Tracking Causality in Distributed Systems: A Suite of Efficient Protocols

JEAN-MICHEL HÉLARY
IRISA, University of Rennes 1, France

GIOVANNA MELIDEO
DIS, University of Rome and University of L'Aquila, Italy

MICHEL RAYNAL
IRISA, University of Rennes 1, France

Abstract

Vector clocks are the appropriate mechanism 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 considers 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). It first proposes simple and efficient implementations of vector clocks where the size of message timestamps can be less than n, 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. Then, it presents a protocol that provides a correct timestamping of the relevant events in presence of the following constraint: a message timestamp can piggyback at most b event identifiers (where b is a predefined constant, $1 \leq b \leq n$). To ensure this constraint, processes can be forced to produce additional "null" relevant events. Finally, the paper presents and proves correct a protocol that tracks (on-the-fly and without the help of an external observer) the immediate predecessors of each relevant event.

This set of protocols defines a suite of protocols that, in addition to their efficiency, provides a comprehensive view of causality tracking in distributed systems.

Keywords

Asynchronous System, Causality Tracking, Dependency Vector, Hasse Diagram, Immediate Predecessor, 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. Vector clocks have been introduced to allow processes to track causality (and concurrency) between the events they produce. The timestamp of an event produced by a process is the current value of the vector clock of the corresponding process. In that way, by associating vector timestamps with events it becomes possible to safely decide whether two events are causally related or not.

In many applications, detecting causal dependencies (or concurrency) on all events is not desirable. More precisely, when analyzing a distributed computation, one is interested only in a subset of events called the *relevant* events. This paper concentrates on the tracking of the causal relationship on such events. It proposes a suite of protocols that realize such a tracking, 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. It first presents an efficient implementation of vector clocks where the size of message timestamps can be less than n (the number of processes). Then, it introduces a protocol that provides a correct timestamping of the relevant events in presence of the following constraint: a message timestamp can piggyback at most b event identifiers (where b is a predefined constant, $1 \leq b \leq n$). To ensure that this constraint is never violated, this protocol can force a process to produce an additional "null" relevant event just before it sends a message.

Due to its definition, the partial order defined by the causality relation on the relevant events includes transitivity. Said another way, given only the vector timestamp associated with an event it is not possible to determine which events of its causal past are its immediate predecessors. Yet, some applications require to associate (on-the-fly and without additional messages) with each relevant event the set of its immediate predecessor relevant events. The paper presents and proves

correct a protocol that provides each relevant event with a timestamp made up of "pointers" that exactly identify its immediate predecessors. To our knowledge, this is the first time a correctness proof is presented for an immediate predecessors tracking protocol.

The protocols are presented in an incremental way. This helps to a better understanding of their behavior. Moreover, that shows that they actually define a suite of protocols that, in addition to their efficiency, provides a comprehensive view of causality tracking in distributed systems.

The paper is composed of six sections. Sections 2 and 3 introduce the computation model and vector clocks, respectively. Then, Sections 4, 5 and 6 incrementally presents three causality tracking protocols.

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 x-th event produced by process P_i. This set is structured as a partial order by Lamport's *happened before* relation [11], denoted $\stackrel{hb}{\rightarrow}$ and defined as follows:

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

The partial order $\widehat{H} = (H, \stackrel{hb}{\rightarrow})$ constitutes a formal model of the distributed computation it is associated with. Figure 1 depicts a distributed computation using the classical space-time diagram.

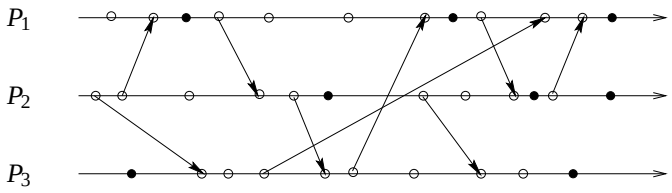

Figure 1: A Distributed Computation

In the rest of the paper, we will use the following definitions and notations:

• The *causal past* of an event e is the set of events f such that $f \overset{hb}{\rightarrow} e$.

• The set $\uparrow(e)$ contains the events of the causal past of e, plus e itself.

• If X_i is a local data structure managed by P_i, then $e.X_i$ denotes the value of X_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 denotes the sequence of events produced by P_i.

2.2 Relevant Events

At some abstraction level only some events of a distributed computation are relevant [5, 12] (those events are sometimes called *observable* events). The decision to consider an event as relevant can be up to the process, or triggered by some protocol. In this paper, we are not concerned by this decision. As an example, in Figure 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:

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

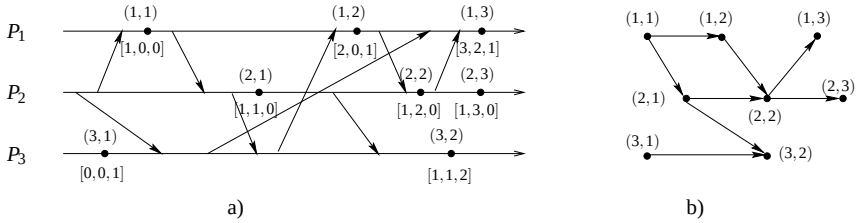

Figure 2: a) Relevant Events and their Timestamps. b) Hasse Diagram

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). Figure 2.a depicts an abstraction of the computation described in Figure 1 where only the black events are relevant. Each relevant event is identified by a pair (process id, sequence number). Finally, Figure 2.b represents the transitive reduction (Hasse diagram) associated with this abstraction[1].

In the following, R_i denotes the set of relevant events produced by P_i.

3 Vector Clocks

Vector clocks have been empirically used as an ad hoc device to solve specific problems before being captured and defined as a first class concept with the associated theory, simultaneously and independently by Fidge [3] and Mattern [13].

A vector clock system is a mechanism that associates timestamps with events in such a way that the comparison of their timestamps indicates whether the corresponding 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. Figure 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 [3, 7, 13]:

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

$(\forall k :\ e.VC[k] \leq f.VC[k]) \wedge (\exists k :\ e.VC[k] < f.VC[k])$ is denoted $e.VC < f.VC$. Let P_i be the process that has produced e. This additional information allows to simplify the previous relation that reduces to [3, 13]:

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

The traditional implementation of a vector clock system is given in [3, 13]. The major drawback of this implementation lies in 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 [1]. Nevertheless, this implementation can be easily

[1] The *transitive reduction* of a dag includes all its edges except its reflexive edges and its transitive edges.

improved in the following way. When P_i sends a message to P_j, it may piggyback only the entries that have been modified since its last sending to this process P_j. This improvement, proposed in [14], requires FIFO channels. It has a small local memory overhead, namely, a process has to manage only two additional arrays of size n.

4 Efficient Implementations of Vector Clocks

This section provides two protocols implementing a vector clock system that satisfy the following property: a message has not to systematically piggyback the whole vector clock of its sender. To our knowledge, the proposed protocols are the first that provide this property (on-the-fly, without additional control messages and without the help of an external observer), in a context where (1) channels are not necessarily FIFO (but, if the channels are FIFO, the protocol can still be improved), and (2) there is no a priori information on the communication graph connectivity or the communication pattern.

In order to attain this goal, each process P_i locally maintains additional control information, namely, a boolean matrix M_i. The protocol $\mathcal{P}1$ uses M_i to reduce the number of vector clock entries that have to be transmitted. The protocol $\mathcal{P}1'$ shows that this number can still be reduced if we allow a message to carry a few boolean vectors. It follows that the protocol $\mathcal{P}1'$ exhibits a tradeoff in the control information piggybacked by messages (integers *vs* boolean arrays).

4.1 Protocol $\mathcal{P}1$

This protocol relies on a very simple idea: P_i has not to transmit to a process P_j an entry of VC_i that is already known by P_j. To implement it, each process P_i has a boolean matrix M_i. Those matrices are managed in order to satisfy the following property:

$$\mathcal{M}_{VC} \equiv (\forall\, (i,j,k) :\ ((M_i[j,k] = 1) \Leftrightarrow (\text{ to } P_i\text{'s knowledge: } VC_j[k] \geq VC_i[k]))\ )$$

Let us note that, from the definition of vector clocks and the previous property on M_i, we can easily deduce the following invariant: $\forall\, (i,j) :\ M_i[j,j] = 1$ and $M_i[i,j] = 1$ (The diagonal and the ith row of M_i remain always equal to 1).

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

[R0] Initialization:
- $VC_i[1..n]$ is initialized to $[0,\ldots,0]$,
- $\forall\, (j,k) :\ M_i[j,k]$ is initialized to 1.

[R1] Each time it produces a relevant event e:
- P_i increments its vector clock entry $VC_i[i]$ ($VC_i[i] := VC_i[i] + 1$),

- P_i associates with e the timestamp $e.VC = VC_i$,
- P_i resets the ith column of its boolean matrix: $\forall j \neq i : M_i[j,i] := 0$ ([2]).

[R2] When it sends a message m to P_j, P_i attaches to m the following set (denoted $m.VC$) of event identifiers: $\{(k, VC_i[k]) \mid M_i[j,k] = 0\}$.

[R3] 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

The case of FIFO **channels.** If communication channels are FIFO, when a process P_i sends a message, in addition to the statements defined in the rule R2, it can immediately execute the update $M_i[j,k] := 1$. The FIFO property of channels can actually reduce the number of pairs piggybacked by messages (if $M_i[j,k] = 1$ at the next sending to P_j, $VC_i[k]$ will not be transmitted again to P_j).

4.2 Correctness Proof

Due to space limitations, the detail of this proof is omitted. The complete proof can be found in [8]. It consists in proving, by induction on the poset $\widehat{H}$, the two following invariants:

$$\mathcal{M}_{VC} \equiv (\forall e \in H_i,\ \forall (j,k) :\ (e.M_i[j,k] = 1) \Leftrightarrow$$
$$((\uparrow(e) \cap R_k = \emptyset) \vee (\exists m \text{ sent by } P_j \text{ to } P_i \text{ s.t. } send(m).VC_j[k] = e.VC_i[k]))$$

$$\mathcal{VC} \equiv (\forall i :\ \forall e \in H_i :\ \ \forall k :\ \ |\uparrow(e) \cap R_k| = e.VC_i[k]).$$

4.3 Protocol $\mathcal{P}1'$

The protocol $\mathcal{P}1'$ 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.

[2] Actually, the value of this column 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, 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 $\mathcal{P}2$, Section 5.2), by executing this "reset" of the column $M_i[*,i]$ only when P_i produces its first relevant event.

The rules R0 and R1 are the same as before. The rules R2 and R3 are modified in the following way. Let $M_i[*,k]$ denote the k-th column of M_i.

[R2'] 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): $\{(k, VC_i[k], M_i[*,k]) \mid M_i[j,k] = 0\}$.

[R3'] When it receives a message m from P_j, P_i executes the following updates:

$\forall (k, VC_j[k], M_j[k][1..n]) \in m.VC$ **do**:
  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 protocol $\mathcal{P}1'$ (left to the reader) is similar to the previous one. In the rest of the paper, $T_{1\rightarrow 1'}$ denotes the modification that transforms protocol $\mathcal{P}1$ into protocol $\mathcal{P}1'$.

5 Bounding the Size of Message Timestamps

5.1 The Problem

This section addresses the problem of associating a timestamp with each relevant event under the following constraint ("b-bound" Constraint): a message can piggyback at most b relevant event identifiers (i.e., pairs of process id + seq number), where b is a predefined constant ($1 \leq b \leq n$).

5.2 Protocol $\mathcal{P}2$

Additional test. A way to solve this problem consists in forcing a process to generate a "null" relevant event when the timestamp it has to send is about to have more than b elements. This "null" relevant event acts as a reset that reduces to 1 the size of the next timestamp. This can be easily realized by adding the following test to the rule R2 of protocol $\mathcal{P}1$.

[Test] When P_i has to send a message m to P_j, it first computes $d = \Sigma_{k\neq j} \mid \{k\ :\ (M_i[j,k] = 0)\} \mid$. If $d > b$, then P_i generates a "null" relevant event (so, it executes R1, which resets d to 1), and afterwards attaches to m its timestamp before sending this message.

Dependency vectors. As the numbers of entries that can be transmitted are now bounded, each vector clock used in protocol $\mathcal{P}1$ can no longer be maintained and has to be replaced by a data structure that "approximates" it, namely, a *dependency vector*. Similarly to a vector clock VC_i, a dependency vector DV_i tracks causal dependencies but in a "looser" way. More precisely, $\forall k \neq i$: $DV_i[k] \leq VC_i[k]$, and $DV_i[i] = VC_i[i]$.

Protocol $\mathcal{P}2$. In addition to the previous points, an important difference between protocol $\mathcal{P}1$ and protocol $\mathcal{P}2$ lies in the update of the matrix M_i when P_i produces a relevant event e (Rule R1). 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 $\mathcal{P}2$ is as follows.

[R0] Initialization:
- $DV_i[1..n]$ is initialized to $[0,\ldots,0]$,
- $\forall\ (j,k):\ M_i[j,k]$ is initialized to 1.

[R1] Each time it produces a relevant event e:
- P_i increments its dependency vector entry $DV_i[i]$ ($DV_i[i] := DV_i[i]+1$),
- P_i associates with e the timestamp $e.DV = DV_i$,
- P_i resets the boolean matrix as follows:
  - $\star\ \forall(j,k)$ such that $j,k \neq i:\ M_i[j,k] := 1$ ([3]).
  - $\star\ \forall j \neq i:\ M_i[j,i] := 0$.

[R2] When it sends a message m to P_j, P_i does the following:

- P_i first computes $d = \Sigma_{k\neq j} \mid \{k\ :\ (M_i[j,k] = 0)\} \mid$. If $d > b$, 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 (denoted $m.DV$) of event identifiers: $\{(k, DV_i[k]) \mid M_i[j,k] = 0\}$.

[R3] When it receives a message m from P_j, P_i executes the following updates:

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

So, protocol $\mathcal{P}2$ is incrementally obtained from protocol $\mathcal{P}1$ by (1) replacing VC_i by DV_i, (2) extending the reset rule in R1, and (3) adding a test in R2. Moreover, let us note that, as before the transformation $T_{1\rightarrow 1'}$ can be applied to protocol $\mathcal{P}2$ to get a protocol $\mathcal{P}2'$.

[3] Note that the ith row of M_i has not to be updated as it is always equal to $(1,\ldots,1)$ (see Section 4.1).

From a dependency vector timestamp to a vector clock timestamp. As we have seen, for any event e, $e.DV$ is not necessarily equal to $e.VC$ (except for the entry i, namely, $DV_i[i] = VC_i[i]$). It has been shown in [6] that the vector clocks VC can be rebuilt form the dependency vectors DV by computing, for each relevant event e, the following value: $e.VC := \max_{\{f \mid f \to e\}}(f.DV)$. This computation is iteratively implemented by the following "fixed-point" procedure. The function $\mathsf{max}(V1, V2)$ used in this procedure is defined in the following way: $\forall \ell \in \{1, \ldots, n\} : \mathsf{max}(V1, V2)[\ell] = max(V1[\ell], V2[\ell])$.

procedure Vector_Timestamp(var e : relevant event)
(1) $VC := e.DV$; % VC is an auxiliary vector variable %
(2) **repeat** $old\_VC := VC$;
(3) $\forall x \in \{1, \ldots, n\}$ **do let** f **be** the event identified $(x, old\_VC[x])$;
(4) $VC := \mathsf{max}(VC, f.DV)$ **enddo**
(5) **until** $(VC = old\_VC)$;
(6) **return** (VC) % the final value of VC defines $e.VC$ %

5.3 Related Work

The particular case where each primitive event is a relevant event and where $b = 1$ corresponds to the *direct dependency* technique introduced in [6]. The problem is easier to solve despite $b = 1$, as all communication events are "visible" (this issue is discussed at length in [4, 10]). The *incremental tracking* technique introduced in [10] has been designed to circumvent the limitations of the direct dependency technique when the relevant events are only a subset of the primitive events. To attain this goal, the proposed protocol tracks dependency incrementally and manages lists of clock values. As it does not manage special data structures to reduce timestamp sizes, messages can never carry less control information than in protocol $\mathcal{P}2$.

A principle similar to the generation of "null" events has been investigated in *communication-based checkpointing* protocols [9]. Independently from one another, processes take *basic* checkpoints (which correspond to relevant events). In order no basic checkpoint be useless (i.e., any local checkpoint has to belong to at least one consistent global checkpoint), the checkpointing protocol can force processes to take additional local checkpoints, namely *forced* checkpoints (which correspond to "null" events). The principle to add "null" events is actually very general (mainly encountered in synchronization problems).

6 Tracking the Immediate Predecessors

6.1 The Notion of Immediate Predecessor

Some applications (e.g., analysis of distributed executions [4], detection of distributed properties [5]) require to determine (on-the-fly and without the help of an external observer) the transitive reduction of the relation $\rightarrow$ (i.e., we must not consider transitive causal dependency). This means that we have to associate with each relevant event e, an attribute $e.TS$ that "points on" all its immediate relevant predecessors and on no other event. As an example, let us consider Figure 2.b). The event e identified $(1,3)$ has a single immediate predecessor, namely the event identified $(2,2)$, hence, $e.TS = \{(2,2)\}$. Differently, the event f=$(2,2)$ has two immediate predecessors, namely, the events $(2,1)$ and $(1,2)$, hence $f.TS = \{(2,1),(1,2)\}$.

To our knowledge, the distributed determination of immediate predecessors has been addressed for the first time in [2]. Some of us have also addressed it [4] to detect properties on the control flows of distributed computations. In both previous papers, this determination is presented with examples and without a correctness proof. So, to our knowledge, the proof of Section 6.3 is the first proof of an immediate predecessor tracking protocol.

The proposed protocol requires each application message to piggyback a complete vector clock. An important issue consists in the design of a protocol that should allow some messages to piggyback less information, in the same manner as causality tracking protocols presented above. To our knowledge, this problem has never been explicitly stated (and no solution has been proposed). This paper leaves the open problem.

6.2 Protocol $\mathcal{P}3$

Data structures. Let us consider a relevant event e produced by a process P_i. The attribute $e.TS$ is actually another kind of *timestamp* associated with events. Such timestamps involve two data structures.

• When it produces e, P_i has to decide, for every j, whether the last relevant event of P_j in the causal past of e is an immediate predecessor of e. This allows P_i to compute which are the members of the timestamp $e.TS$, i.e., the first part of each pair $(j,-)$ of $e.TS$.

To take a correct decision, P_i manages a boolean array IP_i, whose jth entry has the following meaning. $IP_i[j] = 1$ means that, if the event e currently produced by P_i is relevant, then the last relevant event of P_j, known by P_i, is an immediate predecessor of e.

• The second data structure is a vector clock, which allows P_i to determine the second part of each pair $(-,x)$ of $e.TS$.

Protocol $\mathcal{P}3$. The following protocol associates with each relevant event a timestamp made of the identifiers of its immediate predecessors [2, 4].

[R0] Initialization:

- Both $VC_i[1..n]$ and $IP_i[1..n]$ are initialized to $[0,\ldots,0]$.

[R1] Each time it produces a relevant event e:

- P_i increments its vector clock entry $VC_i[i]$ ($VC_i[i] := VC_i[i]+1$),
- P_i associates with e the timestamp $e.TS = \{(k, VC_i[k]) \mid IP_i[k] = 1\}$,
- P_i resets IP_i: $\forall\, \ell \neq i :\ IP_i[\ell] := 0; IP_i[i] := 1$.

[R2] When P_i sends a message m to P_j, it attaches to m the current values of VC_i (denoted $m.VC$) and the boolean array IP_i (denoted $m.IP$).

[R3] When it receives a message m from P_j, P_i executes the following updates:

$$\begin{array}{l}\forall k : \textbf{case } VC_i[k] < m.VC[k] \textbf{ then } VC_i[k] := m.VC[k];\\ \qquad\qquad\qquad\qquad\qquad\quad\ \ IP_i[k] := m.IP[k]\\ \qquad\qquad VC_i[k] = m.VC[k] \textbf{ then } IP_i[k] := \min(IP_i[k], m.IP[k])\\ \qquad\qquad VC_i[k] > m.VC[k] \textbf{ then } skip\\ \qquad \textbf{endcase}\end{array}$$

6.3 Correctness Proof

Lemma 1 $\forall i,\ \forall e \in H_i,\ \forall k :\ e.VC_i[k] = |{\uparrow}(e) \cap R_k|$.

Proof. The proof follows directly from the fact that the management of the vectors VC_i is the same as the traditional vector clock implementation (a formal proof can be found in [7]). □

Let e be an event produced by P_i ($e \in H_i$). In the rest of the proof, $lastr(e,k)$ denotes the $(e.VC_i[k])$th relevant event of P_k. From Lemma 1, we deduce that $lastr(e,k)$ is the last relevant event of P_k that is known by P_i. When $e \in R_i$ we have $lastr(e,i) = e$.

Lemma 2 *Let* $I\mathcal{P}(e,k) \equiv$
$(e.IP_i[k] = 1) \Leftrightarrow [(e.VC_i[k] > 0) \wedge (e \in R \Rightarrow k = i) \wedge (e \notin R \Rightarrow (\nexists e' \in R :\ lastr(e,k) \xrightarrow{hb} e' \xrightarrow{hb} e))]$. $\forall i,\ \forall e \in H_i,\ \forall k$, *we have:* $I\mathcal{P}(e,k)$.

Proof. The proof is by induction on the poset $\widehat{H}$. Due to space limitations, some parts of the proof are omitted here (see [8]).

Base case. $I\mathcal{P}(e,k)$ holds if e the first event of a process P_i. Omitted.

Induction case. Let $e \in H_i$. The induction assumption is: $\forall k,\ \forall e' \in ({\uparrow}(e) - \{e\})$, the predicate $I\mathcal{P}(e',k)$ holds. We have to prove that, $\forall k$, the predicate $I\mathcal{P}(e,k)$ holds. We proceed by case analysis.

• e is an internal but not relevant event, or e is a send event. Omitted.
• e is a relevant event ($e \in R_i$). Omitted.
• e is a receive event. Let $e = receive(m)$ and P_j be the sender of m.

Due to rule R3, we consider three cases:

Case 1. $VC_i[k] < m.VC[k]$. In that case, we have: $e.VC_i[k] = send(m).VC_j[k] > pred(e).VC_i[k]$, from which we conclude that $lastr(e,k) = lastr(send(m),k)$. Moreover, due to R3, we have $e.IP_i[k] = send(m).IP_j[k]$.

From the previous relations, the induction assumption $I\mathcal{P}(send(m),k)$ and $send(m) \notin R$, we obtain:

$(send(m).IP_j[k] = 1) \Leftrightarrow ((send(m).VC_i[k] > 0) \wedge$
$(\nexists e' \in R :\ lastr(send(m),k) \xrightarrow{hb} e' \xrightarrow{hb} send(m)))$, i.e.,

$(e.IP_i[k] = 1) \Leftrightarrow ((e.VC_i[k] > 0) \wedge (\nexists e' \in R :\ lastr(send(m),k) \xrightarrow{hb} e' \xrightarrow{hb} send(m)))$, i.e.,

$(e.IP_i[k] = 1) \Leftrightarrow ((e.VC_i[k] > 0) \wedge (\nexists e' \in R :\ lastr(e,k) \xrightarrow{hb} e' \xrightarrow{hb} send(m)))$.

More, by vector clock properties, $(pred(e).VC_i[k] < send(m).VC_j[k]) \Rightarrow \neg(send(m) \xrightarrow{hb} pred(e)) \Rightarrow (\nexists e' \in R :\ send(m) \xrightarrow{hb} e' \xrightarrow{hb} receive(m) = e)$. Hence,

$(e.IP_i[k] = 1) \Leftrightarrow ((e.VC_i[k] > 0) \wedge (\nexists e' \in R :\ lastr(e,k) \xrightarrow{hb} e' \xrightarrow{hb} e))$, from which we conclude, as e is not a relevant event, that $I\mathcal{P}(e,k)$ holds.

Case 2. $VC_i[k] = m.VC_i[k]$. In that case, we have the following equalities:

$pred(e).VC_i[k] = send(m).VC_i[k] = e.VC_i[k]$,
$lastr(pred(e),k) = lastr(send(m),k) = lastr(e,k)$.

Moreover, due to the induction assumption, we have $I\mathcal{P}(send(m),k)$ and $I\mathcal{P}(pred(e),k)$. We consider two cases.

- $send(m).IP_j[k] = pred(e).IP_i[k]$.

In that case, the second part of the **case** statement entails the same behavior as the first part of rule R3, which has been proved in the previous item (1).

- $send(m).IP_j[k] \neq pred(e).IP_i[k]$.

In that case, we necessarily have $e.IP_i[k] = 0$ (second part of rule R3). Hence, the left part of $I\mathcal{P}(e,k)$ is false. We have to prove that, in this case, its right part is also false.

Whatever the values of the first sub-predicates of the right part, we concentrate only on the third one (note that $e \notin R$). There are two sub-cases.

$\star$ $send(m).IP_j[k] = 0$. Due to the induction assumption $I\mathcal{P}(send(m),k)$ holds, from which we get $\exists e' \in R :\ lastr(send(m),k) \xrightarrow{hb} e' \xrightarrow{hb} send(m)$. As $lastr(send(m),k) = last(e,k)$, we get: $\exists e' \in R :\ lastr(e,k) \xrightarrow{hb} e' \xrightarrow{hb} send(m) \xrightarrow{hb} e$. Hence, in that case, both sides of $I\mathcal{P}(e,k)$ have the same values.

$\star$ $pred(e).IP_i[k] = 0$. This case is actually similar to the previous one, where $I\mathcal{P}(pred(e),k)$ is used instead of $I\mathcal{P}(send(m),k)$ and $lastr(send(m),k)$ is replaced by $lastr(pred(e),k)$.

In both cases, both sides of $I\mathcal{P}(e,k)$ have the same value. Consequently, $I\mathcal{P}(e,k)$ holds.

Case 3. $VC_i[k] > m.VC_i[k]$. Omitted.

Hence, if e is a receive event, we have $\forall k: \ I\mathcal{P}(e,k)$. □

Theorem 1 *The protocol $\mathcal{P}3$ tracks immediate predecessors: for any relevant event e, the timestamp $e.TS$ contains the identifiers of its immediate predecessors and no other event identifier.*

Proof. Let $e \in R_i$. As (rule R1) $e.TS$ is defined before the update of IP_i, we have: $e.TS = \{(k, e.VC_i[k]) \mid pred(e).IP_i[k] = 1\}$. We have to prove that $e.TS$ includes the identifiers of all the immediate predecessors of e, and only them.

Let us consider a pair $(k,x) \in e.TS$. We have first to show that x is the sequence number of the last relevant event of P_k that belongs to $\uparrow(e)$. This follows directly from Lemma 1.

We have now to show that $e.TS$ is made up of exactly the immediate predecessors of e. This can be shown by proving the following statement that characterizes the property for the event $lastr(pred(e),k)$ to be an immediate predecessor of e (remind that e is an internal event):

$$(pred(e).IP_i[k] = 1) \Leftrightarrow (\nexists e' \in R: \ lastr(pred(e),k) \stackrel{hb}{\rightarrow} e' \stackrel{hb}{\rightarrow} e).$$

Proof of "⇒". By assumption, $pred(e).IP_i[k] = 1$. There are two cases.

• If $pred(e) \in R$: In that case, due to Lemma 2 applied to $pred(e)$, we conclude that $k = i$. Moreover, as $pred(e)$ is produced by P_i, we have $lastr(pred(e),i) = pred(e)$. Finally, due to the definition of $pred(e)$, there is no e' such that $pred(e) \stackrel{hb}{\rightarrow} e' \stackrel{hb}{\rightarrow} e$. By combining these assertions, we get $\nexists e' \in R: \ lastr(pred(e),i) = pred(e) \stackrel{hb}{\rightarrow} e' \stackrel{hb}{\rightarrow} e$.

• If $pred(e) \notin R$: In that case, $lastr(pred(e),k) = lastr(e,k)$. From Lemma 2 applied to $pred(e)$, we conclude that $\nexists e' \in R: \ lastr(e,k) = last(pred(e),k) \stackrel{hb}{\rightarrow} e' \stackrel{hb}{\rightarrow} pred(e) \stackrel{hb}{\rightarrow} e$.

Proof of "⇐". By assumption, $pred(e).IP_i[k] = 0$. There are two cases.

• If $pred(e) \in R$: So, both $pred(e)$ and e belong to R_i. Moreover, due to the rule R1, we have $e.VC_i[i] > pred(e).VC_i[i] > 0$, and:

- If $k = i$: Due to the update of $IP_i[i]$ to 1 (rule R1) when $pred(e)$ is produced, we have $pred(e).IP_i[i] = 1$, which contradicts the case assumption. Hence, this case cannot occur.

- If $k \neq i$: In that case, $\exists e' = pred(e) \in R: \ lastr(pred(e),k) \stackrel{hb}{\rightarrow} e' \stackrel{hb}{\rightarrow} e$.

• If $pred(e) \notin R$: From Lemma 2 applied to $pred(e) \notin R$, we conclude $\exists e' \in R: \ lastr(pred(e),k) \stackrel{hb}{\rightarrow} e' \stackrel{hb}{\rightarrow} pred(e)$. Hence, $\exists e' \in R: \ lastr(pred(e),k) \stackrel{hb}{\rightarrow} e' \stackrel{hb}{\rightarrow} e$. □

References

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

[2] C. Diehl, C. Jard, and J-X. Rampon. Reachability Analysis of Distributed Executions. *Proc. TAPSOFT*, Springer-Verlag LNCS 668, 629-643, 1993.

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

[4] E. Fromentin, C. Jard, G-V. Jourdan, and M. Raynal. On-the-fly Analysis of Distributed Computations. *Inf. Proc. Letters*, 54:267-274, 1995.

[5] E. Fromentin, M. Raynal. Shared Global States in Distributed Computations. *Journal of Computer and System Sciences*, 55(3):522-528, 1997.

[6] J. Fowler, W. Zwanepoel. Causal Distributed Breakpoints. *Proc. 10th IEEE Int. Conf. on Distributed Computing Systems*, 134-141, 1990.

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

[8] J-M. Hélary, G. Melideo, and M. Raynal. Tracking Causality in Distributed Systems: A Suite of Efficient Protocols. http://www.irisa.fr/bibli/publi/2000/1301/1301.html

[9] J-M. Hélary, A. Mostéfaoui, R. H. B. Netzer, and M. Raynal. Communication-Based Prevention of Useless Ckeckpoints in Distributed Computations. *Distributed Computing*, 13(1):29-43, 2000.

[10] C. Jard, G-V. Jourdan. Incremental Transitive Dependency Tracking in Distributed Computations. *Parallel Processing Letters*, 6(3):427-435, 1996.

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

[12] K. Marzullo K., L. Sabel. Efficient Detection of a Class of Stable Properties. *Distributed Computing*, 8(2):81-91, 1994.

[13] F. Mattern. Virtual Time and Global States of Distributed Systems. *Proc. Int. Conf. Parallel and Distributed Algorithms*, (Cosnard, Quinton, Raynal, Robert Eds), North-Holland, 215-226, 1988.

[14] M. Singhal, A. Kshemkalyani. An Efficient Implementation of Vector Clocks. *Information Processing Letters*, 43:47-52, 1992.