the event e currently produced by P_i. Let $e.VC$ denote such a timestamp associated with the event e.

The vector VC_i is updated according to following rules:

[R0] $VC_i[1..n]$ is initialized to $[0, \ldots, 0]$.

[R1] Each time it produces an event e:

- P_i increments its vector clock entry $VC_i[i]$ ($VC_i[i] := VC_i[i] + 1$) to indicate it has produced one more event;
- P_i associates with e the timestamp $e.VC = VC_i$.

[R2] When a process P_i sends a message m, it attaches to m the current value of VC_i. Let $m.VC$ denote this value.

[R3] When P_i receives a message m, it updates its vector clock in the following way: $VC_i := max(VC_i, m.VC)$.

3.2.2 Vector Clocks Properties

Fidge and Mattern [17, 30] proved that vector clocks system *characterizes causality on-the-fly*. More specifically, it has been proved the following property:

Property 3.2.1. *Given vectors* $e.VC,\ e'.VC \in \mathbb{N}^n$ *associated with events* $e, e' \in E$*:*

$$e \prec e' \Leftrightarrow e.VC < e'.VC. \tag{3.2.1}$$

Moreover, $e||e' \Leftrightarrow\ \neg(e.VC < e'.VC)\ \wedge\ \neg(e'.VC < e.VC)$.

Property 3.2.1 offers a convenient method to determine causal dependencies between events based on their vector clocks. Instead of comparing the whole vectors, the necessary computations can often be reduced even further, according to the following statement:

Proposition 3.2.2. *For two different events* $e \in E_i$ *and* $e' \in E_j$*, we have:*

- $e \prec e' \Leftrightarrow\ e.VC[i] \leq e'.VC[i]$;
- $e \| e' \Leftrightarrow\ (e.VC[i] > e'.VC[i]) \wedge (e'.VC[j] > e.VC[j])$.

Drawback

A major drawback of vector clocks lies in the fact that each message has to carry a vector of n integers, where n is the total number of processes. This makes them inappropriate to cope with scalability problems. As, nowadays, local memory is cheap and available on-board in large quantity, the scalability issue does not concern vectors stored in local memory, at least if n is in a reasonable range. In a distributed setting, scalability is critical with respect to the control information attached to messages.

Nevertheless, Charron-Bost [14] showed that if a timestamping protocol structures timestamps as vectors of integers, then it can characterize causality on-the-fly only if vectors have at least size n, that is, the size n is a necessary requirement.

3.3 Necessary Conditions to Characterize Causality On-the-fly

A question remains about the *minimum amount of information* that has to be contained in timestamps and in message timestamps in order to define a timestamping protocol which truly characterizes causality on-the-fly.

3.3.1 A Depth Analysis of Strong Completeness Property

In this section we point out that if an efficient timestamping protocol which stores and transmits the bare minimum information is strongly complete, then there is a bijection between $\downarrow [\widehat{\mathcal{E}}]$ and $\phi[\widehat{\mathcal{E}}]$. To achieve this aim we analyze a few fundamental operational aspects of timestamping protocols.

We first analyze some properties which are a direct consequence of the SC-P (3.1.1). Next, we prove that it is necessary for a protocol which characterizes causality on-the-fly to also satisfy a few properties which are not necessarily implied by the SC-P one. This means that this property is too weak to characterize any strongly complete protocol.

Property SC-P allows to derive the following relation between causal pasts and timestamps observed during any given execution of a distributed program:

Proposition 3.3.1. *If a timestamping protocol satisfies the strong completeness property 3.1.1 (SC-P), given a computation $\widehat{E}$, the corresponding timestamps system $(\phi[\widehat{E}], <)$ is an isomorphic embedding of $(\downarrow [\widehat{E}], \subset)$, i.e., for every pair of events e, e' and for every causality relation $\prec \in \mathcal{C}[E]$:*

$$\downarrow (e, \prec) \subseteq \downarrow (e', \prec) \Leftrightarrow \phi(e, \prec) < \phi(e', \prec). \tag{3.3.1}$$

Proof. The thesis is directly implied by properties 2.3.5 and SC-P, as for every $\widehat{E} \in \widehat{\mathcal{E}}$ both causal pasts suborder $(\downarrow [\widehat{E}], \subset)$ and timestamps system $(\phi[\widehat{E}], <)$ are isomorphic embeddings of $\widehat{E}$. □

Proposition 3.3.1 states that, when we consider a *single execution of the program* $\mathcal{P}$, i.e., if we fix a computation $\widehat{E} \in \widehat{\mathcal{E}}$, there is a bijection between causal pasts in $\downarrow [\widehat{E}]$ and timestamps in $\phi[\widehat{E}]$. Namely, timestamps assigned to events during any single execution of $\mathcal{P}$ are actually an encoding of their causal pasts.

In other words, Proposition 3.3.1 says that $\forall \widehat{E} = (E, \prec) \in \widehat{\mathcal{E}}$, there exists an *injective* encoding function $\chi_{\prec} : \downarrow [\widehat{E}] \to T$, depending on $\widehat{E}$, such that the timestamping protocol associates a value $\phi(e, \prec)$ with each event e in this way:

$$\forall \widehat{E} = (E, \prec) \in \widehat{\mathcal{E}},\ \forall e \in E,\quad \phi(e, \prec) = \chi_{\prec}(\downarrow (e, \prec)). \tag{3.3.2}$$

An Observation

We are interested in deriving what relation exists between $\downarrow [\widehat{\mathcal{E}}]$ and $\phi[\widehat{\mathcal{E}}]$, that is between the sets of all the causal pasts and timestamps which could be observed during *several executions* of a distributed program (we recall that $\downarrow [\widehat{E}]$ and $\phi[\widehat{E}]$ only refer to a single execution).

With reference to this we note that by Equation (2.3.5) and by SC-P, nothing else, in addition to the Proposition 3.3.1, can be stated on the relation between them. Namely, SC-P ensures that $(\phi[\widehat{E}], <)$ is an isomorphic embedding of $(\downarrow [\widehat{E}], \subset)$ and yet it *does not imply* that if a timestamping protocol characterizes causality, then $(\phi[\widehat{\mathcal{E}}], <)$ must be an isomorphic embedding of $(\downarrow [\widehat{\mathcal{E}}], \subset)$.

In fact, when considering several executions of a distributed program $\mathcal{P}$, property *SC-P is not sufficient* to ensure that different timestamps, even in different computations, correspond to different causal pasts and conversely: for every pair of computations $\widehat{E} = (E, \prec)$, $\widehat{E}' = (E, \prec') \in \widehat{\mathcal{E}}$ and for every pair of causal pasts $S \in \downarrow [\widehat{E}]$ and $S' \in \downarrow [\widehat{E}']$

$$S \neq S' \Leftrightarrow \chi_{\prec}(S) \neq \chi_{\prec'}(S'). \tag{3.3.3}$$

That is, SC-P *cannot guarantee* that for every pair of computations $\widehat{E},\ \widehat{E}' \in \widehat{\mathcal{E}}$, and for every pair of events $e, e' \in E$:

($\Rightarrow$) different timestamps are always assigned to events with different causal pasts, even in different computations:

$$\downarrow (e, \prec) \neq \downarrow (e', \prec') \ \Rightarrow\ \phi(e, \prec) \neq \phi(e', \prec') \tag{3.3.4}$$

($\Leftarrow$) the same timestamp always corresponds to events with the same causal past:

$$\downarrow_{\prec} (e) = \downarrow_{\prec'} (e') \ \Rightarrow\ \phi_{\prec}(e) = \phi_{\prec'}(e'). \tag{3.3.5}$$

Equivalently: $S \in \downarrow [\widehat{E}] \cap \downarrow [\widehat{E}'] \ \Rightarrow\ \chi_{\prec}(S) = \chi_{\prec'}(S)$.

An Operational Analysis

Even though from a theoretical point of view the timestamping protocol can proceed by assigning different timestamps to events with a same causal past or same timestamps to events with different causal pasts, the following proposition shows that this is not acceptable from an operational point of view.

Proposition 3.3.2. *If a timestamping protocol characterizes causality, then* $\forall S, S' \in \downarrow [\widehat{\mathcal{E}}]$ *and* $\forall \widehat{E}, \widehat{E}' \in \widehat{\mathcal{E}}$ *such that* $S \in \downarrow [\widehat{E}]$ *and* $S' \in \downarrow [\widehat{E}']$*:*

$$S \neq S' \Leftrightarrow \chi_{\prec}(S) \neq \chi_{\prec'}(S').$$

Proof. The *necessity* is obvious, since, under the hypothesis that the protocol stores the minimum information to encode causal pasts, Property (3.3.5) must hold.

As regards the *sufficiency*, let us consider two events $e, e' \in E$ in different computations $\widehat{E}, \widehat{E}' \in \widehat{\mathcal{E}}$ whose causal pasts are different and whose timestamps coincide, i.e., $\downarrow (e, \prec) \neq \downarrow (e', \prec')$ and $\phi(e, \prec) = \phi(e', \prec')$.

As stated by property (2.3.5), to correctly deduce all causal dependencies and concurrencies between events only by comparing their timestamps (*on-the-fly detection*), it is necessary to have perfect knowledge of all past events at any time. At an operational level, this means that any external observer whose role is detect such dependencies has to execute an algorithm characterized by a decoding function φ, which correctly deduces causal pasts from timestamps. As the computation cannot be known a-priori, φ cannot depend on any computation, so must be $\varphi : \phi[\widehat{\mathcal{E}}] \rightarrow\downarrow [\widehat{\mathcal{E}}]$. As a consequence, if there was a same timestamp associated with two different causal pasts, then would exist no decoding function φ and the observer would not be able to find the correct causal past of an event without knowing the computation. □

Corollary 3.3.3. *If a timestamping protocol characterizes causality, there exists a bijective encoding function $\chi :\downarrow [\widehat{\mathcal{E}}] \rightarrow \phi[\widehat{\mathcal{E}}]$ which represents an encoding scheme for the set of all the causal pasts in $\downarrow [\widehat{\mathcal{E}}]$.*

Proof. If a protocol characterizes causality, we know that for every computation $\widehat{E} \in \widehat{\mathcal{E}}$, $\exists \chi_{\prec} :\downarrow [\widehat{E}] \rightarrow \phi[\widehat{E}]$. As a consequence of previous proposition, when we consider different executions of a distributed program, different $\chi_{\prec}$ are related as stated by Equation (3.3.3).

So, we can state that it must exist a well-defined bijective function $\chi :\downarrow [\widehat{\mathcal{E}}] \rightarrow \phi[\widehat{\mathcal{E}}]$ which is the extension of all the $\chi_{\prec}$ to $\downarrow [\widehat{\mathcal{E}}]$. Namely, $\chi = \bigcup_{\widehat{E} \in \widehat{\mathcal{E}}} \chi_{\prec}$. □

Theorem 3.3.4. *If a timestamping protocol characterizes causality, $(\phi[\widehat{\mathcal{E}}], <)$ is actually an isomorphic embedding of $(\downarrow [\widehat{\mathcal{E}}], \subset)$.*

Proof. Previous corollary establishes the existence of a bijective encoding function $\chi :\downarrow [\widehat{\mathcal{E}}] \rightarrow \phi[\widehat{\mathcal{E}}]$. The fact that causality has to be detected on-the-fly, i.e. only by comparing timestamps, and that the computation cannot be known a-priori imply that the bijective decoding function $\varphi \equiv \chi^{-1} : \phi[\widehat{\mathcal{E}}] \rightarrow\downarrow [\widehat{\mathcal{E}}]$ executed by any external observer must satisfy the following property: for every pair of timestamps $d, d' \in \phi[\widehat{\mathcal{E}}]$, $d < d' \Leftrightarrow \varphi(d) \subset \varphi(d')$. □

3.3.2 Causal Pasts Characterization

In this section we present an interesting characterization of the set of causal pasts $\downarrow [\widehat{\mathcal{E}}]$. We first provide a property that every prefix-closed subset of $(E, \prec_l)$ must satisfy in order to be a causal past. Then, as an interesting

consequence, we compute the cardinality of the set of all the causal pasts $\downarrow [\widehat{\mathcal{E}}]$, which is fundamental, as it constitutes a lower bound on the timestamps domain T. In fact, as stated by Corollary 3.3.3 and Theorem 3.3.4, every timestamp is an encoding of a causal past, so $|T| \geq |\phi[\widehat{\mathcal{E}}]| = |\downarrow [\widehat{\mathcal{E}}]|$. Moreover, we can state that $\lceil \log_2 |\downarrow [\widehat{\mathcal{E}}]| \rceil$ *bits* constitutes a lower bound on the amount of information which must be contained in timestamps in order to characterize causality on-the-fly.

Let $2^E_{\prec_l} \subseteq 2^E$ denote the set of all the prefix-closed subsets of $(E, \prec_l)$. For simplicity's sake, we will often say that subsets in $2^E_{\prec_l}$ are $\prec_l$-prefix-closed subsets of events.

We recall that each $\prec_l$-prefix-closed set of events $S \in 2^E_{\prec_l}$ can be decomposed in n subsets $S_i = S \cap E_i$ such that $S = \uplus_{i=1}^n S_i$ is the disjoint union of these sets S_i. We say that $S_1, \ldots, S_n$ constitute the *decomposition* of S.

Theorem 3.3.5. *A set $S \in 2^E_{\prec_l}$ is a causal past ($S \in \downarrow [\widehat{\mathcal{E}}]$) if and only if $S \neq \emptyset$ and when the number k of nonempty subsets in its decomposition is at least 3, $|S| \geq 2(k-1)$.*

Proof. The proof is made of two parts: necessity and sufficiency.

Necessity. By the definition of causal past, at least e belongs to $\downarrow (e, \widehat{E})$, so $S \neq \emptyset$. Moreover, it is trivial to find some computation in which one can observe causal pasts involving events on at most two processes. If $k \geq 3$ processes have events in the causal past S, then at least $k-1$ processes have to send messages in order to establish a dependency. Each of $k-1$ messages contributes two events to S. Hence, $2(k-1)$ is the minimum number of events in S when $k \geq 3$.

Sufficiency. Let $S \in 2^E_{\prec_l}$ be a nonempty set and k (with $k \geq 1$) be the number of nonempty subsets of the decomposition of S. We can distinguish the following cases:

- If $k = 1$, then $S = S_i = \{e_{i,1}, e_{i,2}, \ldots, e_{i,h_i}\}$, for some $i \in \{1, \ldots, n\}$. It is easy to see that $S = \downarrow (e_{i,h_i}, \prec)$, where $(E, \prec)$ is a distributed computation in which every event in S_i is an internal or send event.
- If $k = 2$, then $S = S_i \uplus S_j$ for some pair of indices i, j $(i \neq j)$. In this case S can be the causal past of event e_{j,h_j}, in a computation in which e_{i,h_i} and e_{j,h_j} are communication events such that $e_{i,h_i} \prec_m e_{j,h_j}$, and the other events are internal events.
- If $k \geq 3$, let $l \leq k$ be the number of subsets with only one event. Wlog, we suppose $S_1, \ldots, S_l$ are the subsets of the decomposition with only one event and $S_{l+1}, \ldots, S_k$ are the subsets with at least two events each.

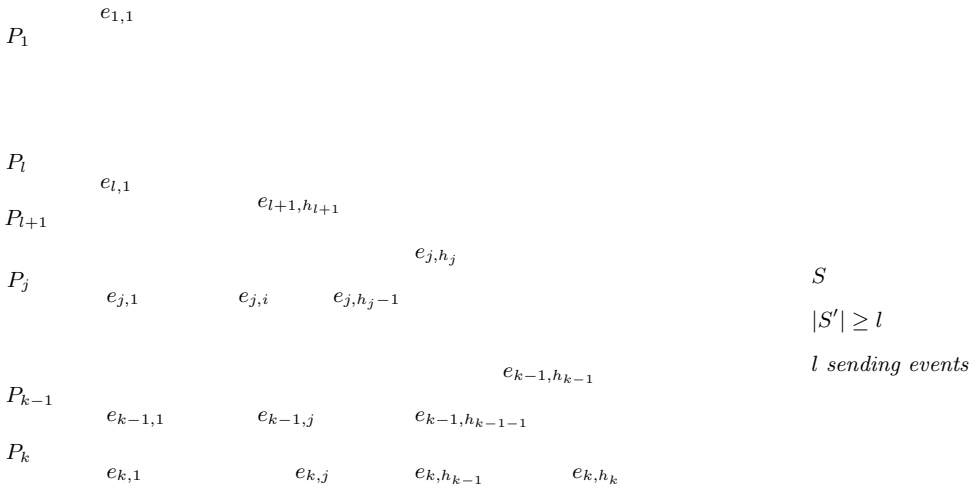


Figure 3.1: Proof of Theorem 3.3.5.

In Figure 3.1 is depicted a computation $\widehat{E} = (E, \prec)$ such that S is actually the causal past of event e_{k,h_k}.

The computation $\widehat{E} = (E, \prec)$ is defined as follows:

1. $e_{j,h_j} \prec_m e_{j+1,h_{j+1}-1}$, for every index $j \in \{l+1, \ldots, k-2\}$, i.e, $\cup_{j=l+1}^{k-2}\{(e_{j,h_j}, e_{j+1,h_{j+1}-1})\} \in \prec$;
2. $e_{k-1,h_{k-1}} \prec_m e_{k,h_k}$, i.e. $(e_{k-1,h_{k-1}}, e_{k,h_k}) \in \prec$;
3. Let the set $S - (\cup_{j=1}^{l} S_i) - (\cup_{j=l+2}^{k-1}\{e_{j,h_j-1}, e_{j,h_j}\}) - \{e_{l+1,h_{l+1}}, e_{k,h_k}\}$ be denoted as S'. There is a sending message from $e_{j,1}$ to some event $e^{(j)} \in S'$, for every $j \in \{1, \ldots, l\}$, i.e., $(e_{j,1}, e^{(j)}) \in \prec$, $\forall j = 1, \ldots, l$. We note that by hypothesis $|S'| = |S| - l - 2(k-l) + 2 \geq (2k-2) - 2k + l + 2 = l$, so there are at least l possible receive events for l sendings of messages. The remaining events are internal events.

It follows that $S = \downarrow (e_{k,h_k}, \prec)$, and, therefore, S is a causal past.

□

Theorem 3.3.5 allows us to compute the number of prefix-closed subsets in $2^E_{\prec_l}$ which are causal pasts, i.e. $| \downarrow [\widehat{\mathcal{E}}]|$. This cardinality is fundamental to obtain both the lower bound on $|T|$ and, consequently, the lower bound on the

amount of information necessary to encode timestamps when the timestamping protocol characterizes causality on-the-fly.

Proposition 3.3.6. *The number of subsets $S \in 2^E_{\prec_l}$ which are not causal pasts is:*

$$|2^E_{\prec_l} \setminus \downarrow [\widehat{\mathcal{E}}]| = 1 + \sum_{k=3}^{n} \binom{n}{k} \binom{2k-3}{k}. \tag{3.3.6}$$

Proof. Every subset $S \in 2^E_{\prec_l} \setminus \downarrow [\widehat{\mathcal{E}}]$ is either the empty set or it has at most $2k-3$ events when its decomposition has $k \geq 3$ nonempty subsets.

By applying basic mathematical enumeration, since

- k nonempty subsets can be on any of n processes, and
- the number of prefix-closed sets S of size h which can be decomposed in k *nonempty* subsets is $\binom{h-1}{h-k}$,

it results that the number required is:

$$\sum_{k=3}^{n} \binom{n}{k} \sum_{h=k}^{2k-3} \binom{h-1}{h-k} = \sum_{k=3}^{n} \binom{n}{k} \sum_{h=0}^{k-3} \binom{h+k-1}{h}.$$

The value $\binom{h+k-1}{h}$, denoted $N(h,k)$, represents the number of prefix-closed subsets of size h whose decomposition has *at most* k nonempty subsets. It can be easily proved that (for every h, k):

$$\binom{h+k-1}{h} = N(h,k) = \sum_{i=0}^{h} N(i,k-1), \tag{3.3.7}$$

so the thesis (3.3.6) follows by considering that both

- the empty set is not a causal past, i.e., $\emptyset \in 2^E_{\prec_l} \setminus \downarrow [\widehat{\mathcal{E}}]$, and
- by Equation (3.3.7), there are $\sum_{k=3}^{n} \binom{n}{k}\binom{2k-3}{k}$ nonempty subsets in $2^E_{\prec_l} \setminus \downarrow [\widehat{\mathcal{E}}]$, as $\sum_{h=0}^{k-3} \binom{h+k-1}{h} = \sum_{h=0}^{k-3} N(h,k) = N(k-3,k+1) = \binom{2k-3}{k}$.

□

Since $|2^E_{\prec_l}| = (m+1)^n$, as an immediate consequence one has the following result.

Corollary 3.3.7. *If n processes generate m events each, then*

$$|\downarrow [\widehat{\mathcal{E}}]| = (m+1)^n - 1 - \sum_{k=3}^{n} \binom{n}{k} \binom{2k-3}{k}.$$

Proof. If each process generates m events, we have $(m+1)^n$ different $\prec_l$-prefix-closed subsets of events. The thesis follows by Equation (3.3.6), by considering that there are $1 + \sum_{k=3}^{n} \binom{n}{k}\binom{2k-3}{k}$ prefix-closed subsets in $2^E_{\prec_l}$ which cannot be causal pasts. □

3.3.3 Timestamps Characterization

In previous sections we pointed out that a timestamping protocol which characterizes causality on-the-fly must associate with each event e a timestamp which is really an encoding of its causal past. From an operational point of view, timestamps evolve dynamically during the computation at the same time as causal pasts.

Let $\chi :\downarrow [\widehat{\mathcal{E}}] \rightarrow \phi[\widehat{\mathcal{E}}]$ be the encoding function (which is an isomorphism of partially ordered sets) which characterizes the protocol which executes timestamping of events (see Corollary 3.3.3). At each time, $\phi(e, \prec)$ must actually be the encoding $\chi(\downarrow (e, \prec))$ of the causal past of e.

Since each causal past evolves during the computation according the following rule:

$$\downarrow (e, \prec) \;=\; (\downarrow (pred_l(e), \prec) \;\cup\; \{e\}) \cup \;\downarrow (pred_m(e), \prec) \tag{3.3.8}$$

(being $\downarrow (pred_m(e), \prec) = \emptyset$ when e is not a receive event), the timestamps domain $(T, <)$ must be equipped with an operator $*$ which permits to assign timestamps to events in an incremental way, by satisfying the following equality:

$$\chi(\downarrow (e, \prec)) \;=\; \chi(\downarrow (pred_l(e), \prec) \;\cup\; \{e\}) * \chi(\downarrow (pred_m(e), \prec)). \tag{3.3.9}$$

Obviously, this update is correct only if, during the evolution of any distributed computation $\widehat{E} \in \widehat{\mathcal{E}}$, the operator $*$ between timestamps is consistent with the union operator between causal pasts. That is, for every pair of causal pasts $S, S' \in\downarrow [\widehat{\mathcal{E}}]$:

$$\chi(S \cup S') = \chi(S) * \chi(S'). \tag{3.3.10}$$

So, the timestamps domain $(T, <)$ must be equipped of an operator $* : T \times T \rightarrow T$, which is consistent with the union between causal pasts (Equation (3.3.10)).

Closure Properties of Set $\downarrow [\widehat{\mathcal{E}}]$

By providing appropriate counterexamples, we will prove that the set $\downarrow [\widehat{\mathcal{E}}]$ of all causal pasts is not closed with respect to the union and minus operators.

Proposition 3.3.8. *The set* $\downarrow [\widehat{\mathcal{E}}]$ *is not closed with respect to the union operator, i.e.,* $\forall S, S' \in 2^E_{\prec}$, $S, S' \in\downarrow [\widehat{\mathcal{E}}] \not\Rightarrow S \cup S' \in\downarrow [\widehat{\mathcal{E}}]$.

Proof. Let $S \in\downarrow [\widehat{\mathcal{E}}]$ be a subset of events with only one element on a process P_i, i.e., $S = \{e_{i,1}\}$. Moreover, let $S' \in\downarrow [\widehat{\mathcal{E}}]$ be a subset of two events on two processes P_j and P_k, with $i \neq j \neq k$, i.e., $S' = \{e_{j,1}, e_{k,1}\}$. Such subsets are depicted in Figure 3.2.

Figure 3.2: Proof of Proposition 3.3.8

The union set $S \cup S'$ is a $\prec_l$-prefix-closed subset with three events on three processes. Namely, it is $S \cup S' = \{e_{i,1}, e_{j,1}, e_{k,1}\}$, with $i \neq j \neq k$.

Theorem 3.3.5 implies $S \cup S' \notin\downarrow [\widehat{\mathcal{E}}]$, since when a subset has $k \geq 3$ nonempty subsets in its decomposition then, in order to be a causal past, it must have at least $2(k-1)$ events. In this example we have that $k = 3$, but $|S \cup S'| < 4$. □

Proposition 3.3.9. *The set* $\downarrow [\widehat{\mathcal{E}}]$ *is not closed with respect to the minus operator, i.e.,* $\forall S, S' \in 2^E_{\prec}$, $S, S' \in\downarrow [\widehat{\mathcal{E}}] \not\Rightarrow S \setminus S' \in\downarrow [\widehat{\mathcal{E}}]$.

Proof. Let us consider the following two subsets of events, depicted in Figure 3.3:

- $S = \{e_{1,1}, e_{2,1}, e_{3,1}, e_{4,1}, e_{4,2}, e_{5,1}, e_{5,2}, e_{5,3}, e_{5,4}\}$;
- $S' = \{e_{5,1}, e_{5,2}, e_{5,3}, e_{5,4}\}$.

Theorem 3.3.5 implies that $S, S' \in\downarrow [\widehat{\mathcal{E}}]$ and yet one has

$$S \setminus S' = \{e_{1,1}, e_{2,1}, e_{3,1}, e_{4,1}, e_{4,2}\} \notin\downarrow [\widehat{\mathcal{E}}].$$

In fact, there are $k = 4$ nonempty subsets in its decomposition, but its cardinality is $|S \setminus S'| = 5 < 2(k-1) = 6$. Hence, $S \setminus S'$ cannot be a causal past. □

Figure 3.3: Proof of Proposition 3.3.9

3.3.4 Efficient Timestamping Protocols: A Formal Framework

In this section we provide a formal framework for effective timestamping protocols which manage the minimum information required to characterize causality on-the-fly.

Definition 3.3.1. A timestamping protocol $\mathcal{A}$ *characterizes causality on-the-fly* if, given a partially ordered set $(T, <)$ equipped with an operator $*$:

- there exists an injective *encoding function* $\chi :\downarrow [\widehat{\mathcal{E}}] \rightarrow T$ which satisfies the following properties for every pair of prefix-closed subsets of events $S, S' \in \downarrow [\widehat{\mathcal{E}}]$:
  - $S \subseteq S' \;\Leftrightarrow\; \chi(S) < \chi(S')$;
  - $\forall S, S' \in \downarrow [\widehat{\mathcal{E}}], \quad S \cup S' \in \downarrow [\widehat{\mathcal{E}}] \;\Rightarrow\; \chi(S \cup S') = \chi(S) * \chi(S')$.
- the timestamping function is $\phi = \chi \circ \downarrow$.

We note that, by definition, if a timestamping protocol meets previous characterization, as a consequence the timestamping function ϕ satisfies the strong completeness property 3.1.1.

The Vector Clocks case

Vector Clocks system was analyzed in Section 3.2. As remarked in [36], a vector clock associated with an event e, denoted $e.VC$, is an encoding of the causal past of event e. Moreover, the timestamps domain is equipped with the operator max which is consistent with the union operator between prefix-closed subsets of events in $2^E_{\prec_l}$.

Vector Clocks as Encoding of Causal Pasts

As E_i is a sequence of events $\{e_{i,1}, \ldots, e_{i,x}, \ldots\}$, the fact that $e_{i,x} \in{\downarrow_i}(e)$ implies $e_{i,1}, \ldots, e_{i,x-1} \in{\downarrow_i}(e)$. Therefore, for every i the set ${\downarrow_i}(e)$ is sufficiently characterized by the largest index among its members, i.e., its cardinality. Thus, if n is the number of processes involved in the distributed computation, a causal past ${\downarrow}(e) = {\downarrow_1}(e) \uplus {\downarrow_2}(e) \uplus \ldots \uplus {\downarrow_n}(e)$ can be *uniquely* represented by a vector clock, where

$$\forall i = 1, \ldots, n, \quad e.VC[i] = |{\downarrow_i}(e)|. \tag{3.3.11}$$

Conversely, the use of n-dimensional integer vectors can be generalized in a straight-forward way to represent an arbitrary prefix-closed set of events under $S \in 2^E_{\prec_l}$, again by taking the locally largest event index: given $V \in \mathbb{N}^n$, the set of events S represented by V, denoted $S(V)$, can be defined as the disjoint union $S(V) = \biguplus_{i=1}^{n} S_i(V)$, where

$$\forall i = 1, \ldots, n, \quad S_i(V) = S(V) \cap E_i = \{e_{i,1}, \ldots, e_{i,V[i]}\}. \tag{3.3.12}$$

So, the Vector Clocks protocol is characterized by an injective encoding function $\chi :{\downarrow}\,[\widehat{\mathcal{E}}] \rightarrow \mathbb{N}^n$ which associates a vector clock with each causal past $S \in{\downarrow}\,[\widehat{\mathcal{E}}]$ in this way:

$$\chi(S)[i] = |S \cap E_i|, \ \forall i = 1, \ldots, n. \tag{3.3.13}$$

Moreover, the *max* operator is consistent with the union operator. In fact:

$$\chi(S \cup S') = \max(\chi(S), \chi(S')),$$

i.e. $\chi(S \cup S')[i] = \max(\chi(S)[i], \chi(S')[i])$ for each $i = 1, \ldots, n$.

3.3.5 Message Timestamps Characterization

In Section 3.1.1 we stated that a timestamping protocol is also characterized by a set of rules implementing the protocol, which decide control information to piggyback on outgoing messages, called message timestamps, in order to update timestamps associated with events.

The aim of this section is to investigate the minimum amount of information to piggyback on messages, when the timestamping protocol characterizes causality on-the-fly. To achieve this purpose, because of the bijection between timestamps and causal pasts and by considering that timestamps are incrementally updated at the same time as causal pasts according to the rule (3.3.8), we can state message timestamps properties in terms of sets of events

to be piggybacked in order to update the causal past of the event actually produced.

More formally, let $\gamma(send(m), \prec) \in 2^E_{\prec_l}$ denote the control information piggybacked on the message m.

Notations. Given a distributed computation $\widehat{E} = (E, \prec)$, we will denote as:

- $\gamma(\widehat{E}) = \bigcup_{e \in E}\{\gamma(e, \prec)\}$ (if e is not a send event we assume $\{\gamma(e, \prec)\} = \emptyset$) the set of message timestamps piggybacked during the execution of $(E, \prec)$;
- $\gamma[\widehat{\mathcal{E}}] = \bigcup_{\widehat{E} \in \widehat{\mathcal{E}}} \gamma(\widehat{E})$ the set of all the message timestamps which could be piggybacked on messages during several executions of the program.

Proposition 3.3.10. *A timestamping protocol characterizes causality on-the-fly if and only if for every message m sent during the evolution of a computation, it is*

$$\gamma(send(m), \prec) \supseteq \downarrow (send(m), \prec) \setminus E_{receiver(m)}. \tag{3.3.14}$$

Proof. For every computation $\widehat{E} \in \widehat{\mathcal{E}}$, we recall the causal past of a receive event $receive(m)$ is incrementally defined as $\downarrow (receive(m), \prec) = \downarrow (pred_l(receive(m)), \prec) \cup \downarrow (send(m), \prec) \cup \{receive(m)\}$.

Since $(\downarrow (send(m), \prec) \cap E_{receiver(m)}) \subset (\downarrow (pred_l(receive(m)), \prec) \cup \{receive(m)\}) \cap E_{receiver(m)}$, in order to incrementally compute $\downarrow (receive(m))$, it is necessary and sufficient to piggyback on m the following set of events:

$$\downarrow (send(m), \prec) \setminus E_{receiver(m)}. \tag{3.3.15}$$

□

As a consequence, every message has to piggyback only a set of events involving at most $n-1$ processes. We remark that we will implicitly consider these $n-1$ processes as fixed with respect to the receiving process.

We will denote as $2^E_{\prec_l}|_{n-1}$ the set of prefix-closed subsets of events of $(E, \prec_l)$ which involve $n-1$ (fixed) processes. The value $|2^E_{\prec_l}|_{n-1}| = (m+1)^{n-1}$ represents a *first lower bound* on the cardinality of the set of all the message timestamps $\gamma[\widehat{\mathcal{E}}]$, i.e., $|2^E_{\prec_l}|_{n-1}| \geq |\gamma[\widehat{\mathcal{E}}]|$.

In order to provide a precise reckoning of all the message timestamps in $\gamma[\widehat{\mathcal{E}}]$, we prove a property that every prefix-closed subset in $2^E_{\prec_l}|_{n-1}$ has to satisfy in order to be a message timestamp.

Theorem 3.3.11. *A set $S \in 2^E_{\prec_l}|_{n-1}$ is a message timestamp ($S \in \gamma[\widehat{\mathcal{E}}]$) if and only if $S \neq \emptyset$, and when the number k of nonempty subsets in its decomposition is at least 2, $|S| \geq k+1$.*

Proof. The proof is made of two parts: necessity and sufficiency.

Necessity. If S is a message timestamp ($S \in \gamma[\widehat{\mathcal{E}}]$), then $S = \gamma(send(m), \prec)$, for some sending event of a message m produced during the evolution of a computation $(E, \prec) \in \widehat{\mathcal{E}}$. In such a case, at least the send event $send(m)$ has to belong to S, so a message timestamp cannot be an empty set. If $k \geq 2$ processes have events in S (we recall that S does not contain events in the receiving process) then in the sending process $P_{sender(m)}$ at least a receiving event $receive(m')$ has to locally precede $send(m)$, in order to establish a dependency. Hence, $k+1$ is the minimum number of events in $S \in \gamma[\widehat{\mathcal{E}}]$ when $k \geq 2$.

Sufficiency. Let $S \in 2^E_{\prec_l}|_{n-1}$ be a nonempty set and k ($k \geq 1$) be the number of nonempty subsets in its decomposition. We can distinguish both cases $k = 1$ and $k \geq 2$.

If $k = 1$, let P_j be the process whose events are in S ($S = S_j$). Following (3.3.8) and (3.3.15), in a distributed computation where $e_{j,|S|}$ is a send event and all the events which locally precede $e_{j,|S|}$ are internal events, the whole set S has to be piggybacked on the message in order update the causal past of the receive event.

If $k \geq 2$ and $|S| \geq k+1$, as, by definition, S involves at most $n-1$ processes, there is at least one process in $\mathcal{P}$ whose events are not in S. Let P_i be any process without events in S. Moreover, as it is $|S| \geq k+1$, there exists at least one process with at least two events in S. Let P_j denote any such process. Let $S_{i_1}, S_{i_2}, \ldots, S_{i_{k-1}}$ be all the other $k-1$ nonempty subsets in the decomposition of S.

In order to show that $S \in \gamma[\widehat{\mathcal{E}}]$, we have to prove the existence of a distributed computation in which the set S could be really piggybacked on a message m.

In Figure 3.4 it is depicted an example of such a computation.

More formally, let $\widehat{E}$ be a distributed computation defined as follows:

- $e_{j,|S_j|}$ is a sending event of a message m to P_i;
- events $e_{i_h,|S_{i_h}|}$, for $h = 1, \ldots, k-1$, are sending events of messages m_{i_h};
- as regards their corresponding receive events: if $|S_j| \geq k$, they are produced by P_j itself before producing $e_{j,|S_j|}$; otherwise, we consider $|S_j|-2$ receive events on P_j before ($e_{j,|S_j|}$) and all the remaining $k - |S_j| - 3$

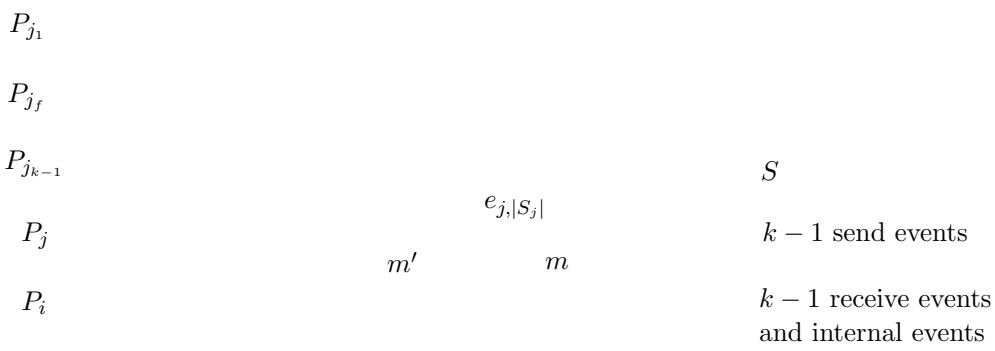


Figure 3.4: Proof of Theorem 3.3.11

receive events on process P_i, before producing a send event $send(m')$ to $e_{j,|S_j|-1}$. Clearly, $send(m') \prec_l receive(m)$.

By Equation (3.3.15), during the execution the set $S = \downarrow (e_{j,|S_j|}) \setminus E_i$ must be piggybacked on m. □

Remark 3.3.1. In Proposition 3.3.9, we proved the set of causal pasts $\downarrow [\widehat{\mathcal{E}}]$ is not closed with respect to the minus $\setminus$ operator. Previous theorem confirms this result. In fact, each message timestamp is an encoding of a prefix-closed set of events which is the difference between (i) the causal past of the sending event and (ii) the set of events in the causal past produced by the receiver process, which could be a causal past in a different distributed computation. Theorem 3.3.11 shows that this difference not necessarily is a causal past. In fact, a set S with $k+1$ events which involves $k \geq 3$ processes could be really a message timestamp, but cannot be a causal past, as stated in Theorem 3.3.5.

Theorem 3.3.11 is fundamental to provide the exact cardinality of the set $\gamma[\widehat{\mathcal{E}}]$ of all message timestamps. In particular, we can compute the number of $\prec_l$-prefix-closed subsets which cannot be message timestamps.

Proposition 3.3.12. *The number of subsets $S \in 2^E_{\prec_l}|_{n-1}$ which are not message timestamps is:*

$$|2^E_{\prec_l}|_{n-1} \setminus \gamma[\widehat{\mathcal{E}}]| = 2^{n-1} - n + 1. \tag{3.3.16}$$

Proof. Each subset $S \in 2^E_{\prec_l}|_{n-1} \setminus \gamma[\widehat{\mathcal{E}}]$ is either the empty set or it has k events when there are $k \geq 2$ nonempty subsets in its decomposition. Since these k nonempty subsets can be on any of $n-1$ processes and k ranges from 2 to $n-1$,

the thesis follows by considering that $|2^E_{\prec_l}|_{n-1} \setminus \gamma[\widehat{\mathcal{E}}]|$ is equal to the cardinality of power sets of sets of $n-1$ elements (i.e., 2^{n-1}), by excluding all the subsets of cardinality one (i.e., $n-1$ subsets), which are message timestamps. □

As $|2^E_{\prec_l}|_{n-1}| = (m+1)^{n-1}$, Proposition 3.3.12 directly implies that:

Corollary 3.3.13. *If n processes generate m events each, then*

$$|\gamma[\widehat{\mathcal{E}}]| = (m+1)^{n-1} - 2^{n-1} + n - 1. \tag{3.3.17}$$

3.3.6 Lower Bounds on The Amount of Information Managed by Timestamping Protocols

Later on we provide lower bounds on the amount of information necessary to encode timestamps and message timestamps, for any timestamping protocol which characterizes causality on-the-fly.

Corollary 3.3.14. *If a timestamping protocol characterizes causality the* encoding of each timestamp in T *requires at least the following number of bits:*

$$\lceil \log_2 ((m+1)^n - \sum_{k=3}^{n} \binom{n}{k} \binom{2k-3}{k}) \rceil.$$

Proof. The thesis is a direct consequence of the bijection between causal pasts and timestamps, and Corollary 3.3.7. Since $|T| \geq |\phi[\widehat{\mathcal{E}}]| = |\downarrow [\widehat{\mathcal{E}}]|$, and $|\downarrow [\widehat{\mathcal{E}}]| = (m+1)^n - 1 - \sum_{k=3}^{n} \binom{n}{k}\binom{2k-3}{k}$, to encode such a number of elements we need at least $\lceil \log_2 |\downarrow [\widehat{\mathcal{E}}]| \rceil = \lceil \log_2 ((m+1)^n - 1 - \sum_{k=3}^{n} \binom{n}{k}\binom{2k-3}{k}) \rceil$ bits. The result follows by considering that processes use an initial timestamp which does not belong to the set $\phi[\widehat{\mathcal{E}}]$, so we point out that $|T| \geq |\phi[\widehat{\mathcal{E}}]| + 1$. □

Corollary 3.3.13 directly implies the following lower bound on the amount of information that has to be piggybacked in order to represent causality in an isomorphic way.

Corollary 3.3.15. *If a timestamping protocol characterizes causality on-the-fly the* encoding of each message timestamp *requires at least the following number of bits:*

$$\lceil \log_2 ((m+1)^{n-1} - 2^{n-1} + n - 1) \rceil.$$

Analysis of Lower Bounds

Corollaries 3.3.14 and 3.3.15 give a lower bound on amount of information necessary to characterize causality on-the-fly.

This result relates to vector clocks [17, 30] as follows: Charron-Bost in [14] showed that *if a timestamping protocol structures timestamps as vectors of integers*, then it can characterize causality on-the-fly only if vectors have at least size n, being n the number of processes of the computation, as in vector clocks. This directly provides an upper bound on the amount of information necessary to encode timestamps.
In fact, if n processes generate m events each, vector clocks protocol is characterized by a timestamps domain $(T, <)$ actually defined as the set $T = \{0, \ldots, m\}^n$. So, $|T| = (m+1)^n$, and the number of bits necessary to encode each vector is

$$\lceil \log_2 (m+1)^n \rceil$$

which is really an upper bound on the number of bits necessary to encode timestamps when using a protocol characterizing causality on-the-fly.

Vector clocks also provide an upper bound on the amount of information necessary to encode message timestamps.

The canonical implementation of vector clocks protocol presented in Section 3.2.1 can be easily improved by piggybacking on outgoing messages only a vector of dimension $n-1$, by excluding by the local vector clock the entry of the receiver process. The set of message timestamps is then really the set of $(m+1)^{n-1}$ vectors of integers of dimension $(n-1)$. As a consequence, an upper bound on the minimal amount of bits to be piggybacked on messages is

$$\lceil \log_2 (m+1)^{n-1} \rceil.$$

A system of vector clocks uses more than the necessary information. This is due to the fact that it actually encodes all the prefix-closed subsets of events under $\prec_l$, i.e.,

- as regards timestamps, it also encodes sets in $2^E_{\prec_l} \setminus \phi[\widehat{\mathcal{E}}]$, that is $\prec_l$-prefix-closed subsets which cannot be causal pasts;
- as regards message timestamps, it also encodes sets in $2^E_{\prec_l}|_{n-1} \setminus \gamma[\widehat{\mathcal{E}}]$, that is $\prec_l$-prefix-closed subsets on $n-1$ processes which cannot be message timestamps.

This arises an interesting question on the tightness of these gaps. In other words, are vector clocks an optimal encoding of timestamps and message timestamps?

From an operational point of view, different encodings that exclude the sets $2^E_{\prec_l} \setminus \phi[\widehat{\mathcal{E}}]$ and $2^E_{\prec_l}|_{n-1} \setminus \gamma[\widehat{\mathcal{E}}]$ seem to be impracticable. As a consequence a timestamping protocol based on vector clocks seems to provide the closest encoding to the minimal quantity of information required.

From a theoretical point of view, we have the following fact [14, 36]:

Proposition 3.3.16. *Any timestamping protocol that characterizes causality on-the-fly must use a timestamps domain $(T, <)$ which is a partial order of dimension at least n, if n processes are involved in the computation.*

In addition, we stated that the timestamps domain $(T, <)$ must be equipped of an operator $* : T \times T \to T$, which is consistent with the union between causal pasts (Equation (3.3.10)). So, a strategy to answer to this question could be to take into account the cost of updating a data structure (e.g., vectors of integers) that encodes causal pasts in an incremental way.

3.4 Conclusions and Future Work

Main result of this chapter concerned the study of the minimum amount of information managed by any timestamping protocol which characterizes causality on-the-fly. Lower bounds on these information were expressed in terms of amount of non-structured information (i.e., number of bits) necessary to encode timestamps and message timestamps when considering a timestamping protocol which captures causality in an isomorphic way.

In order to reach our aim we first characterized a few properties which prefix-closed subsets of events of $(E, \prec_l)$ have to satisfy in order to be causal pasts. Next, we provided a precise counting of the number of all the causal pasts which could be associated with events during any arbitrary execution of a program $\mathcal{P}$. Next, we counted timestamps in $\phi[\widehat{\mathcal{E}}]$ and message timestamps in $\gamma[\widehat{\mathcal{E}}]$. Lower bounds are a direct consequence of this reckoning.

We remark that the result we concerned with in this chapter is more general than the well-known Charron-Bost's result [13], which considers only the minimum size of structured information. In fact, it states that *when timestamps are structured as vectors of integers*, the size n is necessary to characterize causality on-the-fly, n being the number of processes involved in the computation.

A part of the results of this chapter appears in [4, 5, 7].

As regards future work, the most important question is to prove that vector clocks protocol is really the *optimal protocol*. This would confirm the Charron-Bost's result and, more important, it would exclude the existence of a protocol

different from vector clocks, which is able to structure timestamps by using less bits. The results obtained in this chapter suggest an effort to prove that the gap between lower bounds and upper bounds (provided by vector clocks), is really tight. We remark the lower bounds are already very close to upper bounds.

Chapter 4

Answers to Scalability Issue

4.1 Introduction

Vector Clocks protocol [17, 30] gives rice to scalability problems, since each message has to piggyback a vector whose size is given by the number n of processes forming the distributed computation. The scalability issue does not concern vectors stored in local memory as, nowadays, local memory is cheap and available on-board in large quantity. In a distributed setting, scalability is critical with respect to the control information attached to messages (message overhead). Having a bounded, possibly fixed, control information size, would help protocol designers to develop efficient protocols for tracking causal dependencies.

Charron-Bost in [14] proves this size is a requirement necessary for the on-the-fly characterization of causality, if timestamps are structured as vectors of integers. Moreover, in a more recent study [5] presented in previous chapter it has been showed that a timestamping protocol which characterizes causality on-the-fly has to store locally at each process and to attach on each application message an amount of unstructured information which is slightly smaller than a vector of integers of size n.

Therefore, the answer to the scalability issue in a system of vector clocks has to be found by exploiting some tradeoffs. This chapter provides a comprehensive list of the tradeoffs in a vector clocks system answering to the scalability issue.

Up to now, two tradeoffs have been pointed out in the literature:

1. *trading* message overhead *versus* local memory overhead. This class includes "Efficient implementations of vector clocks" introduced in [37] and in a recent study [22] presented in the following chapter. These implementations try to move locally as many as possible of the complexity of managing a vector clocks system while maintaining the on-the-fly

characterization of causality. However, in the worst case the information attached to messages is $O(n)$.Efficient implementations can be used when the interaction between processes is localized. In this case the communication overhead can be cut substantially.

2. *trading* message overhead and local memory overhead *versus* missing some concurrencies between events (i.e., concurrent events can be perceived as ordered). This tradeoff is exploited by Plausible clocks [38], presented in Section 4.4. This clocks system bounds the size of message overhead and local overhead to some integer k less than n. As a consequence, plausible clocks does not characterize causality on-the-fly, as the set of timestamps assigned to events is an extension of the partial order of the computation. Such clocks are useful in systems where ordering of concurrent events only impacts performance and not correctness. This is true for many consistency maintenance and resource allocation algorithms.

In this chapter, we point out a third tradeoff: *trading message overhead versus detection time in causality tracking.* i.e., the clock system bounds the size of message overhead to some integer k less than n at the expenses of a *delay* at the checker to detect the causal relation (or the concurrency) between any pair of events e and e' (non-on-the-fly causality detection). This delay is due to the fact that a checker needs to receive timestamps of other events (distinct from the ones of e and e') before detecting the causality relation.

Figure 4.1: Timestamping Protocols: derived from vector clocks

In the chapter, a property is introduced, namely *Consistency Property with Delay* (dC-P), that captures this new tradeoff. More formally, this property states that timestamps have to be assigned to events from a suitable set D and that the transitive closure of the timestamps has to be a partial order isomorphic to the one produced by the computation. Then, we present *k-Dependency Vectors*, a scalable protocol for causality-tracking that implements the dC-P. It is interesting to remark that if $k = n$, k-dependency vectors become vector clocks. If $k = 1$, k-dependency vectors boil down to a timestamping protocol, namely Direct Dependencies protocol, proposed by Fowler and Zwaenepoel in [18] in the context of distributed debugging. We present several strategies to select k entries of the local dependency vector. In particular, we discuss the relationship between the detection delay and message deliveries out of causal ordering [11, 2] and we show the impact of these strategies on length of the detection delay of a causality relation.

The remaining of this chapter is composed of six sections. Section 4.2 surveys the efficient implementation of vector clock proposed by Singhal and Kshemkalyani. The consistency property C-P and the plausible clocks protocol are presented in Sections 4.3 and 4.4. Section 4.5 introduces the dC-P, i.e., the "Completeness Property with Delay" and Section 4.6 presents the k-dependency vectors protocol and formally proves that this protocol satisfies the dC-P. Finally, in the same section, the selection strategies are introduced and discussed. Section 4.7 concludes the chapter by proposing a table that summarizes existing tradeoffs which answer to the scalability issue of a system of vector clocks.

4.2 An efficient Implementation of Vector Clock

Many problems which require dependency, such as consistent global state selection [12, 23], detection of obsolete data [28], replicated data management [32] and distributed debugging [18], require the capability to detect *concurrency or causal dependency* of events forming a distributed computation concurrently with the execution of the computation.

In [37] Singhal and Kshemkalyani proposed an improvement of the implementation of vector clocks that, when FIFO communication channels are available, reduces in the average case the size of information piggybacked on messages. This technique is based on empirical observation that a process tends to interact frequently with only a small set of other processes and, then, between two successive sending events from process P_i to process P_j only a few entries of the vector clock are expected to change.

This efficient implementation is based on the following idea. When P_i sends a message m to P_j, it may attach to the message only the entries that have

been modified since its last sending to P_j. Namely, if $e = send(m)$ and $e' = send(m')$ are sending events from P_i to P_j and $e \prec_l e'$, this is equivalent to piggyback on the message only an encode of the difference between their causal pasts, i.e., $\downarrow (e') \setminus \downarrow (e)$, which represents the part of the causal past unknown to P_j when producing $receive(m')$ (see Figure 4.2). This observation gave rise to the "Differential Technique" used by Singhal and Kshemkalyani [37].

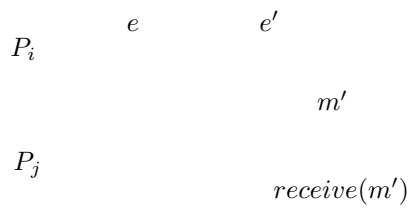


Figure 4.2: An example of the use of the "Differential Technique"

The reduction of the communication overhead is obtained at the expense of an extra storage space at each process. However, this is not serious because main memory is cheap and is available in large quantities, and it is more desirable to reduce traffic on a communication network whose capacity is limited and is often the bottleneck.

4.2.1 The Protocol

From an operational point of view, each process P_i keeps two additional vectors of size n:

- $LU_i \in \mathbb{N}^n$, called *Last Update* vector, such that $LU_i[k] = x$ means that the event of P_i that occurred when $VC_i[k]$ has been updated for the last time is its x-th event, i.e., it occurred when $VC_i[i] = x$;
- $LS_i \in \mathbb{N}^n$, called *Last Sent* vector, such that $LS_i[k] = x$ means that the last send event to P_k is the x-th event of P_i, i.e., it occurred when $VC_i[i] = x$.

The fact that since the last communication from process P_i to process P_j, only those elements $VC_i[k]$ have changed for which $LS_i[j] < LU_i[k]$ suggests an improvement of communication rules [R2] and [R3] of the canonical implementation of Vector Clocks protocol introduced in Section 3.2.1.

Then, by using extra local vectors LU_i and LS_i, vector clock V_i is updated according the following rules:

[R0] $VC_i[1..n]$ is initialized to $[0, \ldots, 0]$.

[R1] Each time it produces an event e, P_i increments its vector clock entry $VC_i[i]$ and associates with e the timestamp $e.VC = VC_i$.

[R2] When P_i sends a message m to P_j, it piggybacks a list of pairs $m.L = \{(k, VC_i[k]) \mid LS_i[j] < LU_i[k])\}$;

[R3] When P_i receives a message m with a list $m.L$ it updates its vector clock in the following way: $\forall k : (k, VC[k]) \in m.L$, $VC_i[k] := max(VC_i[k], VC[k])$.

The drawback. This technique reduces in the average case the size of the message timestamps, however this size could grow up to n. Moreover, each process maintains three vectors of n integers.

4.3 Consistency Property

In some applications we are only interested in an approximation of the causality relation. For instance, to solve the mutual exclusion problem [27] it is not crucial to design a clock that captures all the concurrencies.

In general, protocols which only miss some concurrencies between events can be used in systems where ordering of concurrent events only impacts performance. We say that such protocols and their corresponding timestamps system, which capture the order between causally related events but do not necessarily detect concurrency, are *consistent with causality* [27] (or capture causality).

This is usually formalized by requiring that the following *Consistency Property* (C-P) is satisfied:

Property 4.3.1 (C-P).

$$\forall \prec \in \mathcal{C}[E],\ \forall e, e' \in E,\quad e \prec e' \Rightarrow \phi(e, \prec) < \phi(e', \prec). \tag{4.3.1}$$

Proposition 4.3.2. *If a timestamping protocol is consistent with causality then for every pair of distinct events $e, e' \in E$:*

- $\phi(e, \prec) = \phi(e', \prec) \ \Rightarrow e \| e'$
- $\phi(e, \prec) < \phi(e', \prec) \Rightarrow (e \prec e') \vee (e \| e')$

Proof. The thesis directly follows by the C-P. □

As a consequence, the timestamps system $(\phi[\widehat{E}], <)$ associated with events places on E a partial order which is an *order extension of* $\widehat{E} = (E, \prec)$. In fact, events e, e' which are concurrent could appear related, if they have different timestamps ($\phi(e, \prec) < \phi(e', \prec)$ or vice-versa).

Experimental results [38] show that with $n = 100$ and $2 < k < 5$, the percentage of situations in which the checker reports a dependency between two events e and e', while actually e and e' are concurrent, is less than 10%.

On the contrary, the system never confuses the direction of causality between any two causally ordered events. At the same time if the system states that $e \| e'$, this *necessarily is correct.*

4.4 Plausible Clocks

Plausible clocks, introduced by Torres and Ahamad [38], represent a class of protocols that are only consistent with causality. However, the loss of accuracy in detecting concurrencies between events is compensated by the potential for scalability, as they can be implemented using constant size structures, and by the saving in communications overhead.

This class of protocols exploits the tradeoff between the size k of vector timestamps and the accuracy with which they detect dependencies between events. By increasing the number of components in vector clocks, we are increasing the level of ordering accuracy.

4.4.1 The Protocol

Plausible clocks protocol provides a simple mechanism that allows to associate with each event a vector of k integers as timestamp, with $1 \leq k \leq n$, that is it is characterized by $T = \mathbb{N}^k$ as its timestamps domain[1].

Let $k \in \{1 \ \ldots, n\}$ be a given constant and $f_k : \{1, \ldots, n\} \to \{1, \ldots, k\}$ be any deterministic function.

Each process P_i has a local k-dimensional vector of integers $L_i[1..k]$, and, for each process P_i, the function f_k associates with i the entry $f_k(i)$ of the local vector. This implies that multiple processes share the same entry in the local vectors.

Such a timestamping protocol is implemented in the following way:

[1] Details of the timestamp's structure are left open [38].

- initially, $L_i = 0$ for each index $i = 1, \ldots, k$;
- each process P_i increments its local vector L_i when executing an internal event or a send event: $L_i[f_k(i)] := L_i[f_k(i)] + 1$;
- on sending a message m, process P_i carries on m the current value of L_i, denoted by $m.L$;
- on receiving a message m with the value $m.L$, process P_i updates L_i in the following way:
  - $L_i := \max(m.L, L_i)$;
  - $L_i[f_k(i)] := L_i[f_k(i)] + 1$.

Lamport's Clock protocol: the case $k = 1$

If $k = 1$, then for each $i = 1, \ldots, n,\ f_k(i) = 1$ and all processes share the unique entry of the (degenerated) local vector. The resulting timestamping protocol is nothing else than the well known Lamport's Clock protocol, introduced by Lamport in 1978 [27]. Let $e.L$ be the integer timestamp associated with event e.
A system of Lamport's clocks places on the set E a *total order* $\prec_L$ defined as follows [27]: $\forall\ e,\ e' \in E$,

$$e \prec_L e' \Leftrightarrow\ (e.L < e'.L)\ \vee\ (e.L = e'.L\ \wedge\ e.producer < e'.producer)$$

As a consequence, $(E, \prec_L)$ is a linear extension of $(E, \prec)$.

Remark 4.4.1. We cannot expect the converse of consistency property C-P to hold as well, since that would imply that any two concurrent events must occur at the same time, that is $e \| e' \Rightarrow e.L = e'.L$.

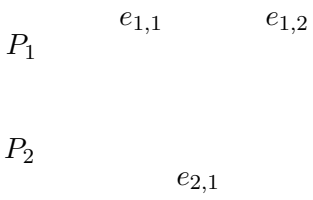


Figure 4.3: Lamport's Clock protocol: a remark on the converse of the C-P

For example, in the distributed computation depicted in Figure 4.3, being $e_{2,1} \| e_{1,1}$ and $e_{2,1} \| e_{1,2}$, the converse of the property 4.3.1 (C-P) would imply $e_{1,1}.L = e_{1,2}.L = e_{2,1}.L$. Nevertheless, this would contradict the C-P, as $e_{1,1} \prec e_{1,2} \Rightarrow e_{1,1}.L < e_{1,2}.L$.

Vector Clocks: the case $k = n$

If $k = n$ and we consider $f_k(i) = i$ for each $i = 1, \ldots, n$, then we get classic vector clocks, presented in Section 3.2. In this case the entry i is not shared with any process, so only P_i can entail its increase.

4.5 Completeness Property with Delay

Protocols which are consistent with causality [38] allow to obtain an order extension of the computation, so events which are concurrent may appear causally dependent. For some applications like mutual exclusion (as described by Lamport himself in [27]) this defect is not noticeable. For other purpose like distributed debugging, however, this is an important defect. On the other hand, the main disadvantage of the Singhal and Kshemkalyani's efficient implementation is that the size of message timestamps is still linear in n in the worst case; also, three vectors per process are needed instead of just one as in the basic approach.

A different approach can be used in some applications where the causal dependencies can be performed off-line, for instance for a trace-based off-line analysis of the causality relation. In these cases one can try to further reduce the size of message timestamps in order to maintain vector clocks at the cost of an increased computational overhead for the calculation of the vector clocks assigned to events.

Let us introduce a new property that a timestamping function should satisfy in order that a timestamp system may preserve causality without addition of causality dependencies between events. The aim is to assign values in T to events so that the transitive closure $(\phi[\widehat{E}], <^+)$ of the timestamps system may be an isomorphic embedding of the computation $(E, \prec)$. We stress that in this case it is required for a timestamping protocol to assign on-the-fly to each event e a timestamp $\phi(e)$ belonging to a suitable domain T such that $(T, <^+)$ has to be a partial order. That is, not necessarily $(T, <)$ is a partial order, but $<$ has to be an antisymmetric relation on T. This can be formalized by requiring the protocol *characterizes causality with delay*, by only failing to represent *transitive dependencies*.

More formally, we say that ϕ *characterizes causality with delay* or *is complete (not necessarily strong)* if ϕ is injective and the following *Completeness Property with Delay* (dC-P) is satisfied:

Property 4.5.1 (dC-P).

$$\forall e, e' \in E : \ [\ e \prec e' \Rightarrow \phi(e) <^+ \phi(e')\ ] \ \wedge \ [\ \phi(e) < \phi(e') \Rightarrow e \prec e'\ ]. \quad (4.5.1)$$

The completeness property with delay (dC-P) corresponds to require that for every computation $\widehat{E} \in \widehat{\mathcal{E}}$ the transitive closure of $(\phi[\widehat{E}], <)$, i.e., the partially ordered set $(\phi[\widehat{E}], <)^+$, is an isomorphic embedding of the computation $\widehat{E}$. This means that there is neither loss of dependency information nor addition of causal dependencies between events, contrary to Plausible Clocks. Property dC-P is a particular case of SC-P (3.1.1) one when considering $(T, <)$ is a partial order.

Proposition 4.5.2. *If a timestamping protocol characterizes causality with delay then for every pair of distinct events $e, e' \in E$:*

- $\phi(e) < \phi(e') \Rightarrow e \prec e'$
- $\phi(e) \| \phi(e') \Rightarrow e \| e' \ \vee \ e \prec e'$.

Proof. The thesis directly follows by the dC-P. □

As a consequence, when $\phi(e)$ and $\phi(e')$ are incomparable, in order to decide the exact relation between events e and e', the checker has to analyze all the possible transitive dependencies for checking if $\phi(e) <^+ \phi(e')$, by backward exploring timestamps associated with events in the causal past of e': when there is no timestamp $\phi(e'')$ such that $\phi(e) <^+ \phi(e'') <^+ \phi(e')$, it can conclude that really e and e' are concurrent, $e \prec e'$ otherwise.

4.5.1 Detection Delay

To answer the question "*given a pair of events e and e', does e causally precede e'?*", the checker might need to analyze some timestamps associated with events e'' belonging to $\downarrow(e') - \downarrow(e)$. Let us denote as $chk(e) = (e, \phi(e))$ the receipt event of the information sent to the checker by an event e.

From an operational viewpoint, as the network can reorder messages, information relative to events in the set $\downarrow(e') - \downarrow(e)$ could arrive at the checker after the receipt of timestamps associated with e and e' (see Figure 4.4). This actually creates a causality violation [11][2]. In this case, if the checker needs a timestamp which has not yet been received, it waits until its receipt.

This waiting actually creates a variable delay, namely *DeTection Delay* (DTD), defined as the time elapsed between $chk(e) = (e, \phi(e))$ and $chk(e') = (e', \phi(e'))$ have been received and the time in which the checker returns an answer. An example is depicted in Figure 4.4.

The detection delay is influenced only by the late arrival of some event at the checker that belongs either to the causal past of e'. We stress that although

[2]Two messages are causally ordered if the order of their deliveries at the destination process is consistent with the causality relation between their sending events.

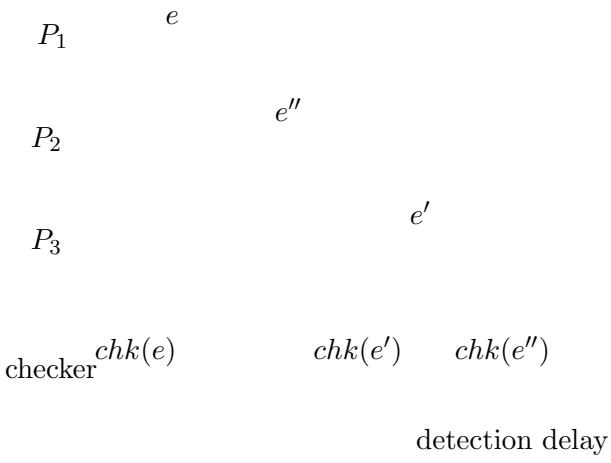


Figure 4.4: An example of Detection delay experienced at the checker detecting $e \prec e'$

$chk(e)$ and $chk(e')$ could be themselves received by the checker out of causal order, this cannot influence in any case the detection delay. This means that *if* DTD is greater than zero *then* messages are delivered at the checker out of causal ordering (the vice-versa is not necessarily true).

Remark 4.5.1. Let us remark that a protocol *characterizes causality on-the-fly* if (i) ϕ characterizes causality with delay (dC-P is satisfied) and (ii) the timestamps system $(\phi[\widehat{E}], <)$ is already transitively closed (i.e. $<^+ \equiv <$), or, equivalently, the timestamping domain $(T, <)$ is a partial order.

When the protocol characterizes causality on-the-fly, the checker can *"on-the-fly"* detect the causality dependency or the concurrency between events by only comparing their timestamps, i.e. $DTD = 0$, as the structure of causality is represented in a isomorphic way by the timestamps system $(\phi[\widehat{E}], <)$. On the contrary, let us remark that when the dC-P is satisfied, but the suborder $(\phi[\widehat{E}], <)$ is not transitively closed, ϕ characterizes causality, but *not on-the-fly*. Namely, the checker might experience a delay in detecting a causality relation.

4.6 k-Dependency Vectors

In this section we introduce k-dependency vectors, a timestamping protocol which characterizes causality with delay. We show the reconstruction algorithm and the detection algorithm executed by the checker to detect causal

dependencies and concurrencies between events. We point out a tradeoff between k (message overhead) and the DTD (detection delay).

k-Dependency Vectors protocol piggybacks on each application message from P_i to P_j $k \leq n$ selected components of the vector of size n local at process P_i including the i-th component of the vector (as in direct dependency protocol) and other $k-1$ components. The choice of the other $k-1$ values is left to the scheduling policy of the protocol.

This section also presents effective strategies that, for a given k, reduce the average detection delay at a checker.

We point out that by using a specific strategy, namely *Fixed-Set* strategy, where each process piggybacks the *same* $k-1$ entries on each message (the Fixed-Set), all causal relations $e \prec e'$, such that e has been produced by one of the $k-1$ processes in the Fixed-Set, are on-the-fly detectable by a checker (i.e., their DTD is zero as for vector clocks).

Moreover, we show that if a process selects the k entries of the local dependency vector corresponding to the indices of the sender processes of the k Most Recently Received messages by the process (MRR strategy), then even for a very small value of k we get small causality relation detection delays.

4.6.1 Basic idea

The idea behind k-Dependency Vectors timestamping protocol is the following: on generating a send event $e = send(m)$, a process P_i attaches to m a message timestamp which is an encoding of a subset of the causal past known to the sender process. This subset is a $\prec_l$-prefix-closed subset of events whose decomposition has at most k nonempty subsets.

Namely, a message timestamp is composed of k components selected as follows:

- an encoding of $\downarrow(e)_i$;
- $k-1$ ($k < n$) encodings of distinct sets $\overline{\downarrow}_{\ell_1}(e), \ldots, \overline{\downarrow}_{\ell_{k-1}}(e)$ (with $\ell_j \neq i$) selected according to some strategy (see Section 4.6.5), where each $\overline{\downarrow}_i(e)$ *is the largest prefix-closed subset of* $\downarrow_i(e)$ known to the sender process, i.e., $\overline{\downarrow}_i(e) \subseteq \downarrow_i(e)$.

For a send event $e \in E_i$, let $\overline{\downarrow}(e)|_m$ denote the set of events $(\uplus_{j=1}^{k-1} \overline{\downarrow}_{\ell_j}(e)) \uplus \downarrow(e)_i$ whose encoding is the message timestamp.

If $k < n$, each process P_i transfers on each message m an encoding of a subset of what it knows about the causal past of $send(m)$ (an example is depicted in Figure 4.5).

That is, each process knows only a part of the causal past of all the events e which it produces, denoted as $\overline{\downarrow}(e) = \uplus_{h=1,\ldots n} \overline{\downarrow}_h(e)$ and, when producing a

send event $e = send(m)$, it propagates on m the information about a $\prec_l$-prefix-closed subset of it, i.e., $\overline{\downarrow}(e)|_m$, whose decomposition has at most k nonempty subsets of events.

We note that it results $\downarrow_i (e) = \overline{\downarrow}_i(e) = \overline{\downarrow}_i(e)|_m$ for every send event $e \in E_i$, and, when $k = n$, $\overline{\downarrow}(e)|_m = \overline{\downarrow}(e) = \downarrow(e)$.

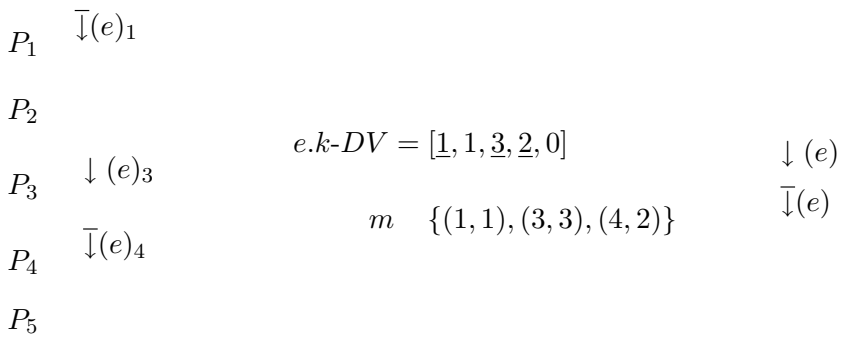


Figure 4.5: Example of 3-Dependency Vectors

4.6.2 The Protocol

From an operational point of view, each process P_i maintains a vector of n integers, denoted $k\text{-}DV_i$, which represents the timestamp $e.k\text{-}DV$ of the event e currently produced by P_i.

Each time a process P_i sends a message m to process P_j, it selects from its local dependency vector k values, corresponding to $k-1$ entries $\ell_1, \ldots \ell_{k-1}$ selected according to some policy, plus the i-th entry, and then it piggybacks this information, which is actually an encoding of $\overline{\downarrow}(e)|_m$, onto m.

The local vector $k\text{-}DV_i$ is updated according to the following rules:

[R0] $k\text{-}DV_i[1..n]$ is initialized to $[0, \ldots, 0]$.

[R1] Each time P_i produces an event e:

- it increments its local vector entry $k\text{-}DV_i[i]$;
- it associates with e the timestamp $e.k\text{-}DV = k\text{-}DV_i$.

[R2] When a message m is sent to P_i, a set $m.L$ of k pairs $(process\_index, k\text{-}DV_i[process\_index])$ is piggybacked on m. That is, $m.L$ is composed of:

- the pair $(i, k\text{-}DV_i[i])$;

– $k-1$ pairs $(\ell_j, k\text{-}DV_i[\ell_j])$, $j = 1, 2, \ldots, k-1$ *and* $\ell_j \neq i$, where $\ell_1, \ldots, \ell_{k-1}$ are $k-1$ distinct indices selected according to some policy.

[R3] When a message m is received by P_i:

$$\forall (l, k\text{-}DV[l]) \in m.L, \quad k\text{-}DV_i[\ell] := max(k\text{-}DV[l], k\text{-}DV_i[l]).$$

In the rest of the thesis, pairs $(j, k\text{-}DV_i[j])$ will be called *events identifiers*. In particular, we will say *direct dependency* the pair $(i, k\text{-}DV_i[i])$.

For example, Figure 4.5 shows a computation using 3-dependency vectors protocol. When process P_3 sends message m (event e), it carries on m the pair $(3, 3 - DV_3[3])$, i.e., the direct dependency, plus two selected pairs which correspond, in this case, to the ones relative to P_1 and to P_4 (the selected entries are underlined in the figure). That is, in this case $m.L = \{(1, 3 - DV_3[1]), (3, 3 - DV_3[3]), (4, 3 - DV_3[4])\}$.

The case $k = 1$: Direct Dependency

If $k = 1$, k-dependency boils down to the approach known as Direct Dependency protocol [18] and introduced in the context of distributed debugging. In such a timestamping protocol, when a process P_i sends a message, it piggybacks only the direct dependency, that is the pair $(i, k\text{-}DV_i[i])$.

In [10] an algorithm it is presented which reconstructs off-line vector clocks associated to events from their vector timestamps.

Remark 4.6.1. As k-dependency vectors protocol always piggybacks the direct dependency, they can reconstruct the vector clock associated to each event as it has been shown for the direct dependency protocol [10]. This corresponds to reconstruct the whole causal past of each event. We remark that piggybacking $k > 1$ events identifiers only impacts performance and not correctness.

The Formal Characterization

The k-Dependency Vector protocol is characterized by a timestamps domain $(T, <)$, where:

- $T = \{1, \ldots, n\} \times I\!N^n$;
- the relation $<$ is defined in following way:

$$(i, U) < (j, V) \;\Leftrightarrow\; U[i] \leq V[i]. \tag{4.6.1}$$

Later on, the aim is to prove that the system of k-dependency vectors satisfies the completeness property with delay (dC-P).

To reach this purpose it is useful to introduce the notions of *causal path* between two events and *length* of a causality relation between two given events.

Definition 4.6.1. Given two distinct events e and e' with $e \prec e'$, we say that a sequence of messages $< m_1, \ldots, m_h >$ is a causal path from e to e', denoted as $CP_e^{e'}$, if:

1. $e \prec_l send(m_1)$ or $e = send(m_1)$;
2. $\forall i = 1, \ldots, h-1, \quad receive(m_i) \prec_l send(m_{i+1})$;
3. $receive(m_h) \prec_l e'$ or $e' = receive(m_h)$.

For instance, in Figure 4.6 there is the causal path $CP_{e_1}^{e'} =< m_1, m_2, m_3 >$.

Definition 4.6.2. Given two distinct events e and e' with $e \prec e'$, we say that the *length* of the causality relation between e and e' is h, denoted by $e \prec^h e'$, if:

- $e \prec^- e'$, for $h = 1$, where $\prec^- = \prec_l \cup \prec_m$ is the transitive reduction of $\prec$;
- $\exists e'' \in E$ such that $e \prec^{h-1} e''$ and $e'' \prec^1 e'$, for $h > 1$.

The following theorem proves that the system of k-dependency vectors satisfies the completeness property with delay (dC-P). Property 4.6.2 gives an intuition.

Theorem 4.6.1. *A system of k-Dependency Vectors characterizes causality with delay, i.e., it satisfies dC-P.*

Proof. The formal proof is made of two parts. In fact, by dC-P, we have to prove that for every pair of events $e, e' \in E$, if they are produced by processes P_i and P_j, respectively, both following conditions are satisfied:

$$(i, e.k\text{-DV}) < (j, e'.k\text{-DV}) \Rightarrow e \prec e', \text{ and} \tag{4.6.2}$$

$$e \prec e' \Rightarrow (i, e.k\text{-DV}) <^+ (j, e'.k\text{-DV}). \tag{4.6.3}$$

If $(i, e.k\text{-DV}) < (j, e'.k\text{-DV})$, i.e., $e.k\text{-DV}[i] \leq e'.k\text{-DV}[i]$, then the $e.k\text{-DV}[i]$-th event of P_i, that is e itself, locally precedes or coincides with the $e'.k\text{-DV}[i]$-th event of P_i and then it must be $e \prec e'$. This proves condition (4.6.2).

The proof of (4.6.3) is by induction on the length h of $\prec$.

Base case. Let $h = 1$. We can distinguish the two following basic cases:

1. $e \prec_l e'$. Let $e, e' \in E_i$ for some index i. By construction it is $e.k\text{-}DV_i[i] < e'.k\text{-}DV_i[i]$ as P_i increments the i-th entry of the local vector each time an event is produced (see rule 1).

2. $e \prec_m e'$. Let $e \in E_i$ and $e' \in E_j$, $(i \neq j)$. The rule 2 implies that the direct dependency $(i, k\text{-}DV_i[i])$ is always piggybacked on message. So, in such a case $e.k\text{-}DV_i[i] = e'.k\text{-}DV_i[i]$.

Consequently, $e \prec^1 e' \Rightarrow (i, e.k\text{-DV}) < (j, e'.k\text{-DV})$.

Induction. Let $h \geq 2$. The induction assumption is: $\forall e, e' \in E$ such that $e \in E_i$ and $e' \in E_j$, it is $e \prec^{h-1} e' \Rightarrow (i, e.k\text{-DV}) <^+ (j, e'.k\text{-DV})$. We have to prove that condition (4.6.3) also holds if $e \prec^{h+1} e'$.

By definition, $e \prec^{h+1} e'$ only if $\exists e'' \in E_l$ (for some l) such that $e \prec^h e''$ and $e'' \prec^1 e'$. By induction assumption, it is $e \prec^h e'' \Rightarrow (i, e.k\text{-DV}) <^+ (l, e''.k\text{-DV})$ and by basic case one has $(l, e''.k\text{-DV}) < (j, e'.k\text{-DV})$. As a consequence, $(i, e.k\text{-DV}) <^+ (l, e'.k\text{-DV})$, that is the property also holds for length h. □

The fact that k-Dependency Vector protocol characterizes causality with delay implies that it is possible to transitively close the timestamps system to guarantee the correct detection of causality.

Threshold values for k

k-dependency vectors protocol piggybacks less information, in terms of number of bits, than the vector clocks one, if the number of pairs $(j, k\text{-}DV_i[j])$ is below a given threshold.

Let s be the number of bits required to encode a sequence number identifying an event on a process. To encode a message timestamp, k-dependency vectors require to represent a list $m.L$ of k pairs $(j, k\text{-}DV_i[j])$ selected from the local n-dimensional vector of integers.

A natural way is to use $s + \log_2 n$ bits for encoding each of k pairs in the message timestamp $m.L$. This require to transmit $k * (s + \log_2 n)$ bits.

Another simple way to encode $m.L$ is the following: for each entry of the local vector $k\text{-}DV_i$, if the entry $(j, k\text{-}DV_i[j])$ is in the message timestamp $m.L$ the s bits encoding value $k\text{-}DV_i[j]$ is prefixed with the boolean value "1", otherwise a value "0" is put and it is followed by the next entry. This require to transmit $n + k * s$ bits rather than $k * (s + \log_2 n)$.

We note that $n + k * s$ is better than $k * (s + \log_2 n)$ if $k > \frac{n}{\log_2 n}$.

Vector clocks protocol requires to transmit $n * s$ bits of control information.

It is easy to see that, for a message m, k-dependency vectors protocol is better that vector clocks one if:

- when $k \leq \frac{n}{\log_2 n}$, it is $k < \frac{n*s}{(s+\log_2 n)}$;
- when $k > \frac{n}{\log_2 n}$ it is $k < s - \frac{n}{s}$.

On-the-fly Detection

We recall that since k-dependency does not transfer the whole causal past of the sending event to the receiver, there can be some cases in which $e \prec e'$ and $e.DV[i] > e'.DV[i]$. This implies that, several events can be "on-the-fly" perceived as concurrent by a checker, when they are actually causally ordered.

The following property states the condition in which a causality relation is detected on-the-fly by a checker.

Property 4.6.2. *Let e and e' be two events of a distributed computation $\widehat{E}$ which adopts the k-dependency vector protocol to timestamp its events. A checker process is able to detect on-the-fly $e \prec e'$ if and only if a causal path $CP_e^{e'} =< m_1, \ldots, m_h >$ exists such that:*

$$e \in \bigcap_{i=1}^{h} \overline{\downarrow}(send(m_i))|_{m_i}.$$

For example, in Figure 4.6 the checker is able to detect on the fly the relation $e_1 \prec e'$, but it is not able to detect $e_2 \prec e'$. In fact, in the first case there is the causal path $CP_{e_1}^{e'} =< m_1, m_2, m_3 >$ such that $e_1 \in \bigcap_{i=1}^{3} \overline{\downarrow}(send(m_i))|_{m_i}$, while in the other one $e_2 \notin \overline{\downarrow}(e_3)|_{m_3}$.

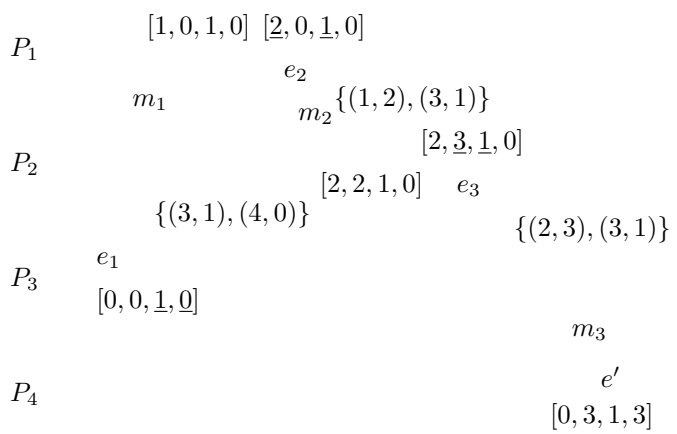


Figure 4.6: A distributed computation using 2-dependency vectors

Putting another way, Property 4.6.2 states that in order to detect on-the-fly a causality relation between e (produced by P_i) and e', there must be a causal path $CP_e^{e'} =< m_1, \ldots, m_h >$ *and* each message of the path has to piggyback the entry related to process P_i.
In fact, in the computation depicted in Figure 4.6 the relation $e_2 \prec e'$ is not on-the-fly detectable as m_3 does not piggyback the information related to P_1.

A checker needs to reconstruct, at least partially (see the next section), vector clocks of events e and e' and then compare them each time that (i) they are concurrent or (ii) they are causally dependent but do not satisfy property 4.6.2.

4.6.3 Reconstruction of Vector Clock from k-Dependency

If a checker is aware of all the k-DVs associated with the events of a computation, it can reconstruct all vector clocks VCs associated with events by back-tracking k-dependencies. This means that to compute the vector clock $e.VC$ of an event e, a checker needs to receive k-dependency vectors of events belonging to $\downarrow(e)$.

The Reconstruction Algorithm. From an operational point of view, the checker has n queues, one for each process, where it stores dependency vectors received by corresponding processes.

We consider that each time an event e is generated by a process P_i, a message is sent from P_i to the checker containing the identifier of the process i, the timestamp associated with e, i.e. $e.DV$, and a boolean flag $e.stable$ whose meaning will be explained in the following. This flag is set to $false$ by the process at the time of the sending.

Let us denote for simplicity's sake the $(e.k\text{-}DV[j])$-th event generated by process P_j as $e_{k\text{-}DV[j]}$.

procedure vector_clock(var e: event);
    repeat
        $old\_DV := e.k\text{-}DV$;
        for each $j \in \{1, \ldots, n\}$ **do**
            $e.k\text{-}DV = \mathsf{max}(e.k\text{-}DV, e_{k\text{-}DV[j]}.k\text{-}DV)$
        enddo
    until $(e.k\text{-}DV = old\_DV)$;
    $e.stable := true$;

The **repeat** statement actually computes the transitive closure of the causality relation. The inner loop moves forward the current k-dependency vector

timestamp of e by incorporating the new dependencies, stored in $e_{k\text{-}DV[j]}.k\text{-}DV$, revealed by events belonging to old_DV and unknown to $e.k\text{-}DV$. Note that if the timestamp of an event $e_{k\text{-}DV[j]}$ is not arrived yet at the checker, the reconstruction of the vector clock is blocked till its arrival. The sum of these delays in a run of the algorithm corresponds to the detection delay.

When $e.k\text{-}DV = old\_DV$ there are no more dependencies to be incorporated in $e.k\text{-}DV$, and then $e.k\text{-}DV$ contains the vector clock of e. The flag $e.stable$ is set to true to indicate that the reconstruction algorithm has terminated its run.

This algorithm is a modification of the algorithm proposed in [3] and [10] to reconstruct vector clocks from direct dependency vectors in the context of asynchronous distributed computations and distributed checkpointing respectively.

The Detection Algorithm. Concurrently with the reconstruction of vector clocks of events, a checker has to detect if two events e and e' are causally dependent. This is executed by the following function:

function causality_relation(e, e': event): boolean;
  wait until ($e.stable \wedge\ e'.stable$)
    $\vee$
    ($e.k\text{-}DV[e.producer] \leq e'.k\text{-}DV[e.producer]$)
  if $e.k\text{-}DV[e.producer] \leq e'.k\text{-}DV[e.producer]$
  then $causality\_relation := true$
  else $causality\_relation := false$

This function acts as a watchdog process waiting for one of the following two conditions:

- the checker has reconstructed both vector clocks, that is $e.stable \wedge e'.stable$, or
- the causal dependency has been detected, that is $e.k\text{-}DV[e.producer] \leq e'.k\text{-}DV[e.producer]$.

Note that in the latter case, the function can return *true* even though the vector clocks of e and e' have not been completely reconstructed yet.

The concurrency between two events can be detected by testing if the following predicate is false:

$$\text{causal\_relation}(e, e') \vee \text{causal\_relation}(e', e)$$

About the Detection Delay

As depicted in Figure 4.4, the detection delay of the causality relation (or the concurrency) between e and e' is the time elapsed between $chk(e)$ and $chk(e')$ have been executed and the time in which the function causality_relation(e, e') at the checker returns an answer (in some case this could happen before the reconstruction algorithm ends its run). If we consider the execution time of statements of the functions causality_relation and vector_clock as instantaneous, then the detection delay corresponds to the time the reconstruction algorithm is blocked to wait for timestamps associated with events belonging to $\downarrow (e')$. More specifically, the following assertion holds (whose proof is trivial).

Assertion 1. If there is a blocking of the reconstruction algorithm then there is a causality violation that involves (i) the last event delivered at the checker between e and e' and (ii) the delivery at the checker of an event e'' such that e'' belongs to $\downarrow (e') - \downarrow (e)$ [3].

As the detection delay is influenced by specific causality violations of message deliveries, the detection delay can be zero even in presence of causal ordering violations, if these do not involve the last event delivered at the checker between e and e'.

An example is shown in Figure 4.4. If we consider $k = 1$ (i.e., the direct dependencies protocol), the relation between e and e' is detected at the time the information related to the event e'' arrives at the checker (event $chk(e'')$) during that delay the reconstruction algorithm is blocked.

4.6.4 Trading k vs DTD

From the previous section, the tradeoff between k and DTD has emerged.

When k is close to n, the detection delay goes to zero as messages bring a lot of control information and then the probability that the reconstruction algorithm blocks to wait control information of some event is close to zero. In particular, when $k = n$, k-dependency vectors become vector clocks. As a consequence, $DTD = 0$.

When k is much less than n, the probability that the algorithm blocks becomes quite high. In particular, if k is equal to one (direct dependency), the vice-versa of assertion 1 is also true. As a consequence as k goes to 1, we get the largest average detection delay.

[3] As $DTD > 0$ only if some message arrives at the checker out of causal order, to ensure on-the-fly dependency tracking, one could use 1-dependency vectors and a subsystem that ensures causal deliveries. However, let us remark that the price to pay, in terms of message overhead, to ensure causally ordered deliveries in a point-to-point environment is $O(n^2)$ [35].

4.6.5 Selection Strategies

Property 4.6.2 defines a set of causality relations between pairs of events that can be detected by a checker just by comparing the k-dependency vectors associated with events. To detect the other causality relations a checker needs to rebuild, at least partially, the vector clocks related to the involved events. This may introduce a detection delay at the checker as remarked in the previous section.

Given a value of k, in this section we consider the following problem: *"how to select $k-1$ entries of a local dependency vector in order to minimize the detection delay at the checker?"*.

Simple selection heuristics could be: the *random* and the *static* one. The random strategy selects the $k-1$ entries randomly. In the static strategy each process selects the *same* $k-1$ entries each time it sends a message.

However, despite their simplicity, both these heuristics do not attack the problem of the minimization of the detection delay. To reach this purpose we introduce the following two strategies, namely, the *Fixed-Set* strategy and the *Most Recently Received* (MRR) strategy.

Fixed-Set Strategy

Let us consider a class of applications where one is interested in capturing only a subset of all causality relations. This subset contains all causality relations $e \prec e'$ such that e is an event produced by a process in the set $P_{\ell_1}, \ldots P_{\ell_{k-1}}$ with $1 \leq k < n$. These processes form the Fixed-Set.

The Fixed-Set strategy works as follows: each process P_i piggybacks on each message m its entry $k\text{-}DV_i[i]$ plus the entries related to processes in the Fixed-Set.

If $e \prec e'$ and e has been produced by a process in the Fixed-Set, then this relation is on-the-fly detectable by a checker as the entry of the process producing e, say P_{ℓ_i}, is piggybacked onto *all* messages of the computation. As a consequence, e satisfies Property 2. Hence for this subset of causality relations, k-dependency vectors are equivalent to transitive vector clocks.

We note that, by using this heuristic, a process does not need to piggyback pairs (process identifiers, entry of the local dependency vector) on each message as all processes agree on the members of the Fixed-Set. Then information on the process identifier can be omitted.

MRR Strategy

From the point of view of a process P_i at the time it is sending a message (event e), the best heuristic to minimize the detection delay is to make on-

the-fly detectable by a checker the causality relations that involve events that have more probability to create a causal ordering violation at the checker with e. If a message m has been received by P_i a lot of time before e, it will be extremely unlikely that the information related to the event $send(m)$ will cause a causal violation at the checker with the information related to e, i.e. $chk(e) \prec_l chr(send(m))$ (see Figure 4.7).

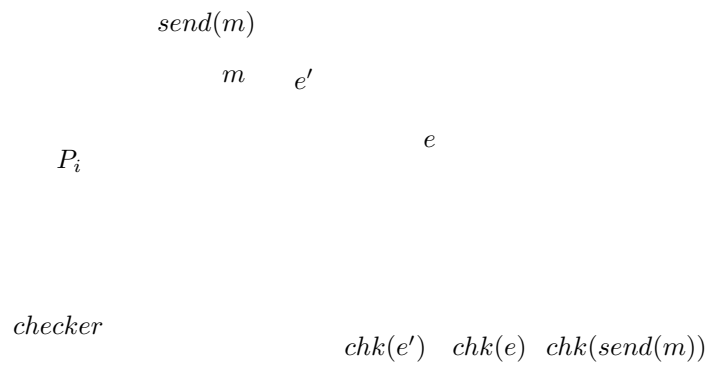


Figure 4.7: Example of a low probability causal ordering violation

Hence the events having a high probability of causal ordering violations at the checker are the ones connected to the last receipt of process P_i before producing e (such as event e' in Figure 4.7). An efficient heuristic consists, therefore, in selecting the $k-1$ entries of a dependency vector of a process P_i related to distinct senders of the $k-1$ Most Recently Received (MRR strategy) messages by P_i[4].

In this way the effect is twofold. On the one hand we make on-the-fly detectable by a checker the causality relations that involve events that could create with high probability a causal ordering violation with e (for example $e' \prec e$ in Figure 4.7). On the other hand, when considering non-on-the-fly detectable causality relations, it will be extremely likely that all the information related to dependency vectors necessary as an input to the reconstruction algorithm of the previous section will have already arrived at the time it runs. In other words, no blocking is introduced to wait for the arrival of some timestamps.

[4]If such processes are less than $k-1$, one option is to fill the missing pairs with processes having a non-zero entry in k-DV.

Simulation study

The simulation study carries out a performance comparison between k-dependency vectors, using MRR and Random, and direct dependency vectors (i.e. 1-Dependency vectors) in a specific application scenario. In particular, given a pair of events e and e', we measure the *DeTection Delay* (DTD) as a function of the number of processes of the computation.

We simulated a point-to-point environment in which each process can send messages to any other process and the destination of each message is an uniformly distributed random variable. Each process generates an internal, send or receive event with the same probability. Transmission delays of each directed point-to-point channel are distinct uniformly distributed random variables with mean value of 10 time units. Each simulation consists of one million of events and for each value of n in the set $\{5, 7, 10, 15, 20, 25, 35, 50, 70, 100\}$ we plot the ratio (R) between the mean DTD of k-dependency vectors (adopting MRR and the random strategies with three distinct value of k) and the mean DTD of direct dependencies. We did ten runs of each simulation with different seeds and the results were within 6% of each other. Thus variance is not reported in the plots. Results of the simulation study are plotted in Figure 4.8.

Figure 4.8: R *vs* Number of processes n.

When considering the plots of MRR strategy, they point out clearly the trade-off between k and the detection delay. As k increases the detection delay decreases. In particular, 2-dependency vectors *adopting MRR* show a reduction in DTD of 10 times with respect to direct dependencies almost indepen-

dent of the number of processes. This saturation behavior can be justified by pointing out that k-dependency vectors protocol using MRR, independently of the number of processes, always chooses to transfer information related to events with the highest probability to create a causality ordering violation at the checker. Moreover, an additional increment of k over a certain value produces a little reduction of R as k-dependency vectors protocol transmits information related to events that cause a causality ordering violation with small probability.

Let us compare k-dependency vectors adopting MRR and the Random strategy. Graphs of Figure 4.8 give an idea on (i) how a selection strategy impacts the detection delay and (ii) the performance distance between MRR and the random strategy. Let us consider, as an example, the point of MRR with k equal to 2 and the one of the Random strategy with k equal to 7 in a system with 10 processes. MRR performs better than Random even though it piggybacks only one 20% of the local vector clock of size n while Random piggybacks 70% this vector. This confirms the cruciality of piggybacking the information that could create causal ordering violations with high probability. Of course the performance distance can change when considering another application scenario. However, the better behaviour of MRR is a firm point.

4.7 Conclusions and Future Work

In this chapter we presented a global view of the main tradeoffs in order to answer to the problem of reducing the size of Vector Clocks [18, 37, 38].

Up to now the only known ways to get bounded dependency vectors were missing some concurrency between events, that is to accept some concurrent events appear as causally ordered, or having a local storage overhead.

This chapter pointed out another tradeoff: bounded dependency vectors can be traded for missing some on-the-fly detection of causality relations at a checker. We then presented a general scheme for dependency tracking, namely k-dependency vector protocol, which lies on that tradeoff. This scheme has Direct Dependency protocol and vector Clocks protocol as extreme cases when considering k equal to one and to n, respectively. Moreover, it provides scalability, with respect to control information piggybacked on application messages, as it attaches only $k \leq n$ entries of a vector of integer on each message. These entries are selected from a vector of n entries according to some strategy.

Simulations showed that when k-dependency vectors adopt the MRR strategy even for small (and fixed) value of k, the delay in detecting a causality relation by a checker is kept little.

Let us finally remark that for particular classes of distributed applications

where one is interested in detecting causality relations $e \prec e'$ such that e has been produced by a process in a set of $k-1$ processes, k-dependency vectors adopting the Fixed-Set strategy makes all such causality relations on-the-fly detectable by a checker.

A part of results of this chapter appears in [6, 24, 8].

	Timestamp	Message Timestamp	Local storage	Relation with $(E, \prec)$	On-the-fly detection of causality relations
VC	n	n	n	Isomorphic	all
SK-VC	n	$O(n)$ w.c.	$3n$	Isomorphic	all
PC	k	k	k	Extension	all but miss some concurrencies
k-DV	n	$O(k)$	n	Isomorphic	a subset

Table 1: Comparison between different aspects of dependency vector techniques.

Table 1 summarizes the feature of Vector Clocks (VC), efficient implementation of Singhal and Kshemkalyani (SK-VC), Plausible Clocks (PC) and k-dependency vectors (k-DV).

As regards future work, in order to complete the results contained in this chapter, an interesting issue is to obtain performance evaluation of the Fixed-Set heuristic by using the R metric to compare the speedup in detecting causality relations with respect to the Direct Dependency protocol.

Another question is to present simulation experiments to measure the magnitude of the detection delay compared to the instantaneous evaluation possible with Vector Clocks.

Chapter 5

Efficient Causality-Tracking between Relevant Events

This chapter considers the tracking of causal dependencies on a subset of events of asynchronous message-passing distributed computations, which are defined as "relevant" from the application point of view. We investigate the problem of reducing the size of the control information piggybacked by application messages when tracking causality on-the-fly between events in any set of relevant events. Moreover, this investigation is done in a context where communication channels are not required to be FIFO, and where there is no a priori information on the communication graph connectivity or the communication pattern.

The main issue concerns the *on-the-fly reduction of the number of entries of message timestamps.* Several protocols are proposed: each of them is based on a concrete approximation of an abstract condition. These protocols provide efficient implementations of vector clocks, in the sense that the size of message timestamps can be less than n.
The case where the channels are FIFO is first examined. In this context, we first extend the well-known timestamping protocol introduced by Singhal and Kshemkalyani [37] and presented in Section 4.2 to suit to the case where only a subset of the primitive events are relevant. Then, we consider the general case where channels are not necessarily FIFO and we propose two protocols, namely $\mathcal{P}1$ and $\mathcal{P}2$. From a practical point of view, these form a suite of protocols suited to be embedded in an *adaptative timestamping software layer* which can be used to reduce as much as possible the control information piggybacked on application messages. When a message m is sent by an application, the timestamping software layer on-the-fly selects the protocol of the suite that minimizes the size of the control information attached to m. It is finally shown that when FIFO channels are available, $\mathcal{P}1$ is strictly more efficient than the

extended Singhal-Kshemkalyani's protocol.

The chapter is composed of four sections. In Section 5.1 we define relevant events and re-examine the vector clocks protocol (Section 5.1.1) by considering the case where only relevant events are tracked. Section 5.2 introduces the abstract condition and proves that it is necessary and sufficient. Then, Section 5.3 considers the particular case where the channels are FIFO and presents the extended Singhal-Kshemkalyani's protocol E_SK. The two protocols $\mathcal{P}1$, $\mathcal{P}2$ and the comparison with E_SK are presented in Section 5.4.

5.1 Relevant Events

At some abstraction level only some events of a distributed computation are relevant [19, 20, 29]. So analyzing a distributed computation requires to define precisely a set of *relevant events*, also called *observable events*. The decision to consider an event as relevant can be up to the process that produces it, or triggered by some protocol. In this thesis, we are not concerned by this decision.

Let $O \subseteq E$ be the set of relevant events. At the considered abstraction level, the distributed computation is characterized by the order restriction $\widehat{O} = (O, \prec_o)$ of $(E, \prec)$, i.e.,

$$\forall\ (e, e') \in O \times O,\quad e \prec_o e' \ \Leftrightarrow\ e \prec e'. \tag{5.1.1}$$

In the following we consider a distributed computation at such an abstraction level.

Assumption. Let us assume that *only internal events can be observable*. We note that this is not a restriction, because if a communication event must be "observed", it is equivalent to create an internal observable event immediately following the communication event.

Notations. In this context we will adopt these notations:

- for every process P_i, O_i denotes the sequence of relevant events produced by P_i;
- $e_{i,x}$ indicates the *x-th relevant event* generated on process P_i;
- for every event e, $pred_l(e)$ denotes the event (not necessarily relevant) immediately preceding e on the same process (if it exists);

- if Var_i is a local variable of P_i and e is produced by P_i, then $e.Var_i$ denotes the value of the variable Var_i just after the occurrence of e and before the occurrence of the next event on P_i.

For example, if m is message sent by P_i to P_j, then $pred_l(receive(m)).VC_j[k]$ denotes the value of the k-th entry of VC_j just before the receipt of m by P_j, and $receive(m).VC_j[k]$ denotes this value just *after* the processing of $m.VC$ by P_j. We note that $send(m).VC_i[k]$ denotes the value of the k-th entry of VC_i just before or just after the event $send(m)$, as VC_i is not updated when a sending event occurs (by assumption a sending event cannot be a relevant event).

Without loss of generality, we do not represent non-observable internal events in time-space diagrams.

5.1.1 Vector Clocks

The vector clocks system is the only mechanism that associates timestamps with relevant events in such a way that the comparison of their timestamps indicates whether the corresponding (relevant) events are or are not causally related, and if they are, which one is the first.

In Section 3.2 we presented the protocol in the canonical implementation, by implicitly assuming that all the events are relevant. This implementation can be extended in a trivially way to track only relevant events.

More precisely, if VC_i is the local vector of process P_i, $VC_i[j]$ is the number of relevant events produced by process P_j that belong to the current causal past of P_i. The canonical protocol proposed in Section 3.2.1 can be easily extended by only adapting the rule [R2] in such a way that the i-th entry of the local vector VC_i of each process is incremented *only when P_i produces a relevant event*. Moreover, when a process P_i produces a relevant event e, it associates with it a vector timestamp whose value (denoted $e.VC$) is equal to the current value of VC_i.

We will refer to this extension as the *canonical implementation* of vector clocks protocol $\mathcal{P}0$.

Discussion

The major drawback of the canonical implementation $\mathcal{P}0$, as said in Section 3.2, lies in the fact that each message has to carry an array of n integers. It has been shown that, in the worst case, this is a necessary requirement [14, 5]. The efficient implementation proposed in [37] and presented in Section 4.2 requires FIFO channels and does not consider the notion of relevant event, i.e.,

it implicitly assumes that all the events are relevant. It has a local memory overhead, namely, a process has to manage two additional vectors of size n.

5.2 An Abstract Condition to Reduce the Size of Message Timestamps

This section defines an abstract condition that allows a process P_i not to transmit its whole vector clock VC_i each time it sends a message. Then, it is shown that this condition is both sufficient and necessary.

This condition expresses which part of control information can be omitted from a message timestamp without preventing a correct causality-tracking event timestamping. Nevertheless, it is abstract in the sense that it does not rely on particular data structures that could be directly accessed by processes.

To Transmit or not to Transmit Control Information

If we consider the rule [R3] of vector clocks protocol in Section 3.2.1, it is clear that a process P_j does not systematically update an entry $VC_j[k]$ each time it receives a message m from a process P_i. Namely, there is no update of $VC_j[k]$ when $VC_j[k] \geq m.VC[k]$ and, in such a case, the value $m.VC[k]$ is useless, and could be omitted from the control information transmitted with m from P_i to P_j.

This observation leads to the definition of the abstract condition $K(m, k)$ that allows a process P_i, sending a message m to P_j, to decide which entries of VC_i have to be transmitted with m.

Notation. Let $K(m, k)$ denote the following condition:

$$K(m, k) \equiv (send(m).VC_i[k] \leq pred_l(receive(m)).VC_j[k]).$$

If P_i was capable to evaluate $K(m, k)$ when it sends m to P_j, the management of vector clocks could be improved by modifying rules in the following way:

[R2] the pair $(k, VC_i[k])$ is transmitted with m if and only if $\neg K(m, k)$;

[R3] on receiving a message m, only the pairs carried by m are updated.

Let the pair $(k, VC_i[k])$ be the *identifier* of the $VC_i[k]$-th relevant event of P_k. When it is transmitted with m, this pair is denoted $(k, m.VC[k])$.

5.2.1 A Necessary and Sufficient Condition

We show that the condition $\neg K(m,k)$, used to decide whether or not the pair $(k, VC_i[k])$ has to be transmitted with m, is both necessary and sufficient.

Theorem 5.2.1. *The condition $\neg K(m,k)$ is both necessary and sufficient for P_i to transmit the pair $(k, VC_i[k])$ when it sends a message m to P_j.*

Proof. The proof is made of two parts: necessity and sufficiency.

Necessity. Let us consider a message m and let k be such that $K(m,k)$ is false, i.e.

$$send(m).VC_i[k] > prec_l(receive(m)).VC_j[k].$$

According to the definition of vector clocks (rule [R3]), we must have

$$receive(m).VC_j[k] = send(m).VC_i[k].$$

If the pair $(k, send(m).VC_i[k])$ is not attached to m, then P_i cannot update $VC_j[k]$ to its correct value (namely $send(m).VC_i[k]$).

Sufficiency. Let us consider a message m, and let k such that $K(m,k)$ is true, i.e.

$$send(m).VC_i[k] \leq pred_l(receive(m)).VC_j[k].$$

According to the definition of vector clocks (rule [R3]), we have

$$receive(m).VC_j[k] = pred_l(receive(m)).VC_j[k].$$

This means that the value $send(m).VC_i[k]$ is useless to P_j since the value $VC_j[k]$ is not updated upon the receipt of m. □

5.2.2 From the Abstract Condition to a Correct Approximation

Let us examine the condition $K(m,k)$. When P_i sends m to P_j, it knows the exact value of $send(m).VC_i[k]$ which is the current value of $VC_i[k]$. As far as the value $pred_l(receive(m)).VC_j[k]$ is concerned, there are two cases:

1. If $pred_l(receive(m)) \prec send(m)$, then P_i can know the value of $pred_l(receive(m)).VC_j[k]$ and, consequently, can evaluate $K(m,k)$.
2. If $pred_l(receive(m))$ and $send(m)$ are concurrent, then P_i cannot know the value of $pred_l(receive(m)).VC_j[k]$ and, consequently, cannot evaluate $K(m,k)$.

Moreover, when it sends m to P_j, whatever the case that occurs, P_i has no way to know which case *does occur*. In that sense, $K(m,k)$ is an abstract condition, because the "assumption" that P_i can evaluate $K(m,k)$ is not realistic in an asynchronous distributed computation.

The Idea

The idea is to use this abstract condition to define a family of vector clocks protocols. Each protocol of this family provides each process with particular data structures that allow it to evaluate a correct approximation of $K(m, k)$.

Notation. Let $K'(m, k)$ denote an approximation of $K(m, k)$, such that $K'(m, k)$ can be evaluated by P_i when it sends a message m.

The approximation $K'(m, k)$ is used in the following way:

- each time the condition $\neg K'(m, k)$ is satisfied, then the pair $(k, VC_i[k])$ must be attached to m.

To be correct, the condition $K'(m, k)$ must be such that, every time P_i has to transmit a pair $(k, VC_i[k])$ because $K(m, k)$ is false, then the approximation K' directs P_i to transmit this pair. In other words, for every message m and for every k, we must have $\neg K(m, k) \Rightarrow \neg K'(m, k)$, or equivalently, $K'(m, k) \Rightarrow K(m, k)$. This means that K' does not miss pairs whose transmission is required for the correct management of vector clocks.

As an example, let us consider the "constant" condition $K'(m, k) = false$. This trivially correct approximation of K actually corresponds to the canonical implementation $\mathcal{P}0$ presented in Section 5.1.1.

The two sections that follow present conditions that are better approximations of K. Section 5.3 first considers the case where channels are FIFO (condition K_{E_SK}). Then, Section 5.4 considers the general case where channels are not required to be FIFO (condition K_M).

5.3 The Case of FIFO Channels

5.3.1 Singhal-Kshemkalyani's Technique

As mentioned earlier, in the context of FIFO channels, an improvement of the canonical implementation of vector clocks was proposed by Singhal and Kshemkalyani [37]. This improvement, exposed in Section 4.2, does not consider the notion of relevant events; this means that it implicitly assumes that all the events are relevant.

Unfortunately, even under the FIFO assumption, this technique is not appropriate in the present model of computation, i.e., when only a subset of the primitive events are relevant (VC_i counting the number of relevant events per process).

A counterexample

The following counterexample, depicted in Figure 5.1 (the relevant events are denoted as dots), shows that, when using the canonical implementation exposed in Section 4.2, $VC_3[1]$ is not correctly updated when P_3 receives the message m_4.

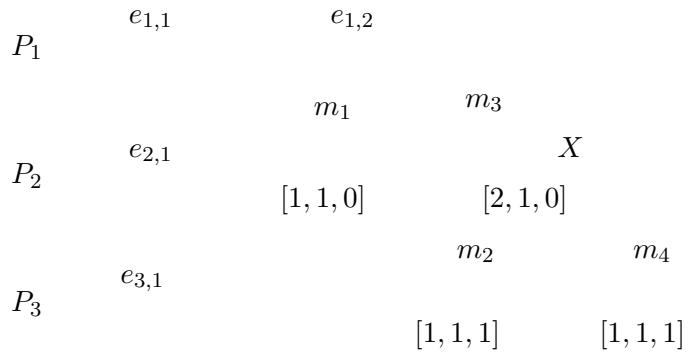


Figure 5.1: Singhal-Kshemkalyani's Technique is Not Appropriate when Communication Events are Not Relevant

The process P_2 updates its local vectors LU_2 and LS_2 in the following way:

- when P_2 receives m_1 from P_1, it updates LU_2 in this way: $LU_2[1] = 1$;
- when P_2 sends m_2 to P_3, it executes the following update: $LS_2[3] = 1$;
- when P_2 receives the message m_3 from P_1, the entry $VC_2[1]$ is incremented but as at the process P_2 no relevant event has been observed, no update of $LU_2[1]$ is occurred;
- when P_2 sends m_4 to P_3 (communication event noted X), since $LU_2[1] = 1$ and $LS_2[3] = 1$, the pair $(1, VC_1[1]) = (1, 2)$ is not transmitted with m_4.

As a consequence, on the receipt of m_4, no update of $VC_3[1]$ occurs at process P_3 and the causality relation $e_{1,2} \prec receive(m_4)$ is not taken into account.

This counterexample shows from an operational point of view that it is not sufficient for each process P_i to manage the set of extra local variables LU_i and LS_i to preserve all the causality relations between relevant events.

5.3.2 An Extension of Singhal-Kshemkalyani's Technique

This section proposes a concrete condition K_{E_SK}, based on an extension of Singhal-Kshemkalyani's technique.

We note that whereas the basic protocol described in [37] is presented without proof, in this section it is then shown that this condition is correct. Namely, in every computation, for every pair of processes P_i and P_j, for every message m sent by P_i to P_j and for every k,

$$K_{E\_SK}(m,k) \Rightarrow K(m,k).$$

Data structures

From an operational point of view, each process P_i manages the following set of local variables:

- an integer value X_i counting non-relevant events between two consecutive relevant events on process P_i.
- a vector $LU_i[1..n]$ whose entries belong to $\mathbb{N} \times \mathbb{N}$, called "Last Update", such that $LU_i[k] = (x, \alpha)$ means that $VC_i[i] = x$ and $X_i = \alpha$ when process P_i last updated the entry $VC_i[k]$.

  Note that, if $k \neq i$, $VC_i[k]$ can be updated only when a receive event (i.e., a non-relevant event) occurs, and thus $LU_i[k] = (x, \alpha)$ with $x \leq VC_i[k]$ and $\alpha > 0$. Moreover, $LU_i[i] = (VC_i[i], 0)$.

  Let us denote by $LU_i[k].f$ (resp. $LU_i[k].s$) the first (resp. second) component of the pair $LU_i[k]$.
- a vector $LS_i[1..n]$ whose entries belong to $\mathbb{N}\times\mathbb{N}$, called "Last Sent", such that $LS_i[k] = (x, \alpha)$ means that $VC_i[i] = x$ and $X_i = \alpha$ when process P_i last sent a message to P_k.

  Note that $\alpha > 0$ since a sent event is not a relevant event. Let us denote by $LS_i[k].f$ (resp. $LS_i[k].s$) the first (resp. second) component of the pair $LS_i[k]$.

Notations. Later on, we will use the following notations:

- $LS_i[j] \geq LU_i[k]$ will denote the property:

  $$(LS_i[j].f > LU_i[k].f) \vee ((LS_i[j].f = LU_i[k].f) \wedge (LS_i[j].s \geq LU_i[k].s)).$$

  By an obvious application of boolean calculus, we have $LS_i[j] < LU_i[k]$ denotes:

  $$(LS_i[j].f < LU_i[k].f) \vee ((LS_i[j].f = LU_i[k].f) \wedge (LS_i[j].s < LU_i[k].s)).$$

- The concrete condition K_{E_SK} is defined as follows: let m be a message sent by P_i to P_j. Then,

$$K_{E\_SK}(m,k) \equiv LS_i[j] \geq LU_i[k].$$

Clearly, we have:

$$\neg K_{E\_SK}(m,k) \equiv (LS_i[j] < LU_i[k]).$$

E_SK Protocol

For a process P_i, the extended protocol, denoted E_SK, is defined by the following set of rules:

[E-SK0] (Initialization):

- $VC_i[1..n]$ is initialized to $[0,\ldots,0]$;
- $LU_i[1..n]$ is initialized to $[(0,0),\ldots,(0,0)]$;
- $LS_i[1..n]$ is initialized to $[(0,0),\ldots,(0,0)]$.

[E-SK1.a] Each time P_i produces a non-relevant event (internal/send/receive), it increments its local counter X_i.

[E-SK1.b] Each time P_i produces a relevant event e:

- it increments its vector clock entry $VC_i[i]$;
- it associates with e the timestamp $e.VC = VC_i$;
- it resets the internal counter: $X_i := 0$.

[E-SK2] When P_i sends a message m to P_j:

- it attaches to m the set $m.VC$ of event identifiers $(k, VC_i[k])$ such that $\neg K_{E\_SK}(m,k)$;
- it updates its local vector LS_i by executing: $LS_i[j] := (VC_i[i], X_i)$.

[E-SK3] When P_i receives a message m from P_j, it executes the following updates. $\forall (k, VC[k]) \in m.VC$:

- $VC_i[k] := max(VC_i[k], VC[k])$;
- if $VC_i[k]$ is updated, then $LU_i[k] := (VC_i[i], X_i)$.

Interestingly, when all the primitive events are relevant, this protocol reduces to the basic protocol presented in [37].

5.3.3 Correctness of the Condition K_{E_SK}

Theorem 5.3.1. *For every pair of indices i, j, for every message m sent from P_i to P_j and $\forall k$:*

$$K_{E\_SK}(m,k) \Rightarrow K(m,k).$$

Proof. Let us consider a computation and two processes P_i and P_j. Because of the FIFO assumption, the set of messages sent from P_i to P_j is a sequence $\mathcal{M}_{ij} = [m_1, m_2, \ldots, m_q, \ldots]$. The proof is by induction on this sequence.

Base Case. Let m_1 be the first element of $\mathcal{M}_{ij}$. We have that $send(m_1).LS_i[j] = (0,0)$. If $K_{E\_SK}(m_1,k)$ holds, necessarily $send(m_1).LU_i[k] = (0,0)$. In particular, $send(m_1).VC_i[k] = 0$. Thus,

$$0 = send(m_1).VC_i[k] \leq pred_l(receive(m_1)).VC_j[k]$$

i.e., $K(m_1,k)$ holds.

Induction. Let m_q be any message of the sequence, with $q > 1$. The induction assumption is $\forall r < q,\ \forall k\ :\ K_{E\_SK}(m_r,k) \Rightarrow K(m_r,k)$. We have to show that $\forall k\ :\ K_{E\_SK}(m_q,k) \Rightarrow K(m_q,k)$.

Let us define:

- $(x,\alpha) = (pred_l(send(m_q)).LS_i[j]$
- $(y,\beta) = pred_l(send(m_q)).LU_i[k]$.

If $K_{E\_SK}(m_q,k)$ holds, we have either $(x > y)$ (Figure 5.2.1) or $(x = y) \wedge (\alpha \geq \beta)$ (Figure 5.2.2). Note that in the second case, $\alpha > \beta$ because a receive event and a send event are distinct.

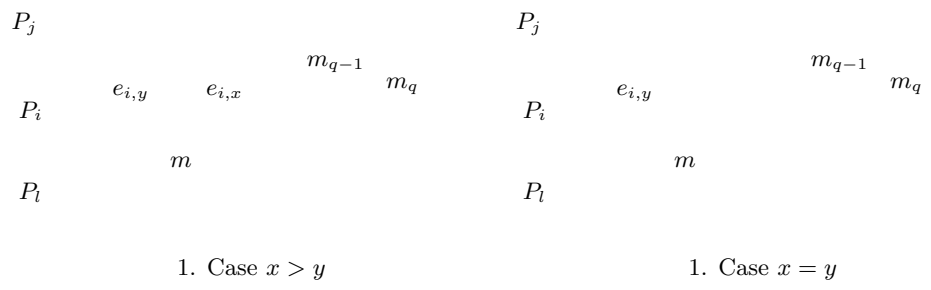


Figure 5.2: Proof of Theorem 5.3.1

Let m denote the first message received by P_i from some P_ℓ, whose receipt has updated $VC_i[k]$ to its current value $send(m_q).VC_i[k]$.

By construction, $receive(m)$ occurred on P_i after the relevant event $e_{i,y}$ and before the relevant event $e_{i,y+1}$. Similarly, $send(m_{q-1})$ occurred on P_i after the relevant event $e_{i,x}$ and before the relevant event $e_{i,x+1}$.

- If $x > y$ or if $(x = y) \wedge (\alpha > \beta)$, then the event $receive(m)$ occurred on P_i before the event $send(m_{q-1})$. Thus,

$$receive(m).VC_i[k] \leq send(m_{q-1}).VC_i[k] \leq send(m_q).VC_i[k].$$

  Since $VC_i[k]$ has not been updated since the event $receive(m)$, we have

$$receive(m).VC_i[k] = send(m_{q-1}).VC_i[k] = send(m_q).VC_i[k].$$

  If $K_{E\_SK}(m_{q-1}, k)$ holds, then by the induction assumption, we have $pred_l(receive(m_{q-1})).VC_j[k] = receive(m_{q-1}).VC_j[k]$ and

$$send(m_{q-1}).VC_i[k] \leq receive(m_{q-1}).VC_j[k].$$

- If, on the contrary, $K_{E\_SK}(m_{q-1}, k)$ does not hold, then the pair $(k, send(m_{q-1}).VC_i[k])$ has been attached to m_{q-1} and thus

$$send(m_{q-1}).VC_i[k] \leq receive(m_{q-1}).VC_j[k].$$

Thus, in both cases, $send(m_q).VC_i[k] = send(m_{q-1}).VC_i[k] \leq receive(m_{q-1}).VC_j[k] \leq receive(m_q).VC_j[k]$, i.e., $K(m_q, k)$ holds. □

5.4 The General Case

This section provides another correct implementation K_M of the abstract condition, from which two protocols are designed.

To our knowledge, the proposed protocols are the first that implement the abstract condition, on-the-fly and without additional control messages, in a context where channels are not necessarily FIFO, and there is no a priori information on the communication graph connectivity or the communication pattern.

Moreover, if the channels are FIFO, the protocols can still be improved and this improvement is more efficient than the extended Singhal-Kshemkalyani's protocol presented in the previous section.

5.4.1 Management of a Boolean Matrix

In order to implement the approximation K_M of the condition K, each process P_i is equipped with a very simple data structure, namely a boolean matrix M_i.

This boolean matrix is managed to satisfy the following property:

Property 5.4.1. *For each message m sent by P_i to P_j:*

$$(send(m).M_i[j,k] = 1) \Rightarrow (send(m).VC_i[k] \leq pred_l(receive(m)).VC_j[k]).$$

Let $K_M(m,k) \equiv (send(m).M_i[j,k] = 1)$.
We will show that $K_M(m,k) \Rightarrow K(m,k)$. This means that $K_M(m,k)$ is a condition that is sufficient to determine whether the pair $(k, send(m).VC_i[k])$ has to be attached to the message m.

In order to attain this goal, the matrix M_i is managed according to the following rules:

[M0] Initialization:

- $\forall\ (j,k):\ M_i[j,k]$ is initialized to 1.

[M1] Each time it produces a relevant event e:

- P_i resets the ith column of its boolean matrix:
  $\forall j \neq i:\ M_i[j,i] := 0$.

[M2] When P_i sends a message: no update of M_i occurs.

[M3] When it receives a message m from P_j, P_i executes the following updates:

$$\begin{array}{l}\forall\ (k, m.VC[k],) \in\ m.VC: \\ \qquad \textbf{case}\ VC_i[k] < m.VC[k]\ \textbf{then}\ \forall \ell \neq i,j,k: \\ \qquad\qquad\qquad\qquad\qquad M_i[\ell,k] := 0;\ M_i[j,k] := 1 \\ \qquad\qquad VC_i[k] = m.VC[k]\ \textbf{then}\ \ M_i[j,k] := 1 \\ \qquad\qquad VC_i[k] > m.VC[k]\ \textbf{then}\ \ skip \\ \qquad \textbf{endcase}\end{array}$$

Remark 5.4.1. As regards the rule [M1], actually, the value of the i-th column of the boolean matrix M_i remains constant after its first update. In fact, $\forall j,\ M_i[j,i]$ can be set to 1 only upon the receipt of a message from P_j, carrying the value $VC_j[i]$ (see [M3]). But, as $M_j[i,i] = 1$, P_j does not send $VC_j[i]$ to P_i. So, it is possible to improve the protocol by executing this "reset" of the column $M_i[*,i]$ only when P_i produces its first relevant event.

5.4.2 Correctness

This section first shows that the previous implementation of the matrix M_i, i.e., rules [M1]-[M3], combined with the vector clock rules [R0]-[R3], provides its correct meaning to M_i, namely (Lemma 5.4.2):

$$(send(m).M_i[j,k] = 1) \Rightarrow (send(m).VC_i[k] \leq pred_l(receive(m)).VC_j[k]).$$

It is then shown that M_i can be used to implement the abstract condition $K(m,k)$ (Theorem 5.4.3).

Lemma 5.4.2. *Let $\mathcal{IM}(e,j,k)$ denote the following property, where $e \in E_i$:*

$$\begin{aligned}[(e.M_i[j,k] = 1) \Rightarrow\ & (j = k)\ \vee (i = j)\ \vee (e.VC_i[k] = 0)\ \vee (\exists m' \text{ from } P_j \text{ to } P_i : \\ & ((receive(m') \prec e) \vee (receive(m') = e))\ \wedge \\ & (send(m').VC_j[k] = m'.VC[k] = e.VC_i[k]))].\end{aligned}$$

$\forall i,\ \forall e \in E_i,\ \forall j,\ \forall k :\ \mathcal{IM}(e,j,k)$ holds.

Proof. The proof is by induction on the poset $\widehat{E}$. Since $\mathcal{IM}(e,j,k)$ trivially holds for every event e such that $e.M_i[j,k] = 0$, we consider only the events e such that $e.M_i[j,k] = 1$.

Moreover, if for an event $e \in E_i$ and a pair (j,k) we have $(pred_l(e).M_i[j,k] = e.M_i[j,k]) \wedge pred_l(e).VC_i[k] = e.VC_i[k]$ then, obviously, $\mathcal{IM}(pred_l(e),j,k) \Rightarrow \mathcal{IM}(e,j,k)$ (referred in the rest of the proof as a "no-change situation").

Base Case. Let e be the first event of P_i. We have $e.M_i[j,k] = 1$ only in the following cases:

- e is neither a relevant nor a receive event. Then we have $e.VC_i[k] = 0$.
- e is a relevant event. So, $\forall j,\ \forall k \neq i,\ e.M_i[j,k] = 1$ and $e.VC_i[k] = 0$.
- e is a receipt of a message m from P_j. From [M3], [R0] and [R3] we have, for every $x \neq i$, $e.M_i[x,y] = 1 \Rightarrow ((x = j) \wedge (y = k)) \vee ((x \neq j) \wedge (y \neq k) \wedge (m.VC[k] = 0))$.

  If the first alternative holds, m satisfies $receive(m) = e \wedge send(m).VC_j[k] = m.VC[k] = e.VC_i[k]$ and thus the right part of $\mathcal{IM}(e,j,k)$ holds.

  If the second alternative holds, $e.VC_i[k] = 0$ and thus the right part of $\mathcal{IM}(e,j,k)$ holds.

  Then, we have, from [M3], $\forall j,\ \forall \ell \neq i :\ e.M_i[j,\ell] = 1$ and $e.VC_i[\ell] = 0$.

So, in every case, $\mathcal{IM}(e, j, k)$ holds.

Induction. Let $e \in E_i$. The induction assumption is: $\forall e' \in \{e' \mid e' \stackrel{hb}{\rightarrow} e\}$, $\forall j$, $\forall k$, the property $\mathcal{IM}(e', j, k)$ holds. We have to prove that, $\forall j$, $\forall k$, the predicate $\mathcal{IM}(e, j, k)$ holds. We proceed by case analysis.

- e is a relevant event. Consider the column $M[*, i]$. Due to [M1], $\forall j \neq i :\ e.M_i[j, i] = 0$. Consider now columns $M[*, k]$, with $k \neq i$. From [R1], we have a no-change situation. So, $\mathcal{IM}(e, j, k)$ holds.

- e is a send event. From [M2] and [R2], this is a no-change situation, so $\mathcal{IM}(e, j, k)$ holds.

- e is the receipt of m from P_j. Due to [M3], the only entries such that $e.M_i[x, y] = 1$ are the following:

  1. $x = i$.
  2. $\forall x$: all the entries (x, x) of M_i.
  3. All the entries (x, k) of M_i where k is such that $(pred_l(e).M_i[x, k] = 1) \wedge (k, m.VC[k]) \notin m$.
  4. All the entries (j, k) of M_i where k is such that $pred_l(e).VC_i[k] \leq m.VC[k]$.
  5. All the entries (x, k) of M_i such that $(pred_l(e).VC_i[k] \geq m.VC[k]) \wedge (pred_l(e).M_i[x, k] = 1)$.

  Case 1 cannot occur, since a process does not send a message to itself.

  Case 2 is trivial ($\forall x : \mathcal{IM}(e, x, x)$ trivially holds).

  Case 3 is a no-change situation.

  Case 4: from [R3] we have $e.VC_i[k] = m.VC[k] = send(m).VC_j[k]$ and thus the message m satisfies the right part of the implication involved in $\mathcal{IM}(e, j, k)$.

  Case 5 is a no-change situation.

  Hence, in every case, $\mathcal{IM}(e, j, k)$ holds.

□

The following theorem shows that the condition $K_M(m, k)$ is correct.

Theorem 5.4.3. *$\forall i$, $\forall j$, $\forall m$ sent by P_i to P_j, $\forall k$: $K_M(m, k) \Rightarrow K(m, k)$.*

Proof. Consider a computation and two processes P_i and P_j. Let m be a message sent by P_i to P_j and let $e = send(m)$.

Let us assume that $send(m).M_i[j,k] = 1$. From Lemma 5.4.2, this implies either one of the four cases (note that $i \neq j$):

1. $j = k$. m is a message from P_i to P_j. Properties of vector clocks imply that $send(m).VC_i[j] \leq pred_l(receive(m)).VC_j[j]$.
2. $send(m).VC_i[k] = 0$. In that case, $send(m).VC_i[k] = 0 \leq pred_l(receive(m)).VC_j[k]$.
3. $\exists m'$ from P_j to P_i s.t. $receive(m') \prec e \wedge (send(m').VC_j[k] = m'.VC[k] = send(m).VC_i[k])$. This implies $send(m') \prec pred_l(receive(m))$ (locally on P_j) and thus $send(m').VC_j[k] \leq pred_l(receive(m)).VC_j[k]$. Hence, $send(m).VC_i[k] \leq pred_l(receive(m)).VC_j[k]$.
4. $\exists m'$ from P_j to P_i s.t. $receive(m') = e \wedge \ldots$. Obviously, this case cannot occur, because e is not a receive event.

It follows that $(send(m).M_i[j,k] = 1) \Rightarrow (pred_l(receive(m)).VC_j[k] \geq send(m).VC_i[k])$ in each case. Hence, $K_M(m,k) \Rightarrow K(m,k)$. □

We present two protocols based on K_M:

1. The protocol $\mathcal{P}1$ directly uses M_i to reduce the number of vector clock entries that have to be transmitted.
2. The protocol $\mathcal{P}2$ shows that this number can still be reduced if we allow a message to carry a few boolean vectors. It follows that the protocol $\mathcal{P}2$ exhibits a tradeoff in the control information piggybacked by messages (integers *vs* boolean arrays).

5.4.3 Protocol $\mathcal{P}1$

For a process P_i, the protocol implementing vector clocks with matrices M is defined by the following set of rules:

[MR0] Initialization:

- $VC_i[1..n]$ is initialized to $[0, \ldots, 0]$;
- $\forall\ (j,k):\ M_i[j,k]$ is initialized to 1.

[MR1] Each time it produces a relevant event e:

- P_i increments its vector clock entry $VC_i[i]$ $(VC_i[i] := VC_i[i] + 1)$;

- P_i associates with e the timestamp $e.VC = VC_i$;
- P_i resets the ith column of its boolean matrix: $\forall j \neq i : M_i[j,i] := 0$.

[MR2] When it sends a message m to P_j, P_i attaches to m the following set of event identifiers: $m.VC = \{(k, VC_i[k]) \mid M_i[j,k] = 0\}$.

[MR3] When it receives a message m from P_j, P_i executes the following updates:

$\forall (k, VC_j[k]) \in m.VC$ **do**:
case $VC_i[k] < m.VC[k]$ **then** $VC_i[k] := m.VC[k]$;
$\forall \ell \neq i, j, k : M_i[\ell, k] := 0$;
$M_i[j,k] := 1$;
$VC_i[k] = m.VC[k]$ **then** $M_i[j,k] := 1$;
$VC_i[k] > m.VC[k]$ **then** *skip*
endcase

Let $M_i[*, k]$ denote the k-th column of M_i.

We remark that actually, the value of the column $M_i[*, i]$ remains constant to zero after its first update (see rule [MR1]). In fact, $\forall j$, $M_i[j,i]$ can be set to 1 only upon the receipt of a message from P_j, including $(j, VC_j[i])$. But, as $M_j[i,i] = 1$, P_j does not send $VC_j[i]$ to P_i.

So, it is possible to improve the protocol $\mathcal{P}1$ and the protocol $\mathcal{P}2$ in Section 6.3.3, by executing this "reset" of the column $M_i[*, i]$ only when P_i produces its first relevant event. This means that at the next sending to P_j, process P_i will transmit again the pair $(i, VC_i[i])$ to P_j, event if this value is already known to P_j.

5.4.4 Protocol $\mathcal{P}2$

The protocol $\mathcal{P}2$ aims to increase the number of entries of M_i that are set to 1, and consequently decrease the number of pairs $(k, VC_i[k])$ that a message has to piggyback. This is obtained without adding new control information, but requires messages to piggyback boolean arrays.

This shows that, for each message, there is a tradeoff between the number of pairs $(k, VC_i[k])$ that are saved and the number of boolean vectors that have to be piggybacked.

The rules [MR0] and [MR1] are the same as before. The rules [MR2] and [MR3] are modified in the following way.

[MR2'] When it sends a message m to P_j, P_i attaches to it the following set ($m.VC$) of triples (each made up of a process id, an integer and a n-boolean array): $\{(k, VC_i[k], M_i[*,k]) \mid M_i[j,k] = 0\}$.

[MR3'] When it receives a message m from P_j, P_i executes the following updates:

$$
\begin{array}{l}
\forall (k, VC_j[k], M_j[k][1..n]) \in m.VC: \\
\quad \textbf{case } VC_i[k] < m.VC[k] \textbf{ then } \; VC_i[k] := m.VC[k]; \\
\qquad\qquad\qquad\qquad\qquad\quad \forall \ell \neq i : M_i[\ell, k] := m.M[k, \ell] \\
\qquad\quad VC_i[k] = m.VC[k] \textbf{ then } \; \forall\ \ell \neq i :\ M_i[\ell, k] := max(M_i[\ell, k], m.M[k, \ell]) \\
\qquad\quad VC_i[k] > m.VC[k] \textbf{ then } \; skip \\
\quad \textbf{endcase}
\end{array}
$$

This shows that, in the first case, values $M_i[\ell, k]$ $(k,\ \ell \neq i)$ are now updated with actual values of the sender's matrix, instead of systematically being reset to 0.

Similarly, in the second case, more values are updated (on the basis of a more recent information) than in protocol $\mathcal{P}1$. The proof of $\mathcal{P}2$ is similar to the previous one. In the rest of the chapter, $T_{1\rightarrow 2}$ denotes the modification that transforms protocol $\mathcal{P}1$ into protocol $\mathcal{P}2$.

5.4.5 An Adaptive Timestamping Layer

The protocol $\mathcal{P}1$ (resp. $\mathcal{P}2$) piggybacks less information, in terms of number of bits, than the protocol $\mathcal{P}0$, if the number of pairs $(k, VC_i[k])$ (resp. triples $(k, VC_i[k], M_i[*,k])$) is below a given threshold.

Let s be the number of bits in a sequence number (identifying a relevant event on a process), and $|m.X|$ denote the number of elements of the set $m.X$. If $m.VC$ and $m.VC'$ are the timestamp associated with m by protocols $\mathcal{P}1$ and $\mathcal{P}2$, respectively, then one has:

- the protocol $\mathcal{P}0$ requires to transmit $n.s$ bits of control information;
- the protocol $\mathcal{P}1$ requires $|m.VC| \cdot (s + \log_2 n)$ bits;
- $\mathcal{P}2$ requires $|m.VC'| \cdot (n + s + \log_2 n)$ bits

It can be easily seen that, for a message m:

1. $\mathcal{P}1$ is better than $\mathcal{P}0$ if

$$|m.VC| < \frac{n \cdot s}{(s + \log_2 n)}.$$

2. protocol $\mathcal{P}2$ is better than $\mathcal{P}0$ and $\mathcal{P}1$ if

$$|m.VC'| < \frac{\min(n \cdot s\ ,\ |m.VC| \cdot (s + \log_2 n))}{(n + s + \log_2 n)}.$$

From an operational point of view, the previous thresholds can be used by a *timestamp software layer*, which embeds $\mathcal{P}0$, $\mathcal{P}1$ and $\mathcal{P}2$ to minimize the information piggybacked by application messages.

This layer is actually composed by a high-level protocol $\mathcal{P}$ that, when a sending event is generated by an application process, evaluates which is the best timestamping protocol to use and formats the piggybacked information according to the selected protocol.

More precisely, $\mathcal{P}$ adds, at sender side, a two bits header to the piggybacked information to indicate which timestamping is used (e.g., 00 for $\mathcal{P}0$, 01 for $\mathcal{P}1$ and 10 for $\mathcal{P}2$). This header is used by $\mathcal{P}$, at the receiver side, in order to execute the receipt rule corresponding to the timestamping protocol executed by the sender.

Concerning the implementation of $\mathcal{P}$, the rules are the following ones:

[MR0] Initialization:

- $VC_i[1..n]$ is initialized to $[0, \ldots, 0]$,
- $\forall\ (j,k):\ M_i[j,k]$ is initialized to 1.

[MR1] Each time it produces a relevant event e:

- P_i increments its vector clock entry $VC_i[i]$ ($VC_i[i] := VC_i[i] + 1$),
- P_i associates with e the timestamp $e.VC = VC_i$,
- P_i resets the ith column of its boolean matrix: $\forall j \neq i:\ M_i[j,i] := 0$.

[MR2"] Each time P_i sends a message m to P_j,

let $m.VC = \{((k, VC_i[k]) \mid M_i[j,k] = 0\}$;
let $m.VC' = \{(k, VC_i[k], M_i[*,k]) \mid M_i[j,k] = 0\}$;
if $|m.VC'| \cdot (n + s + \log_2 n) < min(n \cdot s, |m.VC| \cdot (s + \log_2 n))$
  then attach to m the header "10" and executes rule [MR2']
  else **if** $|m.VC| \cdot (s + \log_2 n) < (n \cdot s)$
    then attach to m the header "01" and executes rule [MR2]
    else attach to m the header "00" and the whole local vector clock
  endif
endif

[MR3"] Each time P_i receives a message from P_j, it applies the rule of the protocol specified by the header.

5.4.6 Coming Back to FIFO Channels

Improving $\mathcal{P}1$ and $\mathcal{P}2$

Let us assume that channels are FIFO. This additional assumption can be used to improve the protocols behavior. More precisely, when a process P_i sends a message m to P_j by attaching to it a pair $(k, VC_i[k])$ or a triple $(k, VC_i[k], M_i[*, k])$, in addition to the statements defined in the rules [MR2] or [MR2'], it can immediately execute the update $M_i[j, k] := 1$.

Consequently, it appears that the FIFO property of channels can reduce the number of pairs piggybacked by a message.

Let us consider that $M_i[j, k] = 1$ when P_i sent a message to P_j after last update of the value $VC_i[k]$: $VC_i[k]$ has not to be re-transmitted to P_j before next update. It follows that process P_i sends the pair $(i, VC_i[i])$ to a process P_j only if, since its last relevant event, this sending is the first sending to P_j.

Comparison with E_SK

This section shows that, in presence of FIFO channels, the protocol $\mathcal{P}1$ is strictly more efficient than the E_SK protocol (denoted $\mathcal{P}1 \prec E\_SK$), where the notion of efficiency is referred to the number of pairs in the list $m.VC$ attached to messages.

Given two protocols $\mathcal{P}$ and $\mathcal{Q}$, based respectively on conditions $K_{\mathcal{P}}$ and $K_{\mathcal{Q}}$ (satisfying $K_{\mathcal{P}} \Rightarrow K$ and $K_{\mathcal{Q}} \Rightarrow K$), we say that $\mathcal{P}$ is more efficient than $\mathcal{Q}$, denoted $\mathcal{P} \prec \mathcal{Q}$, if the following property is satisfied:

In every computation, $\forall i$, $\forall j$, $\forall m$ sent by P_i to P_j, $\forall k$:

$$K_{\mathcal{Q}}(m, k) \Rightarrow K_{\mathcal{P}}(m, k) \wedge \neg(K_{\mathcal{P}}(m, k) \Rightarrow K_{\mathcal{Q}}(m, k)).$$

In fact, this property means that:

- on the one hand, in every computation,

  $$\forall m,\ \forall k\ :\ \neg K_{\mathcal{P}}(m, k) \Rightarrow \neg K_{\mathcal{Q}}(m, k)$$

  and thus, every pair $(k, VC[k])$ attached to m when protocol $\mathcal{P}$ is executed will also be attached to m when protocol $\mathcal{Q}$ is executed;

- on the other hand, there exists a computation,

  $$\exists m\ \exists k\ : K_{\mathcal{P}}(m, k) \wedge \neg K_{\mathcal{Q}}(m, k)$$

  and thus, there exists a message m and a pair $(k, VC[k])$ that is attached to m when $\mathcal{Q}$ is executed but is not attached to m when $\mathcal{P}$ is executed.

To summarize, $\mathcal{P} \prec \mathcal{Q}$ means that the size of control information transmitted by messages is strictly smaller with $\mathcal{P}$ than with $\mathcal{Q}$. Below, we show that $\mathcal{P}1 \prec E\_SK$. Since $\mathcal{P}1$ is based on condition K_M and E_SK is based on condition K_{E_SK}, this amounts to show that $K_{E\_SK} \Rightarrow K_M$ (Lemma 5.4.4) and $\neg(K_M \Rightarrow K_{E\_SK})$ (Lemma 5.4.5).

Lemma 5.4.4. *For every pair of indices i, j, for every message m sent from P_i to P_j, $\forall k$: $K_{E\_SK}(m,k) \Rightarrow K_M(m,k)$.*

Proof. Consider a computation and two processes P_i and P_j in this computation. Let $m \in \mathcal{M}_{ij}$ and k such that $\neg K_M(m,k)$, i.e., $send(m).M_i[j,k] = 0$.

As $M_i[j,j] = 1$ and $M_i[i,j] = 1$ at all time, it is sufficient to consider $j \neq i$, $k \neq j$ and the two cases $k = i$ and $k \neq i$.

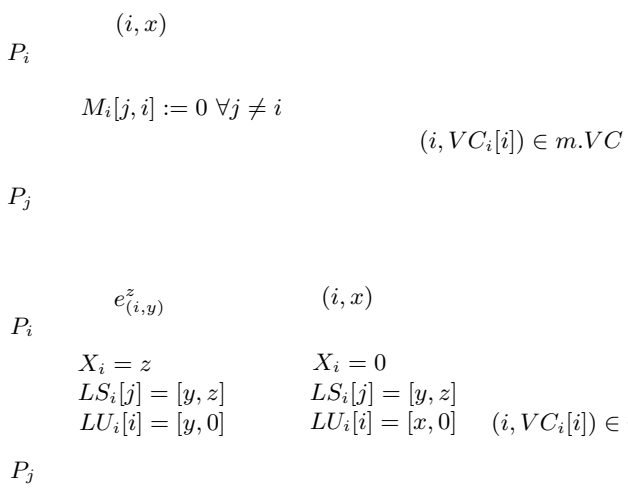


Figure 5.3: Proof of Lemma 5.4.4

1. $k = i$ (and $j \neq i$).

   Let $x = send(m).VC_i[i]$. By construction, $e_{i,x}$ is the last relevant event produced by P_i before $send(m)$ (see Figure 5.3.1).

   If $\mathcal{P}1$ is executed, from rule [M1], we have $e_{i,x}.M_i[j,i] = 0$. Since no relevant event occurred between $e_{i,x}$ and $send(m)$, and since by assumption $send(m).M_i[j,i] = 0$, rule [M2] implies that P_i has not sent any message to P_j between $e_{i,x}$ and $send(m)$.

   If E_SK is executed on the same computation, rules [E-SK0]-[E-SK3] managing the data structures LU_i and LS_i imply that $send(m).LU_i[i] =$

(x, α) and $send(m).LS_i[j] = (y, \beta)$ with $y < x$ (possibly $y = 0$). Thus, $send(m).LS_i[j] < send(m).LU_i[i]$, that is $\neg K_{E\_SK}(m, i)$.

2. $k \neq i$ (and $j \neq i$, $j \neq k$).

   The assumption $send(m).M_i[j, k] = 0$ implies that:

   - m is not the first message in $\mathcal{M}_{ij}$,
   - If m' denotes the message preceding m in $\mathcal{M}_{ij}$, then since the event $send(m')$ P_i has received a message m'' from some P_ℓ ($\ell \neq k$), piggybacking the pair $(k, m''.VC[k])$ and $m''.VC[k] > pred_l(receive(m'')).VC_i[k]$ (see Figure 5.3.2).
     Let $x - send(m').VC_i[i]$ and $y = receive(m'').VC_i[i]$. Thus, $send(m).LU_i[k] = (y, \beta)$ and $send(m).LS_i[j] = (x, \alpha)$ with $x < y$ or $(x = y) \wedge (\alpha < \beta)$. In other words, $send(m).LU_i[k] > send(m).LS_i[j]$, i.e., $\neg K_{E\_SK}(m, k)$.

□

Lemma 5.4.5. *There exists a computation, $\exists i$, $\exists j$, $\exists m$ sent from P_i to P_j, $\exists k$ such that $\neg K_{E\_SK}(m, k) \wedge K_M(m, k)$.*

Proof. Consider the computation described Figure 5.4. If protocol $\mathcal{P}1$ is executed on this computation, the message m' must transmit the pair $(k, send(m').VC_j[k]) = (k, 1)$.

When P_i receives m', we have $pred_l(receive(m')).VC_i[k] = 0$ and thus, from rule [MR3], $receive(m').VC_i[k] = 1$ and $receive(m').M_i[j, k] = 1$. So, when P_i sends m'' to P_j, the condition $K_M(m'', k)$ holds.

Now, consider the execution of protocol E_SK on the same computation. The message m' must transmit the pair $(k, send(m').VC_j[k]) = (k, 1)$ and when P_i receives m', following the application of rule [E-SK3], we have: $receive(m').VC_i[k] = 1$ and $receive(m').LU_i[k] = (0, 1)$. Moreover, $receive(m').LS_i[j] = (0, 0)$. So, when P_i is about to send m'' to P_j, we have $pred_l(send(m'')).LU_i[k] = (0, 0) < pred_l(send(m'')).LS_i[k] = (0, 1)$, hence $\neg K_{E\_SK}(m'', k)$.

□

Theorem 5.4.6. *$\mathcal{P}1 \prec$ E-SK.*

Proof. The proof directly follows from the Lemmas 5.4.4 and 5.4.5. □

In both $\mathcal{P}1$ and E_SK, the reduction of the communication overhead is obtained at the expenses of an extra storage space at each process. $\mathcal{P}1$ uses only a local boolean matrix of size n^2. E_SK uses two arrays of pairs of integers (size $2n$) plus one integer.

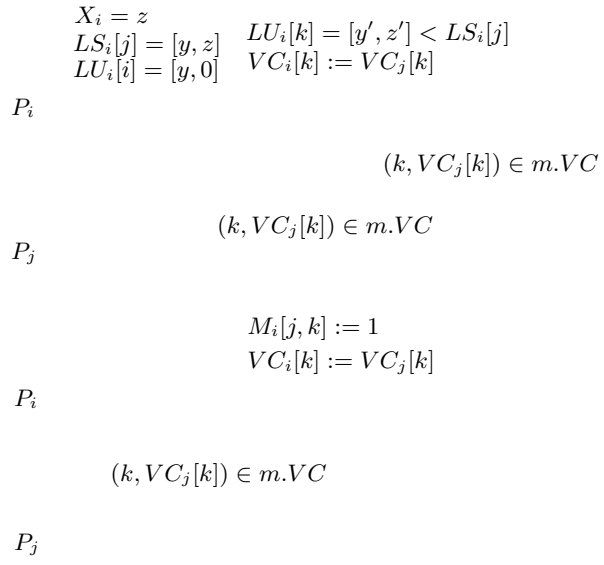


Figure 5.4: Proof of Lemma 5.4.5

5.5 Conclusions and Future Work

This chapter has addressed the tracking of the causality relation on a subset of events of asynchronous message-passing distributed computations, which are defined as "relevant" from the application point of view. A suite of simple and efficient causality-tracking protocols has been presented. These protocols mainly focused on the *reduction of the size of message timestamps* (i.e., the number of entries of a message timestamp can be less than n). These protocols do not require assumptions on channel behavior (such as FIFO), or on the connectivity of the communication graph or the communication pattern. Nevertheless, when FIFO channels are available, protocols are strictly more efficient than the extended Singhal-Kshemkalyani's protocol.

From a practical point of view this suite can be used as a core of an adaptive timestamping software layer. This layer can be used by a distributed application to timestamp its messages in an efficient way.

A part of results of this chapter appears in [22, 24].

As far as future work, an important issue is to carry out to carry out some simulation study similar to the one presented at the Chapter 4 for the protocols proposed in this chapter.

Chapter 6

Minimal Size of Message Timestamps for Tracking Causality between Relevant Events

In this chapter we are interested in investigating timestamping protocols which track causality between any pair of relevant events when *bounding* the number of entries of local vector in message timestamps to a predefined constant k, for $1 \leq k \leq n$. As non-relevant events can establish causal dependencies on relevant events, from a "causality tracking" perspective, a protocol given in a context in which all the events are relevant does not necessarily work when only some events are relevant.

This aspect of causality tracking is addressed with reference to the k-dependency vectors protocol, introduced in Section 4.6 in the context where all the events are relevant, which actually represents a general scheme for causality tracking where the size of message timestamps is bounded to a parameter k statically defined. When $k = n$ this scheme reduces to the vector clocks protocol and, whatever the choice of the set of relevant events $O \subseteq E$, it characterizes causality on-the-fly. On the contrary, when $k < n$, causality could not be tracked on-the fly, and, more important, if the set of relevant events O is not correctly chosen, completeness can be impacted, i.e., some transitive dependency information between events can be irremediably lost.

In this chapter we provide an off-line characterization of the minimal size of piggybacked information ensuring the completeness of causality tracking between a given set of relevant events. This requires to associate a graph with the computation and to perform operations transforming this graph into a

transitively reduced weighted graph (WCG). This transformation is similar to a transitive closure followed by a transitive reduction.

The chapter is composed of five sections. In Section 6.1 we consider and discuss the k-dependency vectors protocol when tracking causality between any pair of relevant events and we show that in this case completeness can be impacted. Sections 6.2 and 6.3 survey two dual approaches proposed in the literature in order to ensure the completeness of encoding of a causality relation, namely (i) dynamically adapting the parameter k to the given set of relevant events O and (ii) dynamically adapting the set O to the given parameter k. The *b-Bounded Vectors protocol* it is introduced in Section 6.3.3 which combines the idea of bounding the size of message timestamps with the protocol $\mathcal{P}1$ introduced in Section 5.4.3 which uses a local boolean matrix to ulteriorly reduce the size of message timestamps. The minimal value of k ensuring a complete encoding of the (relevant) computation is characterized in Section 6.4. A correctness proof of this characterization is provided in Section 6.4.2. Section 6.5 concludes the chapter.

6.1 The problem

The issue we are interested in investigating concerns detection of causality between any pair of relevant events when *bounding* the number of entries of local vector in message timestamps to a constant k. Let us recall that in order to safely detect causality between any pair of relevant events the checker receives pairs $(e, e.DV)$, where e is a *relevant event* and $e.DV$ is the vector timestamp associated with it. By using these information, it maps the partial order of the computation $\widehat{O} = (O, \prec_o)$ in a partial order $\widehat{O}_c = (O, \prec_c)$ called an *observed dependency partial order* (resp. observed dependency graph), defined as follows:

- its elements are relevant events e received by the checker, labeled with the timestamp $e.DV$.
- there is a dependency relation from e to e', denoted $e \prec_c e'$ iff $e.DV[i] \leq e'.DV[i]$, P_i being the process producing e.

Let $\downarrow_c (e)$ be defined as the set $\downarrow_c (e) = \{e' \mid e' \prec_c^+ e\} \cup \{e\}$.
In order to build the transitive closure $\widehat{O}_c^+$ [10, 36], the checker computes, for every relevant event (vertex in the observed dependency graph) e:

$$e.T_c = \max_{e' \in \downarrow_c (e)} e'.DV \tag{6.1.1}$$

We note that this corresponds to execute the reconstruction algorithm presented in Section 4.6.3, which is a modification of the algorithm proposed in [3] and [10] to reconstruct vector clocks by back-tracking k-dependencies.

If the protocol characterizes causality (eventually with delay), then, following dC-P property,

- $e \prec_c e' \Rightarrow e \prec_o e'$, i.e., it is *consistent* with causality, and
- $e \prec_o e' \Rightarrow e \prec_c^+ e'$, i.e., it is *complete*.

In this case, $\widehat{O}_c^+$ is isomorphic to $\widehat{O}$ and $e.T_c = e.VC$, i.e., the vector $e.T_c$ computed by the checker is actually the vector clock $e.VC$.

6.1.1 About the k-Dependency Vectors Protocol

The k-Dependency Vectors protocol presented in Section 4.6.2 represents a general scheme for protocols which manage message timestamps of size bounded to a parameter k statically defined and ranging up to n. Theorem 4.6.1 proves that it characterizes causality with delay.

When considering causality tracking between relevant events in a given set $O \subseteq E$ (we recall we are assuming that only internal events can be relevant events), this protocol can be easily improved as for Vector Clocks in such a way that the local vector $k\text{-}DV_i[i]$ counts only relevant events produced by P_i.

The case $k = n$. When $k = n$, the previous scheme reduces to the known *Vector Clocks protocol* [17, 30], and $e.n\text{-}DV$ is really the vector $e.VC$. In this case, *whatever the choice of* $O \subseteq E$, $\widehat{O}_c$ is isomorphic to $\widehat{O}$. In particular, it is already transitively closed (i.e., $\widehat{O}_c^+ \equiv \widehat{O}_c$).

The case $k < n$. On the contrary, when $k < n$, the partial order $\widehat{O}_c$ is not necessarily transitively closed, as depicted in Figure 6.1. *More important*, if the set of relevant events O is not correctly chosen, $\widehat{O}_c^+$ is not necessarily isomorphic to $\widehat{O}$ (an example is depicted in Figure 6.2) and some transitive information can be irremediably lost if it was not previously recorded by a relevant event, that is if it has not been previously sent to the observer process. So, the *completeness* can be impacted [26].

Example

Figure 6.1 shows the $\widehat{O}_c$ produced by using the 1-dependency protocol, whose transitive closure $\widehat{O}_c^+$ is isomorphic to $\widehat{O}$. If we consider the event $e_{3,2}$, we have $e_{3,2}.T_c = max_{e' \in \downarrow_c(e_{3,2})} e'.k\text{-}DV = max([0,2,2],[0,0,1],[1,2,0],[0,1,0],[1,0,0]) = [1,2,2] = e_{3,2}.VC$.

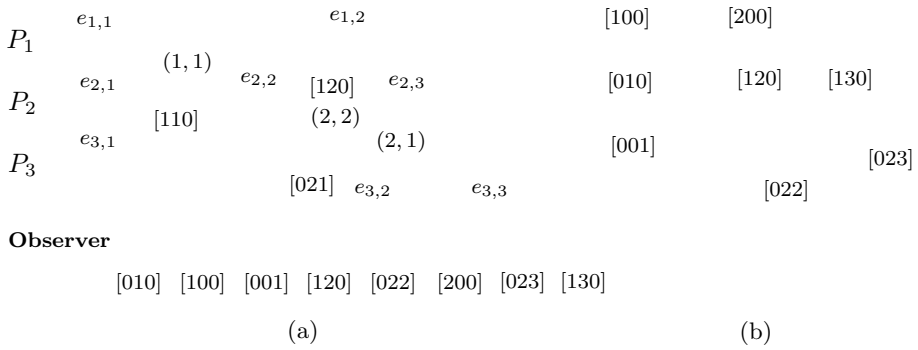


Figure 6.1: A distributed computation using the 1-Dependency Vectors protocol (a) and the related $\widehat{O}_c$ (b).

In Figure 6.2 is depicted the same computation when the relevant event $e_{2,2}$ has been eliminated. In this case, $\widehat{O}_c^+ \subset \widehat{O}$. In fact, $e_{3,2}.T_c = max_{e' \in \downarrow_c(e_{3,2})} e'.k\text{-}DV = [0, 1, 2]$ instead of the correct value $e_{3,2}.VC = [1, 1, 2]$. This means that the causal dependency $e_{1,1} \prec_o e_{3,2}$ cannot be retrieved from the $\widehat{O}_c$. This is obvious by observing the figure 6.2: there is no path from $e_{1,1}$ to $e_{3,2}$ in the $\widehat{O}_c$.

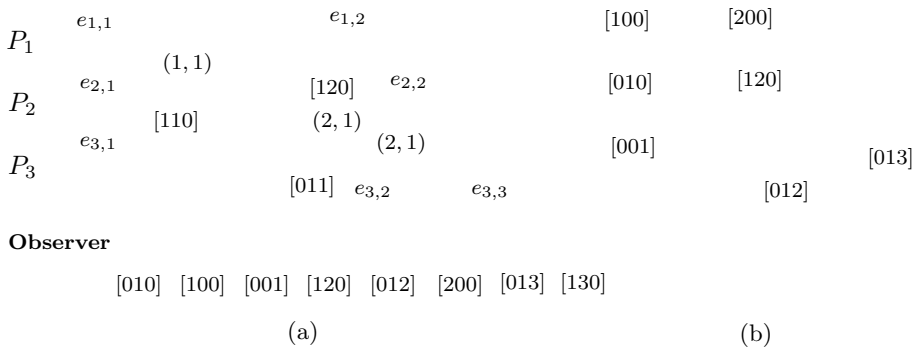


Figure 6.2: A distributed computation using 1-Dependency Vector protocol (a) and the related "incomplete" $\widehat{O}_c$ (b).

Approaches

To ensure the completeness of encoding, up to now two dual approaches have been proposed in the literature: (i) dynamically adapting the value of k to the given set O, and (ii) dynamically "adapting" the set O to the static value k.

6.2 Adapting the size k to O

Jard and Jourdan [26] proposed the *incremental transitive method*, which is *adaptive* with respect to the choice of O. In this case the rules for updating local vectors DV_i and assigning timestamps are the following:

[R1'] when an event e is observed:

- $DV_i[i] := DV_i[i] + 1$;
- $e.DV := DV_i$;
- $\forall j \neq i$ **do** $DV_i[j] := 0$ (*reset step*);

[R2'] upon sending a message m to P_j, a list $m.L$ is piggybacked towards P_j containing all pairs $(j, DV_i[j])$ such that $DV_i[j] \neq 0$;

[R3'] upon reception from P_j with the list $m.L$:

- $\forall j : \ (j, DV[j]) \in m.L$ **do** $DV_i[j] := max(DV_i[j], DV[j])$.

With respect to the k-dependency vectors protocol, there are two main differences:

1. the size of message timestamps is variable, although it can grow up to n. Namely, the size depends on the number of processes traversed without encountering any relevant events.

2. all local information is propagated while no relevant event occurs. When such event e occurs, the information is recorded in the timestamp $e.DV$ and sent to the checker process, so the propagation can stop (*reset step* in rule [R1']). This allows a process to "forget" the information received before the last relevant event.

6.3 Adapting O to a static k

6.3.1 The NIVI Condition

In [19, 26], a sufficient condition on O ensuring the completeness of the direct dependency protocol (i.e., 1-dependency vectors protocol) has been stated. This condition, called NIVI (Non InVisible Interaction), stipulates that after a receipt event there must be a relevant event before a send event (on the same process) occurs.

At the operational level, processes have to define *forced* relevant events when this condition is about to be violated. If the NIVI condition on O is satisfied (or forced), the k-dependency vectors protocol is complete, for any given

$k \geq 1$. In fact, even if in the local vector k-DV_i there are more than k different dependency information, piggybacking the direct dependency is sufficient to ensure the completeness. The other event identifiers have already been recorded in some relevant event. They are redundant and their piggybacking only improves the performance of the reconstruction algorithm presented in Section 4.6.3, in terms of detection delay of causal dependencies between events. That is, piggybacking more than one components of the local vector only impacts performance in terms of graph availability.

6.3.2 Bounding the Size of Message Timestamps at run-time

This method offers an additional service: suppose that we do not want to piggyback more than k integers. So far, the size of the local vectors can be bigger, when there is a path going through more than k processes without relevant events.

So, we add the following *test*: upon sending of a message m, if the there are more than k positive entries then we automatically generate a null relevant event, which resets the local vector.

6.3.3 k-Bounded Vectors Protocol

In [22] the *k-Bounded Vectors protocol* it is presented which combines the test idea upon sending a message, to guarantee the "k" constraint, i.e. that any message piggybacks at most k relevant event identifiers (i.e., pairs of process id + seq number), with the protocol $\mathcal{P}1$, introduced in Section 5.4.3, which uses a local boolean matrix to ulteriorly reduce the size of message timestamp with respect to the previous protocol.

Of course, to be meaningful with respect to the tracking of causality it has to remain possible to reconstruct (possibly off line) the vector clock of any relevant event.

Protocol $\mathcal{BP}1$

This can be easily realized by adding the following test to the rule [MR2] of protocol $\mathcal{P}1$.

An additional test. [Test] When P_i has to send a message m to P_j, it first computes $d = \mid \{h : (M_i[j,h] = 0)\} \mid$, i.e., the number d of event identifiers that P_i has to transmit upon message. If $d > k$, then P_i generates a "null" relevant event (so, it executes rule [MR1], which resets d to 1), and afterwards attaches to m its dependency timestamp before sending this message.

In addition to the previous points, an important difference between protocol $\mathcal{P}1$ and protocol $\mathcal{BP}1$ lies in the update of the matrix M_i when P_i produces a relevant event e (Rule [MR1]). In fact, M_i is reset in such a way that P_i "forgets all the past", except the event e. More precisely, all entries of M_i are set to 1, except the ith column whose entries (but $M_i[i,i]$) are reset to 0. The resulting protocol $\mathcal{BP}1$ is as follows.

[k-MR0] Initialization:

- $DV_i[1..n]$ is initialized to $[0,\ldots,0]$,
- $\forall\ (j,h):\ M_i[j,h]$ is initialized to 1.

[k-MR1] Each time P_i produces a relevant event e:

- it increments its dependency vector entry $DV_i[i]$ ($DV_i[i] := DV_i[i]+1$),
- it associates with e the timestamp $e.DV = DV_i$,
- it resets the boolean matrix as follows:
  - $\forall (j,h)$ such that $j,h \neq i:\ M_i[j,h] := 1$ ([1]).
  - $\forall j \neq i:\ M_i[j,i] := 0$.

[k-MR2] When P_i sends a message m to P_j, it does the following:

- P_i first computes $d =\mid \{h:\ (M_i[j,h]=0)\} \mid$. If $d > k$, then P_i generates a "null" relevant event (so, it executes R1, which resets d to 1).
- Then, before sending m, P_i attaches to it the following set of event identifiers: $m.DV = \{(h, DV_i[h]) \mid M_i[j,h]=0\}$.

[k-MR3] When P_i receives a message m from P_j, it executes the following updates:

$\forall (h, DV_j[h]) \in m.DV$ **do**:
case $DV_i[h] < m.DV[h]$ **then** $DV_i[h] := m.DV[h]$;
$\forall \ell \neq i,j,h : M_i[\ell,h] := 0;\ M_i[j,h] := 1$
$DV_i[h] = m.DV[h]$ **then** $M_i[j,h] := 1$
$DV_i[h] > m.DV[h]$ **then** *skip*
endcase

[1] Note that the ith row of M_i has not to be updated as it is always equal to $(1,\ldots,1)$ (see Section 5.4.3).

So, $\mathcal{BP}1$ is incrementally obtained from $\mathcal{P}1$ by

1. replacing VC_i by DV_i;
2. extending the reset rule in [MR1];
3. adding a test in [MR2].

Moreover, let us note that, as before the transformation $T_{1\rightarrow 2}$ can be applied to $\mathcal{BP}1$ to get a protocol $\mathcal{BP}2$.

Discussion

Both previous approaches are insufficient in the sense that when bounding the size at run-time, in the worst case the size of message timestamps could grow up to n.

On the one hand, bounding the size of piggybacked information, as in the k-dependency protocol, requires additional condition on the set O and the NIVI condition is too restrictive.

Remark 6.3.1. When the NIVI condition is not satisfied, the *reset step is necessary*. In fact, if the reset step is not performed, a process cannot distinguish the "new" information received since the last relevant event from the "old" ones already transmitted to the checker.

In the following section we are interested in finding the minimal size k of piggybacked information in order to guarantee the completeness of the encoding with respect to a given distributed computation and a given set O of relevant events. So, the protocols considered must not introduce additional relevant events and thus must execute a reset step when an event is observed on a process. The next section investigates this relation.

6.4 Characterizing the Minimal Size of Message Timestamps

Given a distributed computation $\widehat{E}$ and a given set $O \subseteq I$ of relevant events, the *minimal* value of k ensuring a complete encoding of $\widehat{O}$ when the size of piggybacked information is limited to k will be characterized.

6.4.1 Construction of a Weighted Covering Graph

This characterization is based on the construction of a weighted graph from which this value can be easily computed.

The Idea

The idea is based on the following observation. Let us consider a process P_i where a sequence of l consecutive *receive* events occurs, each message being received from a different sender.

If the first event following this sequence is a *send* event, it means that the sent message must propagate l causal dependency information towards the relevant event that will follow $receive(m)$.

If, on the contrary, the first event following this sequence is relevant, then all causal dependency information brought by these l messages is transferred towards the observer, and consequently has not to be propagated on the messages that will be sent by P_i after the relevant event.

Thus, every message sent appears to be the messenger of a certain amount of information, depending on how many causal information is stored on the sending process. However, this view is too pessimistic, because it does not take into account the fact that some information could be transferred from a relevant event to another via several alternative paths in the distributed computation.

A careful analysis of this situation leads to the construction of a weighted graph on the set O, whose maximum edge weight is equal to the required minimal value of k.

The construction of this graph is carried out in three steps:

1. a *propagation* graph, built on O and additional vertices modelizing the sequences of receipt events followed by a send event,
2. a weighted *covering* graph, whose transitive closure is isomorphic to $\widehat{O}$, obtained by suppressing additional vertices from the propagation graph and computing weights accordingly, and
3. the transitive reduction of the weighted covering graph, whose aim is to remove redundant information transmission.

Propagation Graph

Let $PG = (V, \prec_{pg})$ defined as follows:

- $V = O \cup P$, where
- an O-vertex is generated by each relevant event, and
- a P-vertex is generated every time that a non empty sequence of consecutive receipt events is followed by a send event (on the same process).

Notations.

- Given two vertices X and Y generated on the same process P_i, we say that $X = pred(Y)$ or, equivalently, $Y = succ(X)$ if the events generating X and Y occur in P_i in that order, and there is no vertex between X and Y.
- With every message m of the computation are associated two vertices $\overleftarrow{m}$ and $\overrightarrow{m}$ in V:
  - $\overleftarrow{m}$ is the vertex corresponding to the last non send event that locally precedes $send(m)$;
  - $\overrightarrow{m}$ is the vertex corresponding to the first non receive event that locally follows $receive(m)$, that is, $\overrightarrow{m}$ is an O-vertex if $receive(m)$ is followed by a relevant event, a P-vertex otherwise.

The set of edges $\prec_{pg}$ of Propagation Graph is defined in this way: there is an edge $X \prec_{pg} Y$ iff

- $X = pred(Y)$, or
- there exists a message m such that $X = \overleftarrow{m}$ and $Y = \overrightarrow{m}$.

O-vertices are labeled like their corresponding relevant event, whereas $p_{i,x,h}$ denotes the h-th P-vertex on process P_i, between O-vertices $e_{i,x}$ and $e_{i,x+1}$.

Example. In Figure 6.3 is shown the propagation graph (b) associated with the distributed computation depicted in (a). O-vertices are depicted as rounded rectangles, whereas P-vertices are depicted as sharp rectangles.

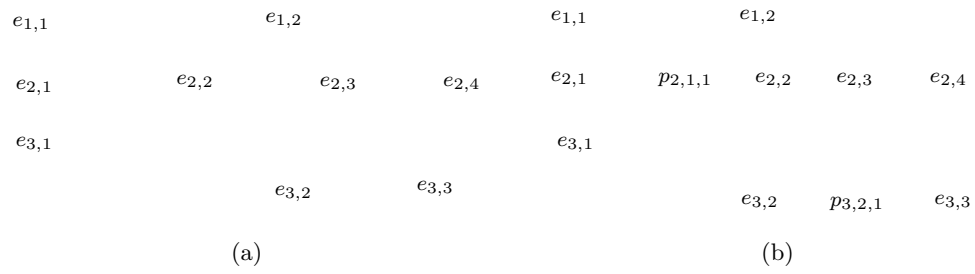


Figure 6.3: a relevant distributed computation (a) its associated propagation graph (b).

Reduction Procedure to Weighted Covering Graph

The weighted covering graph $WCG = (O, \prec_{wcg})$ is a partial representation of $\widehat{O}$ where an edge from e to e' represents a path in the time diagram starting form e and ending to e' without relevant event between e and e'.

At the end of its construction, the weight of each edge from e to e' will represent the minimum size of the piggybacked information which guarantees that the information on e is propagated towards e' without any relevant events after leaving e.

This graph is built from the propagation graph by iteratively removing its P-vertices and replacing the removed edges by weighted ones linking only O-vertices. Rules for building WCG are described as follows.

Initially, $WCG = PG$ with all weights equal to 1. For each P-vertex X whose all predecessors are O-vertices:

- for each Y such that $Y \prec_{wcg} X$,

  if $\neg(Y \prec_{wcg} succ(X))$ then create an edge $Y \prec_{wcg} succ(X)$ with $w(Y, succ(X)) = w(Y, X)$,

  else $w(Y, succ(X)) = \min(w(Y, succ(X)), w(Y, X))$.

- for each Z such that $X \prec_{wcg} Z$, create an edge $prec(X) \prec_{wcg} Z$ with $w(prec(X), Z) = 1$.

- for each $Y \neq prec(X)$, $Z \neq succ(X)$ such that $Y \prec_{wcg} X \prec_{wcg} Z$,

  if $\neg(Y \prec_{wcg} Z)$ then create an edge $Y \prec_{wcg} Z$ with $w(Y, Z) = card(\{y | y \prec_{wcg} X\})$,

  else $w(Y, Z) := \min(w(Y, Z), card(\{y | y \prec_{wcg} X\})$.

- remove X from WCG.

The construction stops when there are no more P-vertices in WCG. It is easy to see that, by construction, $WCG^+ = \widehat{O}$.

Example. In the figure 6.4 is depicted an execution of the reduction's procedure, the propagation graph (a) being the input and the weighted covering graph (c) the output obtained after eliminating two P-vertices $p_{2,1,1}$ and $p_{3,2,1}$, as shown in (b) and (c), respectively.

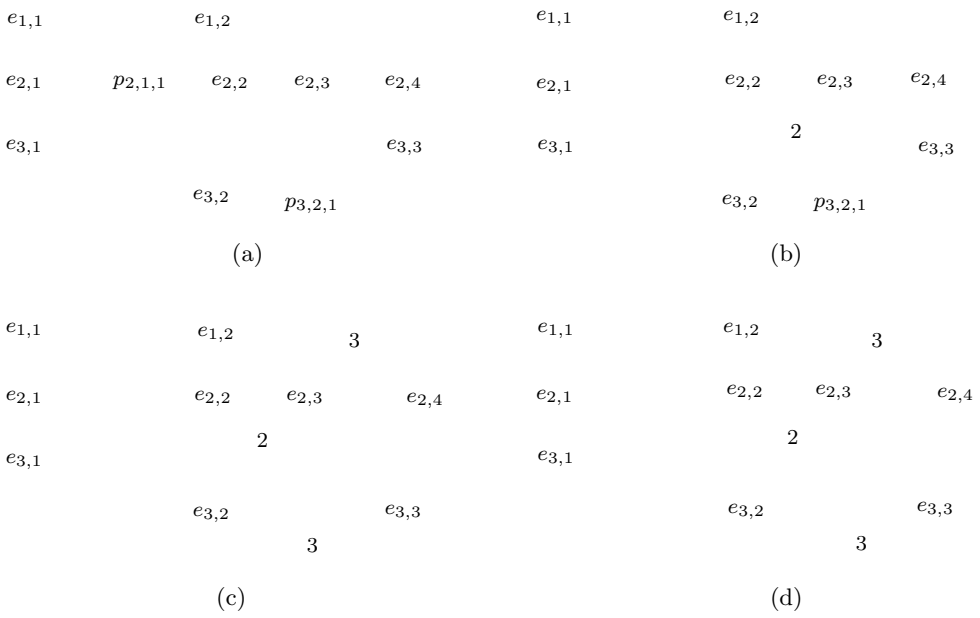


Figure 6.4: An example of execution of the reduction procedure (a,b,c) on the propagation graph associated with the distributed computation depicted in Figure 6.3.

Transitive Reduction of the Weighted Covering Graph.

The Weighted Covering Graph obtained in the previous section can exhibit different paths between two relevant events. Clearly, if a path is "doubled" by an edge, this indicates that the information transmitted along the edge is also transmitted by the path.

So, removing the edge does not modify the possibilities of information transmission represented by the rest of WCG. This well-known procedure, called transitive reduction, can be carried out on WCG, to obtain the graph WCG^{-}. This graph allows to state the desired characterization.

6.4.2 The Characterization

Theorem 6.4.1 (Minimal piggybacking size). *Let $\widehat{E} = (E, \rightarrow)$ be a distributed computation, and O be a set of relevant events. The value k necessary and sufficient to be used in an observation method in order to obtain a complete encoding of the sub-order $\widehat{O}$ is the maximum weight in the transitive reduction WCG^{-} of the weighted covering graph, that is:*

$$w_{max} = \max\{w(e, e') \mid e \prec^{-}_{wcg} e'\}.$$

As an example, Figure 6.4.(d), representing the transitive reduction of the weighted covering graph (c), shows that $w_{max} = 1$ in such a case. Previous property affirms that it is necessary and sufficient to piggyback upon messages only one integer in order to ensure the completeness of the encoding of $\widehat{O}$.

6.4.3 Proof

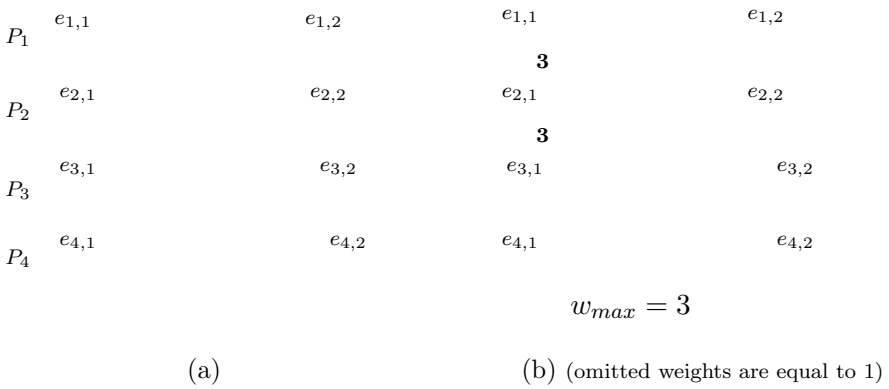
The proof is made of two parts: necessity and sufficiency.

Necessity. We show that, if $k < w_{max}$ then no encoding can be complete. For that purpose, it is sufficient to exhibit a scenario where there exists a relevant event e such that $T_c(e) \neq V(e)$, where $T(e)$ is the timestamp of e in this encoding.

The figure 6.5 describes a computation with relevant events (a) and its associated graph WCG^- (b) built according to he previous procedures. From WCG^- follows that $w_{max} = 3$. Now, let's consider an encoding allowing only $k = 2$ event identifiers per message.

Figure 6.6 shows the values piggybacked on messages (a), the resulting timestamps and the $\widehat{O}_c$ graph (b). From this follows that $T_c(e_{4,2}) = [0, 1, 1, 2] \neq V(e_{4,2}) = [1, 1, 1, 2]$. Clearly, the causal dependency $e_{1,1} \prec_o e_{4,2}$ cannot be retrieved.

Remark that another choice of vector entries to piggyback on the message m (in bold on Figure 6.6-a) would have led to the loss of causal dependency $e_{2,1} \prec_o e_{4,2}$ instead.



(a) (b) (omitted weights are equal to 1)

Figure 6.5: A computation and its associated WCG^-.

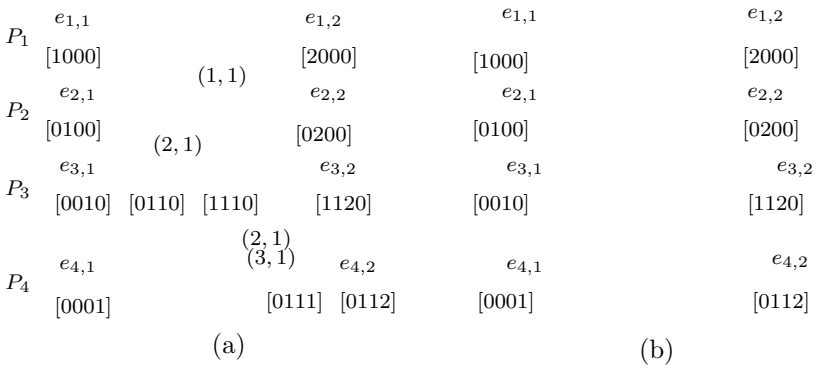


Figure 6.6: Loss of causal dependencies.

Sufficiency. We show that, if $k \geq w_{max}$, it is possible to obtain a complete encoding. For that purpose, we define a protocol that ensures a complete encoding under this assumption. This protocol is based on the k-dependency method with a reset step. It is based on the following rules (see rules [R1'] to [R3'] given in Section 6.2):

[R1"] upon observation of an event e on P_i: $V_i[i] := V_i[i] + 1$; $T(e) := V_i$; $\forall j \neq i \ : \ V_i[j] := 0$; record $(e, T(e))$ on the observer (reset step);

[R2"] upon sending a message m, P_i piggybacks on this one a list $m.L$ containing the pair $(i, V_i[i])$ and at most $k-1$ pairs $(j, V_i[j])$ such that $V_i[j] \neq 0$;

[R3"] upon receiving a message with the list $m.L$, for each j such that $(j, V[j]) \in m.L$, $V_i[j] := max(V_i[j], V[j])$.

To show that this protocol provides a complete encoding, we must prove that $\forall e \in O \ : \ T_c(e) = V(e)$.

Let us consider the transitive reduction $\widehat{O}^-$, *i.e.*, $e' \prec_{o^-} e$ if and only if there is a path in the time diagram, starting from e' and ending to e without relevant event between e' and e. We will proof successively:

1. $e' \prec_{o^-} e \Rightarrow T(e')[i] \leq T(e)[i]$, where P_i is the process producing e' (Lemma 6.4.2).

2. $\forall e \in O \ : \downarrow_o (e) = \downarrow_c (e)$ (Lemma 6.4.3).

3. and, finally, $\forall e \in O \ : \ T_c(e) = V(e)$.(Lemma 6.4.4).

Lemma 6.4.2. *Under the assumption $k \geq w_{max}$, we have:*
$e' \prec_{o^-} e \Rightarrow T(e')[i] \leq T(e)[i]$, *where P_i is the process producing e'.*

Proof. Let $e' \prec_{o^-} e$, e' produced by P_i and e produced by P_j.

If $i = j$ then, from rule 1, $T(e')[i] < T(e)[i]$.

Consider now the case $i \neq j$. Let $O^-(e) = \{e' \mid e' \prec_{o^-} e\} \setminus \{prec(e)\}$, where $prec(e)$ denotes the relevant event preceding e on P_j (if any). If $O^-(e) \neq \emptyset$, P_j has received messages $m_1, m_2, \ldots, m_\alpha$ from different senders after $prec(e)$ (or since the start of P_j if e is the first relevant event) and before e.

For $q = 1, 2, \ldots \alpha$ let us denote $X_q(e) = O^-(e) \cap \downarrow_c (send(m_q))$ (*i.e.*, $X_q(e)$ is the set of relevant events such that there is a path from e' to e without relevant event between e' and e, and whose last message is m_q, see Figure 6.7).

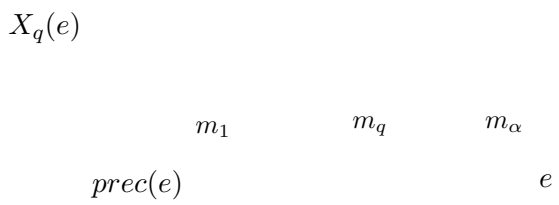


Figure 6.7: Definition of sets $X_q(e)$.

We have $O^-(e) = \bigcup_{q=1}^{\alpha} X_q(e)$ (remark that these sets are not necessarily disjoint). By the construction of WCG, for each $e' \in O^-(e)$ there is an edge from e' to e in WCG, and $w(e', e) = \min(card(X_q(e)) \mid e' \in X_q(e))$. In fact, the sub-graph of $\widehat{E}$ spanned by $\{e\} \cup O^-(e)$ generates during the construction of the propagation graph O-vertices corresponding to e and events $e' \in O^-(e)$, and P-vertices. These P-vertices are eliminated by the reduction procedure producing WCG and generate WCG edges whose weight satisfy the above property.

Let $q(e')$ denote an index for which this minimum is obtained, that is $w(e', e) = card(X_{q(e')}(e))$. By the definition of w_{max}, we have $card(X_{q(e')}(e)) \leq w_{max} \leq k$.

Due to the reset step (rule [R1"]), each message m_q has to piggyback a list of exactly $card(X_q(e))$ event identifiers, comprising all the entries $T(e'')[l(e'')]$ for $e'' \in X_q(e)$, e'' being produced by process $P_{l(e'')}$. Thus, for each $e' \in O^-(e)$, $m_{q(e')}$ has to piggyback a list of exactly $card(X_{q(e')}(e))$ event identifiers, comprising the entry $T(e')[i]$. Since $card(X_{q(e')}(e)) \leq k$ the entry $T(e')[i]$ will

be piggybacked on m_q. From rule 3 (applied to the receipt of $m_{q(e')}$), $V_j[i]$ is updated and $V_j[i] \geq T(e')[i]$. Finally, from rule 1 applied to the observation of e, $T(e)[i] \geq T(e')[i]$. □

Lemma 6.4.3. *Under the assumption $k \geq w_{max}$, we have: $\downarrow_o (e) = \downarrow_c (e)$.*

Proof. By the definition of $\widehat{O}_c$, we have $\downarrow_c (e) \subseteq \downarrow_o (e)$.

To prove the converse, let us consider $e' \in \downarrow_o (e)$. There is a sequence $e' \prec_{o^-} e_1 \prec_{o^-} e_2 \prec_{o^-} \ldots \prec_{o^-} e_\beta \prec_{o^-} e$, where e' is produced by P_i, e_l is produced by P_{i_l} $(l = 1, \ldots, \beta)$ and e is produced by P_j.

From Lemma 6.4.2 we have:
$T(e')[i] \leq T(e_1)[i],\ T(e_1)[i_1] \leq T(e_2)[i_2], \ldots, T(e_k)[i_\beta] \leq T(e)[i_\beta]$.
The observer process records the relevant events together with their timestamp T and builds the ODG. By definition of ODG edges, there is a path $[e', e_1, \ldots, e_\beta, e]$ in the ODG. Thus, $e' \in \downarrow_c (e)$. □

Lemma 6.4.4 (Sufficiency). *Under the assumption $k \geq w_{max}$, we have: $\forall e \in O\ :\ T_{obs(e)} = V(e)$.*

Proof. The proof is by induction on the poset $\widehat{O}$.

Base case. If e has no predecessor in $\widehat{O}$ (*i.e.*, e is the first relevant event of a process), then $V(e) = T(e) = T_c(e)$.

Induction. Suppose $\forall e' \in \downarrow_o (e) \setminus \{e\}$ we have $V(e') = T_c(e')$. Remark that, by construction, $\forall e \in O\ :\ T(e)[j] = V(e)[j] \wedge \forall i \neq j\ T(e)[i] \leq V(e)[i]$ (where e is produced by process P_j). Thus,

$$\begin{aligned}
V(e) &= \max(\max_{e' \in \downarrow_o(e) \setminus \{e\}} V(e'), T(e)) \\
&= \max(\max_{e' \in \downarrow_o(e) \setminus \{e\}} T_c(e'), T(e)) \quad \text{(induction assumption)} \\
&= \max_{e' \in \downarrow_o(e)} T_c(e') \\
&= \max_{e' \in \downarrow_c(e)} T_c(e') \text{ (Lemma 6.4.3)} \\
&= T_c(e) \text{ (computation of } T_c(e) \text{ by the observer).}
\end{aligned}$$

□

6.5 Conclusions and Future Work

In this chapter we partially answered an open problem about the minimal size of piggybacked information ensuring the completeness of causal dependency tracking between a given set of relevant events. We provided an off-line

characterization of this minimal size and we proved that it is necessary and sufficient by using the technique of graph modelization, which is a powerful tool for analyzing properties of distributed computations.

This contribution is important, because, to our knowledge, no completeness characterization has been presented before.

Althought the weighted graph WCG can be completely determined only when all the computation is known, it can be used off line to perform a posteriori analysis of a computation.

A part of results of this chapter appears in [21].

A future work would be to find completeness conditions, less restrictive than the previously known NIVI condition, that could be tested on line. The result presented here constitutes a theoretical basis for such an investigation.

Bibliography

[1] Awerbuch, B. Complexity of network synchronization. *J. ACM 32(4)* (1985), 804–823.

[2] Baldoni, R. A positive acknowledgement protocol for causal broadcasting. *IEEE Transactions on Computers 47(12)* (1998), 1341–1350.

[3] Baldoni, R., Cioffi, G., Hélary, J. M., and Raynal, M. Direct dependency-based determination of consistent global checkpoints. *Journal of Computer Systems Science and Engineering* (2001). To appear.

[4] Baldoni, R., Marchetti-Spaccamela, A., Mechelli, M., and Melideo, G. Necessary conditions for characterizing causality in distributed computations. Tech. rep. 28-99, Dipartimento di Informatica e Sistemistica, University of Rome.

[5] Baldoni, R., Marchetti-Spaccamela, A., Mechelli, M., and Melideo, G. Timestamping algorithms: A characterization and a few properties. In *Proc. of the European Conference on Parallel Computing, Euro-Par 2000)* (2000), vol. 1900 of *LNCS*, pp. 609–616.

[6] Baldoni, R., Mechelli, M., and Melideo, G. A general scheme for dependency tracking in distributed computations. Tech. rep. 17-99, Dipartimento di Informatica e Sistemistica, University of Rome.

[7] Baldoni, R., and Melideo, G. A lower bound on the minimal information to encode timestamps in distributed computations. submitted.

[8] Baldoni, R., and Melideo, G. Tradeoffs in message overhead versus detection time in causality tracking. submitted to IEEE Transactions on Software Engineering, second revision.

[9] Baldoni, R., Quaglia, F., and Ciciani, B. On the "no-z-cycle" property in distributed executions. *Journal of Computer and System Sciences 61* (2000), 400–427.

[10] Baldy, P., Dicky, H., Medina, R., Morvan, M., and Vilarem, J. M. Efficient reconstruction of the causal relationship in distributed systems. In *Proc. of the First Canada-France Conference on Parallel and Distributed Computing* (1994), vol. 805 of *LNCS*, pp. 101–113.

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

[12] Chandy, K. M., and Lamport, L. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems 3(1)* (1985), 63–75.

[13] Charron-Bost, B. Combinatorics and geometry of consistent cuts: Application to concurrency theory. In *J. C. Bermond, M. Raynal eds. Distributed Algorithms* (1989), vol. 392 of *LNCS*, pp. 45–56.

[14] Charron-Bost, B. Concerning the size of logical clocks in distributed systems. *Information Processing Letters 39* (1991), 11–16.

[15] Davey, B. A., and Priestley, H. A. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[16] Fidge, C. J. Logical time in distributed computing systems. *IEEE Comput 24(8)*, 28–33.

[17] Fidge, C. J. Timestamps in message passing system that preserve the partial ordering. In *Proc. of 11th Australian Computer Science Conference* (1988), pp. 55–66.

[18] Fowler, J., and Zwaenepoel, W. Causal distributed breakpoints. In *Proc. of 10th IEEE International Conf. on Distributed Computing Systems* (1990), pp. 134–141.

[19] Fromentin, E., Jard, C., Jourdan, G. V., and Raynal, M. On-the-fly analysis of distributed computations. *Information Processing Letters 54*, 267–274.

[20] Fromentin, E., and Raynal, M. Shared global states in distributed computations. *Journal of Computer and System Sciences 55(3)*, 522–528.

[21] Hélary, J. M., and Melideo, G. Minimal size of piggybacked information for tracking causality: A graph-based characterization. In *Proc. of 26th International Workshop on Graph-Theoretic Concepts in Computer Science, WG 2000* (2000), pp. 49–61.

[22] Hélary, J. M., Melideo, G., and Raynal, M. Tracking causality in distributed systems: a suite of efficient protocols. In *Proc. of 7th International Colloquium on Structural Information and Communication Complexity, Sirocco 2000*, Carleton University Press.

[23] Hélary, J. M., Netzer, R., and Raynal, M. Consistency issues in distributed checkpoints. *IEEE Transactions on Software Engineering 25(2)*, 274–281.

[24] Hélary, J. M., Raynal, M., Melideo, G., and Baldoni, R. Efficient causality-tracking timestamping. *IEEE Transactions on Knowledge and Data Engineering* (2001). To appear.

[25] Hseush, W., and Kaiser, G. E. Modeling concurrency in parallel debugging. *ACM SIGPLAN Notice 25(3)*, 11–20.

[26] Jard, C., and Jourdan, G. V. Incremental transitive dependency tracking in distributed computations. *Parallel Processing Letters 6(3)*, 427–435.

[27] Lamport, L. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM 21(7)*, 558–565.

[28] Liskov, B., and Ladin, R. Highly available distributed services and fault-tolerant distributed garbage collection. In *Proc. of 5th ACM Symposium on Principles of Distributed Computing* (1986), pp. 29–39.

[29] Marzullo, K., and Sabel, L. Efficient detection of a class of stable properties. *Distributed Computing 8(2)*, 81–91.

[30] Mattern, F. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*.

[31] Panangaden, P., and Taylor, K. Concurrent common knowledge: Defining agreement for asynchronous systems. *Distributed Computing 6(2)*, 73–93.

[32] Parker, D. S., and *et al*, G. J. P. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering 9(3)*, 240–247.

[33] Peterson, L. L., Buchholz, N. C., and Schlichting, R. D. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems 7(3)*, 217–246.

[34] Raynal, M. A distributed algorithm to prevent mutual drift between n logical clocks. *Information Processing Letters 24*, 199–202.

[35] Raynal, M., Schiper, A., and Toueg, S. The causal ordering abstraction and a simple way to implement it. *Information Processing Letters 7(3)*, 343–350.

[36] Schwarz, R., and Mattern, F. Detecting causal relationships in distributed computations: in search of the holy grail. *Distributed Computing 7(3)*, 149–174.

[37] Singhal, M., and Kshemkalyani, A. An efficient implementation of vector clocks. *Information Processing Letters 43*, 47–52.

[38] Torres-Rojas, F. J., and Ahamad, M. Plausible clocks: Constant size logical clocks for distributed systems. *Distributed Computing 12(3)*, 179–196.

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.

VII-95-3 Antonella Cresti. *Unconditionally Secure Key Distribution Protocols.*

June 1995.

VII-95-4 Vincenzo Ferrucci. *Dimension-Independent Solid Modeling.* June 1995.

VII-95-5 Esteban Feuerstein. *On-line Paging of Structured Data and Multithreaded Paging.* June 1995.

VII-95-6 Michele Flammini. *Compact Routing Models: Some Complexity Results and Extensions.* June 1995.

VII-95-7 Giuseppe Liotta. *Computing Proximity Drawings of Graphs.* June 1995.

VIII-96-1 Luca Cabibbo. *Querying and Updating Complex-Object Databases.* May 1996.

VIII-96-2 Diego Calvanese. *Unrestricted and Finite Model Reasoning in Class-Based Representation Formalisms.* May 1996.

VIII-96-3 Marco Cesati. *Structural Aspects of Parameterized Complexity.* May 1996.

VIII-96-4 Flavio Corradini. *Space, Time and Nondeterminism in Process Algebras.* May 1996.

VIII-96-5 Stefano Leonardi. *On-line Resource Management with Application to Routing and Scheduling.* May 1996.

VIII-96-6 Rosario Pugliese. *Semantic Theories for Asynchronous Languages.* May 1996.

IX-97-1 Paola Alimonti. *Local search and approximability of MAX SNP problems.* May 1997.

IX-97-2 Tiziana Calamoneri. *Does Cubicity Help to Solve Problems?.* May 1997.

IX-97-3 Paolo Di Blasio. *A Calculus for Concurrent Objects: Design and Control Flow Analysis.* May 1997.

IX-97-4 Bruno Errico. *Intelligent Agents and User Modelling.* May 1997.

IX-97-5 Roberta Mancini. *Modelling Interactive Computing by exploiting the Undo.* May 1997.

IX-97-6 Riccardo Rosati. *Autoepistemic Description Logics.* May 1997.

IX-97-7 Luca Trevisan. *Reductions and (Non-)Approximability.* May 1997.

X-98-1 Gianluca Battaglini. *Analysis of Manufacturing Yield Evaluation of VLSI/WSI Systems: Methods and Methodologies.* April 1998.

X-98-2 Piergiorgio Bertoli. *Using OMRS in practice: a case study with Acl-2.* April 1998.

X-98-3 Chiara Ghidini. *A semantics for contextual reasoning: theory and two relevant applications.* April 1998.

X-98-4 Roberto Giaccio. *Visiting complex structures.* April 1998.

X-98-5 Giampaolo Greco. *Dimension and structure in Combinatorics.* April 1998.

X-98-6 Paolo Liberatore. *Compilation of intractable problems and its application to artificial intelligence.* April 1998.

X-98-7 Fabio Massacci. *Efficient approximate tableaux and an application to computer security.* April 1998.

X-98-8 Chiara Petrioli. *Energy-Conserving Protocols for Wireless Communications.* April 1998.

X-98-9 Giulio Balestreri. *An Algebraic Semantics for the Shared Spaces Coordination Languages.* April 1999.

XI-99-1 Luca Becchetti. *Efficient Resource Management in High Bandwidth Networks.* April 1999.

XI-99-2 Nicola Cancedda. *Text Generation from Message Understanding Conference Templates.* April 1999.

XI-99-3 Luca Iocchi. *Design and Development of Cognitive Robots.* April 1999.

XI-99-4 Francesco Quaglia. *Consistent checkpointing in distributed computations: theoretical results and protocols.* April 1999.

XI-99-5 Milton Romero. *Disparity/Motion Estimation For Stereoscopic Video Processing.* April 1999.

XI-99-6 Massimiliano Parlione. *Remote Class Inheritance.* April 2000.

XII-00-1 Marco Daniele. *Advances in Planning as Model Checking.* April 2000.

XII-00-2 Walter Didimo. *Flow Techniques and Optimal Drawings of Graphs.* April 2000.

XII-00-3 Leonardo Tininini. *Querying Aggregate Data.* April 2000.

XII-00-4 Enver Sangineto. *Classificazione Automatica d'Immagini Tramite Astrazione Geometrica.* April 2001.

XIII-01-1 Camil Demetrescu. *Fully Dynamic Algorithms for Path Problems on Directed Graphs.* April 2001.

XIII-01-2 Giovanna Melideo. *Tracking Causality in Distributed Computations.* April 2001.

XIII-01-3 Maurizio Patrignani. *Visualization of Large Graphs.* April 2001.

XIII-01-4 Maurizio Pizzonia. *Engineering of Graph Drawing Algorithms for Applications.* April 2001.