Efficient Causality-Tracking Timestamping

Jean-Michel Hélary, Michel Raynal, Giovanna Melideo, and Roberto Baldoni

Abstract—Vector clocks are the appropriate mechanism used to track causality among the events produced by a distributed computation. Traditional implementations of vector clocks require application messages to piggyback a vector of n integers (where n is the number of processes). This paper investigates the tracking of the causality relation on a subset of events (namely, the events that are defined as "relevant" from the application point of view) in a context where communication channels are not required to be FIFO, and where there is no a priori information on the connectivity of the communication graph or the communication pattern. More specifically, the paper proposes a suite of simple and efficient implementations of vector clocks that address the *reduction of the size of message timestamps*, i.e., they do their best to have message timestamps whose size is less than n. The relevance of such a suite of protocols is twofold. From a practical side, it constitutes the core of an *adaptive timestamping software layer* that can used by underlying applications. From a theoretical side, it provides a comprehensive view that helps better understand distributed causality-tracking mechanisms.

Index Terms—Asynchronous distributed computation, causality, message-passing, timestamp, vector clock.

1 Introduction

A distributed computation consists of a set of processes that cooperate to achieve a common goal. A main characteristic of these computations lies in the fact that the processes do not share a common global memory, and communicate only by exchanging messages over a communication network. Moreover, message transfer delays are finite but unpredictable. This computation model defines what is known as the *asynchronous distributed system model*. It is particularly important as it includes systems that span large geographic areas, and systems that are subject to unpredictable loads. Consequently, the concepts, tools, and mechanisms developed for asynchronous distributed systems reveal to be both important and general.

Causality is a key concept to understand and master the behavior of asynchronous distributed systems. More precisely, given two events e and f of a distributed computation, a crucial problem that has to be solved in a lot of distributed applications is to know whether they are causally related, i.e., if the occurrence of one of them is a consequence of the occurrence of the other. Events that are not causally dependent are said to be concurrent. Hence, a fundamental distributed computing problem consists in the online tracking of dependencies on events. By associating with each event a vector timestamp of size n (where n is the total number of processes), a causality-tracking protocol makes possible to safely decide whether two events are causally related or not by the only comparison of their timestamps. It has been shown that the size n for the event timestamps is a necessary requirement [3]. Moreover, causality-tracking protocols require that each application message piggybacks control information. The first causality-tracking protocol (denoted here $\mathcal{P}0$ and called "canonical implementation") requires that each application message carries an integer vector of size n [4], [13].

In many applications, detecting causal dependencies (or concurrency) on all events is not desirable. More precisely, one is usually interested only in a subset of events called the *relevant* events (e.g., distributed debugging and rollback-recovery just to name a few). This paper considers the tracking of the causal relationship on the relevant events of asynchronous message-passing distributed computations, and investigates the problem of reducing the size of the control information piggybacked by application messages. It is important to note that tracking causality on only a subset of the events of a computation is a more subtle issue than tracking causality on all the events. This is due to the fact that the nonrelevant events can establish causal dependencies on relevant events. From a "causality-tracking" perspective, 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 aspect of causality-tracking is addressed in Sections 5 and 7). 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.

In order to reduce the number of entries of message timestamps, a condition is first introduced. It expresses which part of control information can be omitted from a message timestamp without preventing a correct causality-tracking event timestamping. This condition is abstract in the sense that it does not rely on particular data structures that could be directly accessed by processes. Several protocols are proposed: Each of them is 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 n. The case where the channels are FIFO is first examined. In this context,

- *J.-M. Hélary and M. Raynal are with IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France. E-mail: {helary, raynal}@irisa.fr.*
- *G. Melideo is with the Computer Science Department, University of L'Aquila, via Vetoio, loc. Coppito, 67070 L'Aquila, Italy. E-mail: melideo@di.univaq.it.*
- *R. Baldoni is with DIS, Università "La Sapienza," Via Salaria 13, 00198 Roma, Italy. E-mail: baldoni@dis.uiroma1.it.*

Manuscript received 22 Sept. 2000; revised 14 Feb. 2001; accepted 20 Mar. 2001.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 112895.

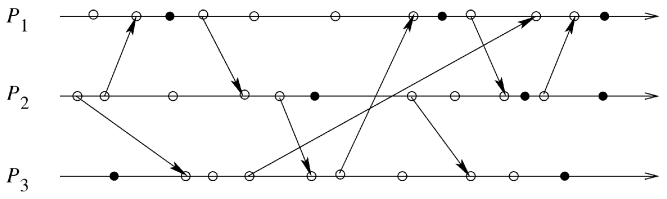


Fig. 1. A distributed computation.

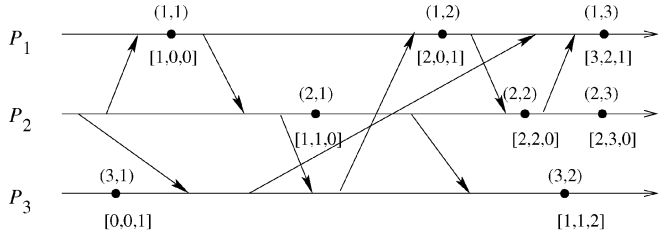


Fig. 2. Relevant events and their timestamps.

the well-known timestamping protocol introduced by Singhal and Kshemkalyani [17] is extended to suit to the case where only a subset of the primitive events are relevant. Then, the general case (where channels are not necessarily FIFO) is considered and two protocols, namely, $\mathcal{P}1$ and $\mathcal{P}2$, are proposed. From a practical point of view, $\mathcal{P}0$, $\mathcal{P}1$, and $\mathcal{P}2$ form a suite of protocols well tailored 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. When a message m is sent by an application, the timestamping software layer 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.

Practically, many applications and toolkits can benefit from this timestamping software layer by reducing their message traffic. For example, a replicated storage system [19], that represents the infrastructure of collaborative applications, uses vector clocks to detect 1) update (write-write) conflicts (i.e., concurrent updates) of replicated data and 2) when an update operation has been executed at all sites. The latter point is also of primary importance when considering group toolkit such as ISIS [2] to implement message stability protocols [1].

Last, but not least, in addition to its practical relevance, the proposed suite of protocols provides a comprehensive view that helps better understand the rules that master causality-tracking in distributed systems.

The paper is composed of eight sections. Sections 2 and 3 introduce the computation model and vector clocks, respectively. Section 4 introduces the abstract condition and proves that it is necessary and sufficient. Then, Section 5 considers the particular case where the channels are FIFO and presents the extended Singhal-Kshemkalyani's protocol E_SK. The two protocols $\mathcal{P}1, \mathcal{P}2$ are presented in Section 6. Assuming FIFO channels, Section 7 compares them to E_SK. Finally, Section 8 concludes the paper.

2 Computation Model

2.1 Distributed Computation

A distributed program is made up of sequential local programs which communicate and synchronize only by exchanging messages. A distributed computation describes the execution of a distributed program. The execution of a local program gives rise to a sequential process. Let $\{P_1, P_2, \ldots, P_n\}$ be the finite set of sequential processes of the distributed computation. Each ordered pair of communicating processes (P_i, P_j) is connected by a reliable channel c_{ij} through which P_i can send messages to P_j. We assume a process does not send messages to itself. Message transmission delays are finite but unpredictable. Moreover, channels are not necessarily FIFO. Process speeds are positive but arbitrary. In other words, the underlying computation model is asynchronous.

The local program associated with P_i can include send, receive, and internal statements. The execution of such a statement produces a corresponding internal/send/receive event. These events are called *primitive events*. Let H be the set of events produced by a distributed computation, and e_i^x be the xth event produced by process P_i. This set is structured as a partial order by Lamport's *happened before* relation [10], denoted $\overset{hb}{\rightarrow}$ and defined as follows:

$$e_i^x \overset{hb}{\rightarrow} e_j^y \Leftrightarrow \begin{array}{l}(i{=}j \wedge x{+}1{=}y)(\text{local precedence}) \vee \\ (\exists m{:}e_i^x{=}send(m) \wedge e_j^y{=}receive(m))(\text{message precedence}) \vee \\ (\exists e_k^z{:}e_i^x \overset{hb}{\rightarrow} e_k^z \wedge e_k^z \overset{hb}{\rightarrow} e_j^y)(\text{transitive closure}).\end{array}$$

Two distinct events e and f are *concurrent* (or *causally independent*) if $\neg(e \overset{hb}{\rightarrow} f) \wedge \neg(f \overset{hb}{\rightarrow} e)$. The *causal past* of an event e is the (partially ordered) set of events f such that $f \overset{hb}{\rightarrow} e$.

The partial order $\widehat{H} = (H, \overset{hb}{\rightarrow})$ constitutes a formal model of the distributed computation it is associated with. Fig. 1 depicts a distributed computation using the classical space-time diagram. Its primitive events are denoted by white and black dots.

2.2 Relevant Events

At some abstraction level only some events of a distributed computation are relevant [5], [6], [12] (those events are sometimes 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 paper, we are not concerned by this decision. As an example, in Fig. 1 only the black events are relevant.

Let $R \subseteq H$ be the set of relevant events. Let $\rightarrow$ be the relation on R defined in the following way:

$$\forall (e, f) \in R \times R : (e \rightarrow f) \Leftrightarrow (e \overset{hb}{\rightarrow} f).$$

The poset $(R, \rightarrow)$ constitutes an abstraction of the distributed computation, namely, a *communication and relevant event pattern*. In the following, we consider a distributed computation at such an abstraction level. Moreover, without loss of generality, we consider that a relevant event is not a communication event (if a communication event has to be observed, a relevant event can be generated just after the corresponding communication event occurred). Fig. 2 depicts an abstraction of the computation described in Fig. 1, where only the shaded events are relevant. Each relevant event is identified by a pair (process id, sequence number).

3 Vector Clocks

3.1 Definition

Vector clocks have been empirically used as an ad hoc device to solve specific problems before being captured and defined as a concept. For example, they were "implicitly" used to detect mutual inconsistencies of copies of a replicated data [15], to detect obsolete data [11], to prevent drift among logical scalar clocks [16], and to track dependencies between local checkpoints [18]. But, as a first class concept with the associated theory, vector clocks have been introduced in 1988, simultaneously and independently, by Fidge [4] and Mattern [13].

A vector clock system is a 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). More precisely, each process P_i has a vector of integers $VC_i[1..n]$ such that $VC_i[j]$ is the number of (relevant) events produced by P_j that belong to the current causal past of P_i. Note that $VC_i[i]$ counts the number of (relevant) events produced so far by P_i. 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. Fig. 2 associates its vector timestamp with each relevant event of the described distributed computation.

Let $e.VC$ and $f.VC$ be the vector timestamps associated with two distinct (relevant) events e and f, respectively. The following property is the fundamental property associated with vector clocks [4], [7], [13]:

$$\forall (e,f) \in R \times R : ((e \rightarrow f) \Leftrightarrow ((\forall k : e.VC[k] \leq f.VC[k]) \ \wedge \ (\exists k : e.VC[k] < f.VC[k]))).$$

where,

$$(\forall k : e.VC[k] \leq f.VC[k]) \ \wedge \ (\exists k : e.VC[k] < f.VC[k])$$

is usually denoted $e.VC < f.VC$. Let P_i be the process that has produced e. This additional information allows one to simplify the previous relation which reduces to [4], [13]:

$$\forall (e,f) \in R \times R : (e \rightarrow f) \Leftrightarrow (e.VC[i] \leq f.VC[i]).$$

3.2 Canonical Implementation $\mathcal{P}0$

The traditional implementation of a vector clock system is as follows. Each process P_i manages its vector clock $VC_i[1..n]$ according to the following rules:

- [R0] $VC_i[1..n]$ is initialized to $[0, \ldots, 0]$.
- [R1] Each time it produces a relevant event e:
  - P_i increments its vector clock entry $VC_i[i]$ ($VC_i[i] := VC_i[i] + 1$) to indicate it has produced one more (relevant) 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: $\forall x : VC_i[x] := max(VC_i[x], m.VC[x])$ (this operation is abbreviated as $VC_i := max(VC_i, m.VC)$).

3.3 Discussion

The major drawback of the canonical implementation $\mathcal{P}0$ 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 [3]. Nevertheless, the previous protocol can be easily improved in the following way. When P_i sends a message to P_j, it may send it only the entries that have been modified since its last sending to this process P_j. This improvement, proposed in [17], requires FIFO channels and does not consider the notion of relevant event (it implicitly assumes that all events are relevant, see Section 5.1). It has a small local memory overhead, namely, a process has to manage only two additional arrays of size n. Another improvement is described in [14]. It exploits information on the connectivity of the communication graph, and works when one is interested only in the causality relationship between messages that are sent to the same process.

Some applications need only an *approximation* of the causality relation $\rightarrow$. A relation $\rightarrow_a$ is an approximation of $\rightarrow$ if it is a partial order and if

$$\forall (e,f) \in R \times R : (e \rightarrow f) \Rightarrow (e \rightarrow_a f).$$

This means that some events are ordered by $\rightarrow_a$ while they are actually not related by $\rightarrow$. It is shown in [20] how to implement such approximations of $\rightarrow$. That approach uses vectors whose size is bounded by a constant k with $1 \leq k \leq n$ (the case $k = 1$ actually gives rise to Lamport scalar clocks [10], while the case $k = n$ corresponds to vector clocks). It is important to notice that, as these vectors aim only to provide an approximation of the causality relation, they do not allow an exact reconstruction of it (even offline).

3.4 Notations

The following notations are used in the rest of the paper. Let e be an event of H.

- 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.
- $pred(e)$ denotes the event immediately preceding e on the same process (if it exists).
- For every process P_i, H_i (resp. R_i) denotes the sequence of events (resp. relevant events) produced by P_i.

For example, if m is message sent by P_i to P_j, then $pred(receive(m)).VC_j[k]$ denotes the value of the kth 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 the processing of $m.VC$ by P_j. Note that $send(m).VC_i[k]$ denotes the value of the kth 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).

4 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.

4.1 To Transmit or Not to Transmit Control Information

Let us consider the rule [R3]. It shows 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: there is no update of $VC_j[k]$ when $VC_j[k] \geq m.VC[k]$. 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.

Definition 1.

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

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 the rule [R2] in the following way: the pair $(k, VC_i[k])$ is transmitted with m if and only if $\neg K(m,k)$. The rule [R3] should be appropriately modified to consider only the pairs carried by m.

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 $(k, m.VC[k])$.

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

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

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. That is

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

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

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

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. □

4.3 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 $send(m).VC_i[k]$ (it is the current value of $VC_i[k]$). As far as the value $pred(receive(m)).VC_j[k]$ is concerned, there are two cases:

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

Hence, 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 $K(m,k)$. Let $K'(m,k)$ be an approximation of $K(m,k)$, such that $K'(m,k)$ can be evaluated by P_i when it sends a message m. This approximation is used in the following way: Each time $\neg K'(m,k)$, 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 3. The two sections that follow present conditions that are better approximations of K. Section 5 first considers the case where channels are FIFO (condition K_{E_SK}). Then, Section 6 considers the general case where channels are not required to be FIFO (condition K_M).

5 The Case of FIFO Channels

5.1 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 [17]. This improvement does not consider the notion of relevant events; this means that it implicitly assumes that all the events are relevant.

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. The reduction of the communication overhead is obtained at the expense of an extra storage space at each process. More precisely, each process P_i keeps two additional vectors of size n:

- $LU_i[1..n]$ (Last Update), such that $LU_i[k] = x$ means that the last event of P_i upon which $VC_i[k]$ has been updated occurred when $VC_i[i] = x$.
- $LS_i[1..n]$ (Last Sent), such that $LS_i[k] = x$ means that the last send event to P_k occurred when $VC_i[i] = x$.

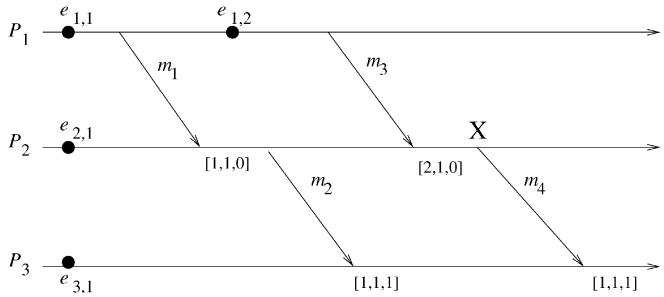


Fig. 3. Singhal-Kshemkalyani's technique is not appropriate when communication events are not relevant.

The improvement consists in the following modification of the rule [R2]: When P_i sends a message m to P_j, it only needs to send those entries $VC_i[k]$ such that $LS_i[j] < LU_i[k]$.

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). The following counter-example, shown in Fig. 3, shows that $VC_3[1]$ is not correctly updated when P_3 receives the message m_4 (on the figure, the relevant events are shown by dots). When P_2 sends m_4 to P_3 (this nonrelevant communication event is denoted X in the figure), we have $LU_2[1] = 1, LS_2[3] = 1$, so the pair $(1, VC_1[1]) = (1, 2)$ is not transmitted with m_4. So, on the receipt of m_4, no update of $VC_3[1]$ occurs and the dependence $e_{1,2} \overset{hb}{\rightarrow} receive(m_4)$ is not taken into account.

However, if LU and LS count *all* events (while the vector clock VC counts only the relevant events), the technique can be adapted, as shown in the next section.

5.2 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. It is then shown that this condition is correct,[1] i.e., 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. Each process P_i manages the following set of local variables:

- an integer value X_i counting nonrelevant 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 receipt event (i.e., a nonrelevant 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 $LU_i[k].f$ (respectively, $LU_i[k].s$) the first (respectively, 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 $LS_i[k].f$ (respectively, $LS_i[k].s$) the first (respectively, second) component of the pair $LS_i[k]$.

1. Note that the basic protocol described in [17] is presented without proof.

E_SK protocol. The concrete condition K_{E_SK} is defined as follows:

Definition 2. *Let m be a message sent by P_i to P_j. Then,*

$$K_{E\_SK}(m,k) \equiv (LS_i[j].f > LU_i[k].f) \vee ((LS_i[j].f = LU_i[k].f) \wedge (LS_i[j].s \leq LU_i[k].s))$$

This property is denoted $LS_i[j] \geq LU_i[k]$. By an obvious application of Boolean calculus, we have:

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

(denoted $LS_i[j] < LU_i[k]$).

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

- [E-SK0] (Initialization):
  - $VC_i[1..n]$ is initialized to $[0, \ldots, 0]$,
  - X_i is initialized to 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 nonrelevant event (internal/send/receive), it increments its local counter X_i before doing any other action (see the rules [E-SK2] and [E-SK3]).
- [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 [17].

5.3 Correctness of the Condition K_{E_SK}

Theorem 2. *$\forall i, \forall j, \forall m$ sent from P_i to P_j,*

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

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 $send(m_1).LS_i[j] = (0,0)$. If $K_{E\_SK}(m_1, k)$ holds, necessarily

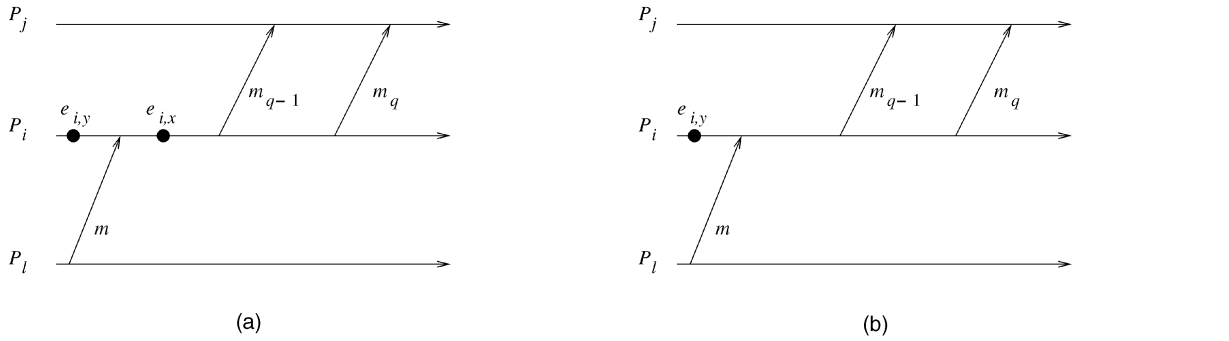


Fig. 4. Proof of Theorem 2.

$$send(m_1).LU_i[k] = (0,0).$$

In particular, $send(m_1).VC_i[k] = 0$. Thus,

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

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

$$(x, \alpha) = (pred(send(m_q)).LS_i[j]$$

and

$$(y, \beta) = pred(send(m_q)).LU_i[k].$$

If $K_{E\_SK}(m_q, k)$ holds, we have either $(x > y)$ (Fig. 4a) or $(x = y) \wedge (\alpha \geq \beta)$ (Fig. 4b). Note that in the second case, $\alpha > \beta$ because a receipt event and a send event are distinct. 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

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

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. □

6 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, without additional control messages), in a context where 1) channels are not necessarily FIFO and 2) there is no a priori information on the communication graph connectivity or the communication pattern.

6.1 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 (introduced for the first time in [9]). This Boolean matrix is managed to satisfy the following property: "For each message m sent by P_i to P_j,

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

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 m: no update of M_i occurs.
- [M3] When it receives a message m from P_j, P_i executes the following updates:

$\forall (k, m.VC[k]) \in m.VC :$
case = $VC_i[k] < m.VC[k]$ **then**

$\forall \ell \neq i, j, k : M_i[\ell, k] := 0; M_i[j, k] := 1;$
$VC_i[k] = m.VC[k]$ **then**
$M_i[j, k] := 1$
$VC_i[k] > m.VC[k]$ **then** *skip*
endcase

6.2 Correctness

In this section, we first show (Lemma 1) that the previous implementation of the matrix M_i (rules [M1]-[M3]), combined with the vector clock rules [R0]-[R3], provides its correct meaning to M_i, namely,

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

It is then shown that M_i can be used to implement the abstract condition $K(m, k)$ (Theorem 3).

Lemma 1. *Let $\mathcal{IM}(e, j, k)$ denote the following property, where $e \in H_i$:*

$$[(e.M_i[j,k] = 1) \Rightarrow (j = k) \vee (i = j) \vee (e.VC_i[k] = 0) \vee (\exists m' \text{ from } P_j \text{ to } P_i \text{ s.t.} ((receive(m') \stackrel{hb}{\rightarrow} e) \vee (receive(m') = e)) \wedge (send(m'). VC_j[k] = m'.VC[k] = e.VC_i[k]))].$$

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

Proof. The proof is by induction on the poset $\widehat{H}$. Since $\mathcal{IM}(e, j, k)$ trivially holds: 1) when $j = k$, 2) when $i = j$, and 3) for every event e such that $e.M_i[j, k] = 0$, we consider only the cases $j \neq k \wedge i \neq j$, and only the events e such that $e.M_i[j, k] = 1$. Moreover, if for an event $e \in H_i$ and a pair (j, k) we have

$$(pred(e).M_i[j,k] = e.M_i[j,k]) \wedge pred(e).VC_i[k] = e.VC_i[k],$$

then, obviously, $\mathcal{IM}(pred(e), j, k) \Rightarrow \mathcal{IM}(e, j, k)$ (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 (recall that $j \neq k \wedge i \neq j$):

- e is neither a relevant nor a receipt event. Then, we have $e.VC_i[k] = 0$.
- e is a relevant event. Then we have $\forall j, \forall k \neq i : e.M_i[j, k] = 1$ 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 $e \in H_i$. 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 $(pred(e).M_i[x, k] = 1) \wedge (k, m.VC[k]) \notin m$.
  4. All the entries (j, k) of M_i where k is such that $pred(e).VC_i[k] \leq m.VC[k]$.
  5. All the entries (x, k) of M_i such that $(pred(e).VC_i[k] \geq m.VC[k]) \wedge (pred(e).M_i[x, k] = 1)$.

  - 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 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 1, 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

$$send(m).VC_i[j] \leq pred(receive(m)).VC_j[j].$$

2. $send(m).VC_i[k] = 0$. In that case,

$$send(m).VC_i[k] = 0 \leq pred(receive(m)).VC_j[k].$$

3. $\exists m'$ from P_j to P_i s.t. $receive(m') \stackrel{hb}{\rightarrow} e \wedge (send(m').VC_j[k] = m'.VC[k] = send(m).VC_i[k])$.

This implies $send(m') \stackrel{hb}{\rightarrow} pred(receive(m))$ (locally on P_j) or $send(m') = pred(receive(m))$ and, thus, in both cases,

$$send(m').VC_j[k] \leq pred(receive(m)).VC_j[k].$$

Hence,

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

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

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

in each case. Hence, $K_M(m,k) \Rightarrow K(m,k)$. □

Two protocols based on K_M are presented. The protocol $\mathcal{P}1$ directly uses M_i to reduce the number of vector clock entries that have to be transmitted. The protocol $\mathcal{P}2$ shows that this number can still be reduced if we allow a message to carry Boolean vectors (namely, a Boolean vector per vector clock entry that has to be transmitted). It follows that the protocol $\mathcal{P}2$ exhibits a tradeoff in the control information piggybacked by messages (integers versus Boolean arrays).

6.3 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: $m.VC = \{(k, VC_i[k]) | M_i[j,k] = 0\}$.
- [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 kth 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 [R1]). 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 $\mathcal{P}1$ 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, even if this value is already known to P_j.

6.4 Protocol $\mathcal{P}2$

The protocol $\mathcal{P}2$ aims at increasing the number of entries of M_i that are set to 1 and, consequently, decreasing 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 an n-Boolean array):

$$\{(k, VC_i[k], M_i[*,k]) | M_i[j,k] = 0\}.$$

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

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

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 $\mathcal{P}2$ (left to the reader) is similar to the previous one.

In the rest of the paper, $T_{1\rightarrow 2}$ denotes the modification that transforms protocol $\mathcal{P}1$ into protocol $\mathcal{P}2$.

6.5 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 required to encode a sequence number (identifying a relevant event on a process), and

$|m.VC|$ denote the number of pairs (or triples) of $m.VC$. The protocol $\mathcal{P}0$ requires to transmit $n * s$ bits of control information, whereas $\mathcal{P}1$ requires $|m.VC| * (s + \log_2 n)$ bits and $\mathcal{P}2$ requires $|m.VC'| * (n + s + \log_2 n)$ bits (where $m.VC$ and $m.VC'$ are the timestamp associated with m by $\mathcal{P}1$ and $\mathcal{P}2$, respectively. $|m.X|$ denotes the number of elements of the set $m.X$). It can be easily seen that, for a message m:

1. $\mathcal{P}1$ is better than $\mathcal{P}0$ if
$$|m.VC| < \frac{n * s}{(s + \log_2 n)}.$$
2. Protocol $\mathcal{P}2$ is better than $\mathcal{P}0$ and $\mathcal{P}1$ if
$$|m.VC'| < \frac{\min(n * s, |m.VC| * (s + \log_2 n))}{(n + s + \log_2 n)}.$$

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 [MR0] and [MR1] are those implementing protocol $\mathcal{P}1$, whereas the other are the following ones:

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

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

7 Coming Back to FIFO Channels

This section shows that, if the channels are FIFO, the previous protocols can be improved and this improvement is more efficient than the extended Singhal-Kshemkalyani's protocol presented in Section 5.2.

7.1 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 $M_i[j,k] := 1$. In fact, consider another message m' sent by P_i to P_j after m. Due to FIFO assumption, m' will be received by P_j after m. If
$$send(m').VC_i[k] = send(m).VC_i[k],$$
the value $send(m').VC_i[k]$ has not to be transmitted by m' because P_j has already "learnt" this value upon receiving m (or earlier). Note that, in that case, $send(m').M_i[j,k] = 1$ because $M_i[j,k]$, set to 1 upon the sending of m, has not meanwhile been reset to 0 ($VC_i[k]$ has not been updated). This also explains why this enhancement is restricted to systems with FIFO communications: If m' could overpass m, then the value $send(m).VC_i[k]$ transmitted by m could be not yet known by P_j when it receives m'.

Consequently, it appears that the FIFO property of channels can reduce the number of pairs piggybacked by a message. Note, in particular, that with this enhancement, a 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.

7.2 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 2) and $\neg(K_M \Rightarrow K_{E\_SK})$ (Lemma 3).

Lemma 2. *$\forall i$, $\forall j$, $\forall m$ 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 $k \neq i$.

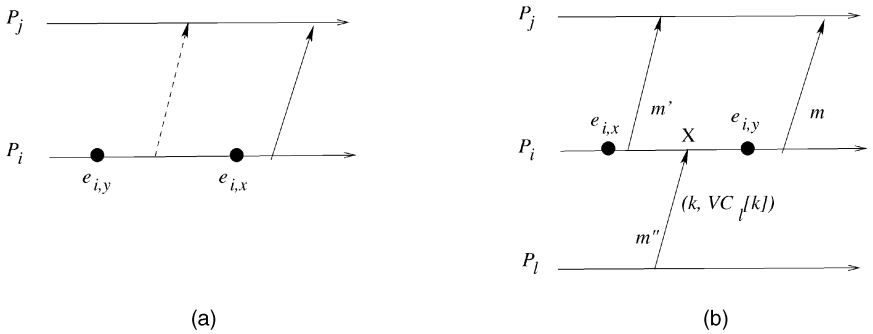


Fig. 5. Proof of Lemma 2.

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 Fig. 5a). 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 $send(m).LU_i[i] = (x, \alpha)$ 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 $m.''VC[k] > pred(receive(m'')).VC_i[k]$ (see Fig. 5b). 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 3. *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 Fig. 6. 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 $pred(receive(m')).VC_i[k] = 0$ 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

$$pred(send(m'')).LU_i[k] = (0,0) < pred(send(m'')).LS_i[k] = (0,1),$$

hence $\neg K_{E\_SK}(m'', k)$. □

Note that this counter-example remains true even in the case where all events are relevant (i.e., all internal events are relevant and one relevant event is generated just after each communication event). In that case, when P_i is about to send m'': $LS_i[k] = (0,0) < LU_i[k] = (2,0)$ and, thus, $\neg K_{E\_SK}(m'', k)$; on the other hand, $M_i[j,k]$ has been set to 1 upon the receipt of m' and has not been reset to 0 upon the relevant event just following this receipt. Thus, $K_M(m'', k)$ holds.

Theorem 4. *$\mathcal{P}1 \prec E\text{-}SK$.*

Proof. The proof directly follows from the Lemmas 2 and 3.□

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.

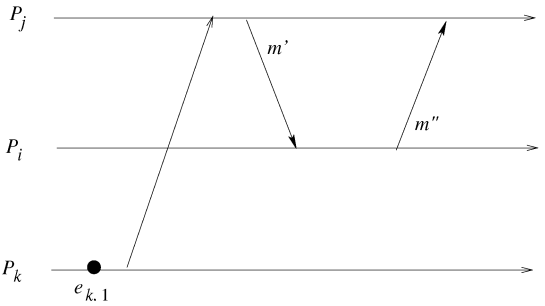


Fig. 6. Proof of Lemma 3.

8 Conclusion

This paper has addressed the tracking of the causality relation on the relevant events produced by asynchronous distributed message-passing computations. A suite of simple and efficient causality-tracking protocols has been presented. These protocols 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). They require no particular assumptions on channel behaviors, (such as FIFO), the connectivity of the communication graph or the communication pattern. From a practical point of view, they provide a suite that can be used as the core of an adaptive timestamping software layer that allows a distributed application to timestamp its messages in an efficient way. Replicated storage systems [19], group toolkits [2], and distributed debugging tools [8] are examples of applications that enjoy a reduction of message traffic by using that timestamping service.

References

[1] R. Baldoni and M. Raynal, "Fundamentals of Distributed Computing: A Practical Tour of Vector Clocks," *IEEE Distributed Systems Online,* vol. 3, no. 2, 2002.

[2] K. Birman, A. Schiper, and P. Stephenson, "Lightweight Causal Order and Atomic Group Multicast," *ACM Trans. Computer Systems,* vol. 9, no. 3, pp. 282-314, 1991.

[3] B. Charron-Bost, "Concerning the Size of Logical Clocks in Distributed Systems," *Information Processing Letters,* vol. 39, pp. 11-16, 1991.

[4] C.J. Fidge, "Timestamps in Message-Passing Systems that Preserve Partial Ordering," *Proc. 11th Australian Computing Conf.,* pp. 56-66, 1988.

[5] E. Fromentin, C. Jard, G.V. Jourdan, and M. Raynal, "On-the-Fly Analysis of Distributed Computations," *Information Processing Letters,* vol. 54, pp. 267-274, 1995.

[6] E. Fromentin and M. Raynal, "Shared Global States in Distributed Computations," *J. Computer and System Sciences,* vol. 55, no. 3, pp. 522-528, 1997.

[7] V.K. Garg, *Principles of Distributed Systems.* Kluwer Academic Press, 1996.

[8] V.K. Garg, "Observation and Control for Debugging Distributed Computations," *Proc. Third Int'l Workshop Automatic Debugging,* pp 1-12, 1997.

[9] J.M. Hélary, G. Melideo, and M. Raynal, "Tracking Causality in Distributed Systems: A Suite of Efficient Protocols," *Proc. Seventh Int'l Colloquium on Structural Information and Communication Complexity (SIROCCO),* pp. 181-195, June 2000.

[10] L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System," *Comm. ACM,* vol. 21, no. 7, pp. 558-565, 1978.

[11] B. Liskov and R. Ladin, "Highly Available Distributed Services and Fault-Tolerant Distributed Garbage Collection," *Proc. Fifth ACM Symp. Principles of Distributed Computing,* pp. 29-39, 1986.

[12] K. Marzullo and L. Sabel, "Efficient Detection of a Class of Stable Properties," *Distributed Computing,* vol. 8, no. 2, pp. 81-91, 1994.

[13] F. Mattern, "Virtual Time and Global States of Distributed Systems," *Proc. Int'l Conf. Parallel and Distributed Algorithms,* Cosnard, Quinton, Raynal, and Robert, eds., pp. 215-226, 1988.

[14] S. Meldal, S. Sankar, and J. Vera, "Exploiting Locality in Maintaining Potential Causality," *Proc. 10th ACM Symp. Principles of Distributed Computing,* pp. 231-239, 1991.

[15] D.S. Parker et al., "Detection of Mutual Inconsistency in Distributed Systems," *IEEE Trans. Software Eng.,* vol. 9, no. 3, pp. 240-246, 1983.

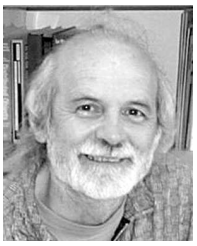
[16] M. Raynal, "A Distributed Algorithm to Prevent Mutual Drift Between n Logical Clocks," *Information Processing Letters,* vol. 24, pp. 199-202, 1987.

[17] M. Singhal and A. Kshemkalyani, "An Efficient Implementation of Vector Clocks," *Information Processing Letters,* vol. 43, pp. 47-52, 1992.

[18] R.E. Strom and S. Yemini, "Optimistic Recovery in Distributed Systems," *ACM Trans. Computer Systems,* vol. 3, no. 3, pp. 204-226, 1985.

[19] D.B. Terry, M.M. Theimer, K. Petersen, A.J. Demers, M.J. Spreitzer, and C.H. Hauser, "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage Systems," *Proc. ACM Symp. Operating Systems Principles,* pp. 172-183, 1995.

[20] F.J. Torres-Rojas and M. Ahamad, "Plausible Clocks: Constant Size Logical Clocks for Distributed Systems," *Distributed Computing,* vol. 12, no. 4, pp. 179-196, 1999.



Jean-Michel Hélary received the Docteur troisième cycle degree in applied mathemetics (numerical analysis of partial differential equations) from the University of Paris, in 1968, and the Habilitation Doctor degree from the University of Rennes in 1988. He is currently professor of computer science at the University of Rennes, France. His research interests are distributed algorithms, protocols, programming languages, graph algorithms, and parallelism. He is a member of the INRIA research team Algorithmes Distribués et Protocoles, led by Michel Raynal. In this team, he is specifically interested by the methodological aspects of distributed algorithms design. Professeur Hélary has published more than 40 scientific publications in journals, reviews, and conferences. He is the coauthor, with Michel Raynal, of a book devoted to Synchronization of Distributed Algorithms and Systems, (John Wiley, 1990). He has been member of the program comitee and co-chairman of the Distributed Algorithms track of the 14th IEEE International Conference on Distributed Computing Systems (Poznan, 1994), cochairman of the Ninth International Workshop on Distributed Algorithms (Le Mont Saint-Michel, France, September 1995), and Local Arrangement cochairman of the Second IEEE International Symposium on object-oriented real-time distributed computing (May 1999, Saint Malo, France).

Michel Raynal received the "doctorat d'Etat" in computer science from the University of Rennes, France, in 1981. In 1981, he moved to Brest (France), where he founded the Computer Science Department in a telecommunications engineer school (ENST). In 1984, he moved to the University of Rennes, where he has been a professor of computer science. At IRISA (CNRS-INRIA-University joint computing research laboratory located in Rennes), he is the founder and the leader of the research group "Algorithmes distribus et protocoles"(Distributed Algorithms and Protocols). His research interests include distributed algorithms, operating systems, computing systems, protocols, and dependability. His main interest lies in the fundamental principles underlying the design and the construction of distributed systems. He has been the principal investigator of a number of research grants in these areas (grants from French private companies, France Telecom, CNRS, NFS-INRIA, and ESPRIT BRA). Professor Raynal has been invited by many universities to give lectures about operating systems and distributed algorithms in Europe, South and North America, Asia, and Africa. He is serving as an editor for two international journals. He has published more than 70 papers in journals and more than 140 papers in conferences. He has has written seven books devoted to parallelism, distributed algorithms, and systems. Among them: *Distributed Computations and Networks* (MIT Press, 1988) and *Synchronization and Control of Distributed Programs*, coauthored with Jean-Michel Hélary, (Wiley, 1990). Dr. Raynal has been invited to serve in program committees for more than 50 international conferences. In 1989 and 1995, he served as the program chair of the WDAG/DISC Symposium on Distributed Algorithms. In 1995, he was the program chair of the IEEE Conference on Future Trends of Distributed Computing Systems (FT DCS'95). In 1999, he served as the general chair of the IEEE Conference on Object-Oriented Real-Time Distributed Computing (ISORC'99) that was held in Saint-Malo (France). He will be the program co-chair of IEEE International Conference on Distributed Computing Systems (ICDCS '02) that will be held in Vienna (Austria), and the conference co-chair of ISORC'02 that will be held in Washington DC.

Giovanna Melideo received the Laurea degree in computer science from the University of L'Aquila, Italy, in 1997 and the PhD degree in computer engineering from the University of Rome "La Sapienza." She is currently an assistant professor in the Computer Science Department at the University of L'Aquila. Her current research interests include distributed algorithms and protocols.

Roberto Baldoni received the laurea degree in electronic engineering in 1990 and the PhD degree in computer science in 1994 from the University of Rome "La Sapienza." From 1994 to 1995, he did postdoctoral work at IRISA/INRIA (France) in the ADP group. In the summer of 1996, he was a visiting assistant professor in the Department of Computer Science of Cornell University. He has published more than 70 scientific papers in the fields of distributed computing, dependable middleware, and communication protocols. His research interests span from theory to practice trying to bring latest research results into distributed applications. He regularly serves as a referee for many international journals. He was on the organizing and program committees of many international conferences and workshops such as WDAG, ICDCS, SRDS, EUROPAR, ISORC, DOA and CoopIS. He was invited to chair the program committee of the "Distributed Algorithms" track of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS-99) and the Sixth IEEE International Workshop on Object Oriented Real-time Dependable Systems (WORDS'01). He is also the tutorial chair of the Third International Symposium on Distributed Objects and Applications (DOA'01). Currently, he is an associate professor at the school of engineering of the University of Rome "La Sapienza."