Illustrating the Use of Vector Clocks in Property Detection: An Example and a Counter-Example

Michel Raynal

IRISA, Campus de Beaulieu, 5042 Rennes Cedex, France
raynal@irisa.fr

1 Introduction

Logical (scalar, vector or matrix) clocks are a powerful mechanism used by a lot of distributed algorithms. This paper is on vector clocks: it surveys their main features and is particularly focused on their power and their limitation. In that sense, this paper complements [5, 6] and may be seen as a critical and practical introduction to vector clocks.

The paper is divided into four main sections. Section 2 introduces a model for distributed executions. Section 3 describes vector clocks and their basic properties. Vector clocks are a simple mechanism that allows processes to track causality between the events they produce. Then, Section 4 and Section 5 study two problems related to causality. The first problem consists in detecting a conjunction of stable local predicates. The second problem consists in recognizing the occurrence of a very simple event pattern. It is shown that simple vector clocks are insufficient to solve the second problem which, actually, requires more sophisticated clocks, namely, vector of vector clocks. So, this paper exhibits a frontier between problems that can be solved using simple vector clocks and problems requiring more sophisticated vector clock systems.

2 Distributed Computations

2.1 Partially Ordered Set of Events

Distributed Computations A distributed program is made up of n sequential local programs which can communicate and synchronize only by exchanging messages. The execution of a local program gives rise to a sequential process. Let $P_1, \ldots, P_n$ be this finite set of processes. We assume that, at run-time, each ordered pair of communicating processes (P_i, P_j) is connected by a reliable channel. Message transmission delays are finite but unpredictable, process speeds are positive but arbitrary: the underlying computation model is asynchronous.

Partial Order of Events Execution of an internal/send/receive statement produces a corresponding internal/send/receive event. Let e_i^x be the x-th event produced by process P_i. The sequence $h_i = e_i^1 e_i^2 \ldots e_i^x \ldots$ constitutes the history of P_i. Let H be the set of events produced by a distributed computation. This set is structured as a partial order by Lamport's causal precedence relation [3], denoted "$\rightarrow$" and defined as follows:

P. Amestoy et al. (Eds.): Euro-Par'99, LNCS 1685, pp. 806–814, 1999.

$$e_i^x \rightarrow e_j^y \Leftrightarrow \begin{array}{l}(i = j \wedge x \leq y) \text{ (local precedence)} \quad \vee \\ (\exists m : e_i^x = send(m) \ \wedge \ e_j^y = receive(m)) \text{ (msg prec.)} \quad \vee \\ (\exists\ e_k^z \ : \ e_i^x \rightarrow e_k^z \ \wedge e\ {}_k^z \rightarrow e_j^y) \quad \text{(transitive closure)}.\end{array}$$

The partial order $\widehat{H} = (H, \rightarrow)$ constitutes a model of the distributed computation it is associated with. Figure 1 depicts a distributed computation where events are denoted by black points. Two events e and f are *concurrent* (or *causally independent*) if $\neg(e \rightarrow f) \wedge \neg(f \rightarrow e)$. The *causal past* of event e is the partially ordered set of events f such that $f \ \rightarrow \ e$. Similarly, the *causal future* of event e is the partially ordered set of events f such that $e \ \rightarrow \ f$.

2.2 Partially Ordered Set of Local States

Local States Let σ_i^0 be the initial state of process P_i. Event e_i^x entails P_i's local state change from σ_i^{x-1} to σ_i^x: σ_i^x is the local state of P_i resulting from its partial history $h_i^x = e_i^1 \ldots e_i^x$. We say that e_i^y *belongs to* σ_i^x (denoted $e_i^y \in \sigma_i^x$) if $y \leq x$. In Figure 1 local states are represented by rectangular boxes.

Partial Order of Local States Let Σ be the set of all local states associated with a distributed computation $\widehat{H}$. Lamport's precedence relation can be extended to local states in the following way:

$$\boxed{(\sigma_i^x \rightarrow \sigma_j^y) \Leftrightarrow (e_i^{x+1} \rightarrow e_j^y) \qquad (SE)}$$

Local states not related by "$\rightarrow$" are said to be *concurrent*, (denoted $\|$). More formally: $(\sigma_i^x \| \sigma_j^y) \Leftrightarrow (\neg(\sigma_i^x \rightarrow \sigma_j^y) \wedge \neg(\sigma_j^y \rightarrow \sigma_i^x))$. In Figure 1, we have $\sigma_3^1 \rightarrow \sigma_1^4$ and $\sigma_1^3 \| \sigma_3^5$. As for events, the *causal past* of a local state σ_i^x is the partially ordered set of local states σ_k^z such that $\sigma_k^z \ \rightarrow \ \sigma_i^x$.

Consistent Global States A global state Σ is a set of n local states $(\sigma_1, \cdots, \sigma_n)$ one from each process. It is consistent if $\forall i \neq j$ we have $\sigma_i \| \sigma_j$. This means that a global state $\Sigma = (\sigma_1, \cdots, \sigma_n)$ is consistent if, for any pair of its local states (σ_i, σ_j), there is no message m such that $receive(m) \in \sigma_i \wedge send(m) \notin \sigma_j$. If such a message m exists, it is called *orphan* with respect to the pair (σ_i, σ_j). Let us again consider Figure 1. $\Sigma_1 = (\sigma_1^3, \sigma_2^4, \sigma_3^2)$ is consistent, while $\Sigma_2 = (\sigma_1^5, \sigma_2^6, \sigma_3^4)$ is not consistent. There is no orphan message with respect to Σ_1, while m_4 is orphan with respect to the pair $(\sigma_3^4, \sigma_2^6) \subset \Sigma_2$. Actually, m_4 creates the dependency $\sigma_3^4 \rightarrow \sigma_2^6$ which makes Σ_2 inconsistent.

3 Vector Clocks

As a concept, with the associated theory, vector clocks have been introduced in 1988, simultaneously and independently by Fidge [1] and by Mattern [4].

A Causality Tracking Timestamping Mechanism A vector clock system is a mechanism that associates timestamps with events (local states) in such a way that the comparison of their timestamps indicates whether the corresponding events (local states) are or not causally related (and, if they are, which one is the first). This timestamping system is implemented in the following way. Each process P_i has a vector of integers $VC_i[1..n]$ and:

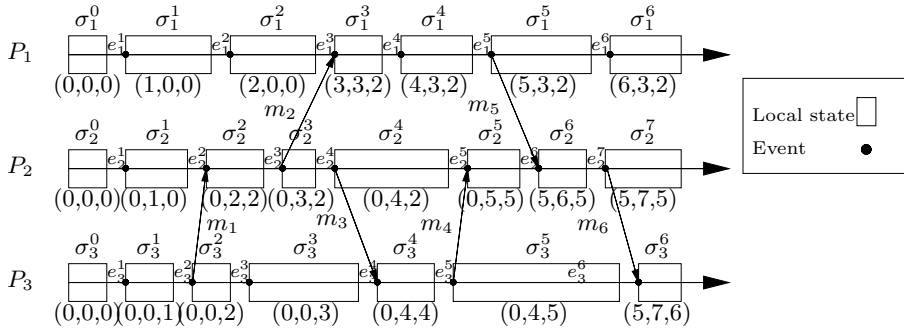

Fig. 1. Local States of a Distributed Computation

R1 Each time it produces an event, P_i increments its vector clock entry $VC_i[i]$ ($VC_i[i] := VC_i[i] + 1$) to indicate it has progressed.

R2 When a process P_i sends a message m, it attaches to it 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 usually abbreviated as $VC_i := max(VC_i, m.VC)$).

Note that $VC_i[i]$ counts the number of events produced so far by P_i. Moreover, $VC_i[j]$ represents the number of events produced by P_j that belongs to the current causal past of P_i. When a process P_i produces an event e (and enters the associated local state σ) it associates with them a vector timestamp whose value is equal to the current value of VC_i, namely, $e.VC = \sigma.VC =$ current value of VC_i. Figure 1 shows vector timestamp values associated with events and local states. As an example we have: $e_4^2.VC = \sigma_4^2.VC = (0, 4, 2)$.

Let $e.VC$ and $f.VC$ be the vector timestamps associated with two distinct events e and f, respectively. The following property is the fundamental property associated with vector clocks [1, 2, 4][1]:

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

Let P_i be the process that has produced e. This additional information allows to simplify the previous relation that reduces to [1, 4]:

$$\boxed{(e \rightarrow f) \Leftrightarrow (e.VC[i] \leq f.VC[i]) \qquad (R)}$$

Let $\sigma 1$ and $\sigma 2$ be local states of P_i and P_j, respectively. From the definition SE and the previous relations we get: $(\sigma 1 \rightarrow \sigma 2) \Leftrightarrow (\sigma 1.VC[i] < \sigma 2.VC[i])$.

Timestamps of Global States Relations between vector clocks and global states have been studied from a formal point of view by several authors [2, 4]. Here, we only consider some of these properties through a set of simple examples taken from Figure 1.

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

Vector clocks allow to associate timestamps to global states in a very easy way. First, let us consider a local state, *e.g.*, σ_3^4. It is timestamped $(0, 4, 4)$ and this timestamp identifies the consistent global state $(\sigma_1^0, \sigma_2^4, \sigma_3^4)$ which is the first consistent global state to which σ_3^4 belongs. More generally, the timestamp of a consistent global state Σ is defined as the component-wise maximum of the timestamps of the local states that compose it. Let us consider $\Sigma = (\sigma_1^4, \sigma_2^4, \sigma_3^4)$. Its timestamp is the component-wise maximum (denoted max) of $(4, 3, 2), (0, 4, 2)$ and $(0, 4, 4)$, *i.e.*, $(4, 4, 4)$.

In the same way if we consider two (or more) consistent global states Σ_1 and Σ_2, timestamped $\Sigma_1.VC$ and $\Sigma_2.VC$, respectively, the global state defined by the timestamp $max(\Sigma_1.VC, \Sigma_2.VC)$ is the first consistent global state (called $max(\Sigma_1, \Sigma_2)$) that includes the causal past of both Σ_1 and Σ_2. As an example, let us consider $\Sigma_1 = (\sigma_1^0, \sigma_2^2, \sigma_3^3)$ timestamped $(0, 2, 3)$, and the global state $\Sigma_2 = (\sigma_1^4, \sigma_2^4, \sigma_3^2)$ timestamped $(4, 4, 2)$. The global state $\Sigma = max(\Sigma_1, \Sigma_2)$ is a consistent global state, namely $(\sigma_1^4, \sigma_2^4, \sigma_3^3)$, timestamped $(4, 4, 3)$, that occurs after Σ_1 and after Σ_2. (Actually, from a theoretic point of view, the set of consistent global states defines a lattice [2, 4, 6]).

4 Detection of a Conjunction of Stable Local Predicates

4.1 The Problem

A predicate is *local* to a process P_i if it is only on local variables of P_i. A local predicate LP_i is *stable* if, as soon as it becomes true, it remains true forever. The notation $\sigma_i \models LP_i$ will be used to indicate that P_i's local state σ_i satisfies the local predicate LP_i. Let LP_1, LP_2, ..., LP_n be n local predicates, one per process. A consistent global state $\Sigma = (\sigma_1, \cdots, \sigma_n)$ satisfies the global predicate $LP_1 \wedge LP_2 \cdots \wedge LP_n$ (denoted $\Sigma \models (\bigwedge_i LP_i)$) if $\bigwedge_i (\sigma_i \models LP_i)$.
The problem consists in detecting on-the-fly, and without using additional control messages, the first consistent global state Σ that satisfies a conjunction of stable local predicates.

4.2 How to Solve It

Let us consider Figures 2.a and 2.b. The fact that local predicate LP_i is satisfied, is indicated by thickening process P_i's axis. In both figures, the global state $\Sigma = (\sigma_1^{y_1}, \sigma_2^{y_2}, \sigma_3^{y_3})$ (timestamped (y_1, y_2, y_3)) is the first to satisfy the conjunction of stable local predicates. Let us first examine Figure 2.a.

- When P_1 receives m_1, it learns nothing new about predicate detection.
- When P_1 receives m_3 (*i.e.*, when it enters local state $\sigma_1^{x_1}$), it can learn (by appropriately tracking causality with vector clocks) that there is a consistent global state, namely $(\sigma_1^0, \sigma_2^{x_2}, \sigma_3^0)$, that partially satisfies the global predicate $(\sigma_2^{x_2} \models LP_2)$.
- Message m_4 gives P_2 the knowledge that $\sigma_3^{x_3} \models LP_3$. So, (using vector clocks) m_5 can carry the information that the global state timestamped $(0, x_2, x_3)$ partially satisfies the global predicate (namely, $(\sigma_1^0, \sigma_2^{x_2}, \sigma_3^{x_3}) \models LP_2 \wedge LP_3$).

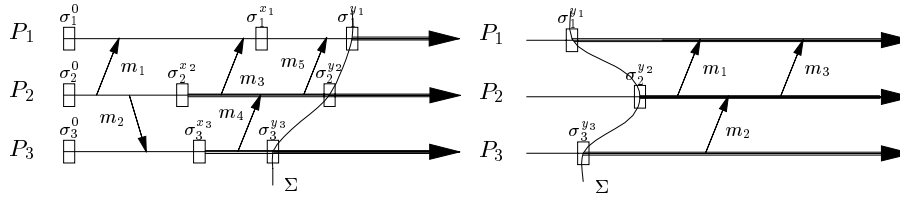

Fig. 2. Detection of $\bigwedge_i LP_i$

• So, when P_1 receives m_5 it learns this, and when LP_1 becomes true, P_1 can safely detects that Σ, timestamped (y_1, y_2, y_3), is the first consistent global state satisfying $\bigwedge_i LP_i$ ($\sigma_2^{y_2}$ and $\sigma_3^{y_3}$ being the local states that immediately follow the sendings of m_5 and m_4, respectively).
Let us now examine Figure 2.b. In that situation, due to flow of exchanged messages, P_1 can learn, when it receives m_3, that Σ, timestamped (y_1, y_2, y_3), is the first consistent global state satisfying the global predicate $LP_1 \wedge LP_2 \wedge LP_3$.

4.3 Data Structures

The previous discussion shows that we need two thinks to solve the problem: (1) Track causality to be able to identify *consistent* global states; and (2) Track which local predicates are *satisfied* and track the *first* consistent global state in which those local predicates are satisfied. Hence, each process P_i is endowed with the following data structures:
• $VC_i[1..n]$: a vector clock (initialized to $(0, 0, ..., 0)$).
• SAT_i : a set (initially empty) of process identifiers, with the following meaning: $j \in SAT_i \Leftrightarrow P_i$ knows P_j entered a local state from which LP_j is true.
• $FIRST_i$: a vector timestamp (initialized to $(0, 0, ..., 0)$) that defines the first consistent global state, known by P_i, in which all the predicates defined in SAT_i are satisfied. More precisely: $\forall\ j\ : (j \in SAT_i \Rightarrow (\sigma_j^{FIRST_i[j]} \models LP_j))$. Note that if $SAT_i = \{1, 2, \cdots, n\}$, then $FIRST_i$ defines the first consistent global state Σ that satisfies $\bigwedge_j LP_j$. In Figure 2.a, $FIRST_1 = VC_1 = (y_1, y_2, y_3)$ when LP_1 becomes true. In Figure 2.b, when P_1 receives m_3, it updates SAT_1 (the value of which becomes $\{1, 2, 3\}$) and $FIRST_1$ takes the value $max((y_1, y_2, 0), (0, y_2, y_3))$, *i.e.*, $FIRST_1 = (y_1, y_2, y_3)$. In both figures, $(\sigma_1^{y_1}, \sigma_2^{y_2}, \sigma_3^{y_3}) \models (LP_1 \wedge LP_2 \wedge LP_3)$.

4.4 The Detection Protocol

The protocol executed by a process P_i is described in Figure 3. Let e and σ denote the last event produced by P_i and the corresponding local state (entered by P_i just after executing e). Their vector timestamps ($e.VC$ and $\sigma.VC$) have the same value which is equal the current value of VC_i. The protocol

is composed of two procedures and three statements $S1$, $S2$ and $S3$ associated with the production by P_i of an internal event, a send event and a receive event, respectively. The local variable $done_i$ (initialized to $false$) is set to the value $true$ the first time LP_i is satisfied (then it remains true forever). The notation $VC := max(VC1, VC2)$ (statement S3) is an abbreviation for $\forall k \in 1..n : VC[k] := max(VC1[k], VC2[k])$. It defines a vector clock update. The protocol consists in:

procedure *detected*? **is**
  if $SAT_i = \{1, 2, \ldots, n\}$ **then** $FIRST_i$ defines the first consistent global state Σ that satisfies $\bigwedge_j LP_j$ **fi**
procedure *check_LP*$_i$ **is**
  if $(\sigma_i^x \models LP_i)$ **then** $SAT_i := SAT_i \cup \{i\}$;$FIRST_i := VC_i$;
    $done_i := true$; *detected*? **fi**

(S1) **when** P_i **produces an internal event** (e)
  $VC_i[i] := VC_i[i] + 1$; execute e and move to σ;
  if $\neg done_i$ **then** *check_LP*$_i$ **fi**

(S2) **when** P_i **produces a send event** (e=send m **to** P_j)
  $VC_i[i] := VC_i[i] + 1$; move to σ; **if** $\neg done_i$ **then** *check_LP*$_i$ **fi**;
  $m.VC := VC_i$; $m.SAT := SAT_i$; $m.FIRST := FIRST_i$;
  send (m) **to** P_j % m carries $m.VC, m.SAT$ and $m.FIRST$ %

(S3) **when** P_i **produces a receive event** (e=receive (m))
  $VC_i[i] := VC_i[i] + 1$; $VC_i := max(VC_i, m.VC)$;
  move to σ; % by delivering m to the process % **if** $\neg done_i$ **then** *check_LP*$_i$ **fi**;
  if $\neg(m.SAT \subseteq SAT_i)$ **then** $SAT_i := SAT_i \cup m.SAT$;
    $FIRST_i := max(FIRST_i, m.FIRST)$;
    detected? **fi**

Fig. 3. Detection Protocol for $\bigwedge_i LP_i$

• The management of the vector clock VC_i (in $S1$, $S2$ and $S3$).
• The combined management of variables SAT_i and $FIRST_i$:
- When LP_i becomes true, VC_i defines the first consistent global state in which LP_i, plus the local predicates already in SAT_i, are satisfied. So $FIRST_i$ is defined as VC_i (procedure *check_LP*$_i$).
- When P_i sends a message m (Statement $S2$), it attaches to it the current values of SAT_i and $FIRST_i$. Those values are denoted $m.SAT$ and $m.FIRST$.
- When P_i receives a message m (Statement S3), it first executes the reception, moves to the new current state and tests if its local predicate becomes satisfied. Then, P_i checks if it learns something new about the satisfaction of local local predicates (by testing $\neg(m.SAT \subseteq SAT_i)$). If it learns something new, it computes the first consistent global state in which the local predicates of $m.SAT \cup SAT_i$ are satisfied. As indicated in the previous discussion, this global state is timestamped $max(FIRST_i, m.FIRST)$.

- When something new happens (from the point of view of local predicates detection), P_i tests the condition $SAT_i = \{1, 2, \cdots, n\}$, to know if the full global predicate, namely $\bigwedge_i LP_i$, is satisfied.

5 Limitation of Simple Vector Clocks

Vector clocks provide each process with a "counter-based view" of its causal past that displays limitations to solve some "causality"-related problems. We present here such a problem and its solution.

5.1 Recognition of a Simple Pattern

Let us consider a distributed execution that produces two types of internal events: *black* and *white* (communication events are tagged *white*). Given two blacks events s and t, the problem consists in deciding if there is another black event u such that $s \rightarrow u \quad \wedge \quad u \rightarrow t$? Let $black(e)$ be a predicate indicating if event e is black. More formally, given two events s and t, the problem consists in deciding if the following predicate $\mathcal{P}(s,t)$ is true:

$$\boxed{\mathcal{P}(s,t) \equiv (black(s) \wedge black(t)) \wedge (\exists u \neq s, t : (black(u) \wedge (s \rightarrow u \wedge u \rightarrow t)))}$$

To show that vector clocks do not allow to solve this problem let us consider Figures 4.a and 4.b. In these two executions, both event s have the same timestamp: $s.VC = (0, 0, 2)$. Similarly, both events t have also the same timestamp, namely, $t.VC = (3, 4, 2)$. But, as the reader can verify, execution (b) satisfies the pattern, while (a) does not.

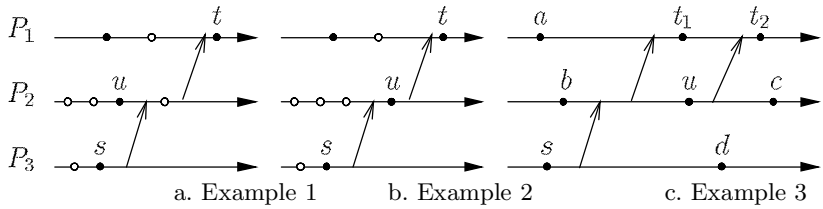

Fig. 4. Recognize a Pattern

5.2 How to Solve It

Observe that for the predicate $\mathcal{P}(s,t)$ to be true, there must be a black event in the causal past of t which has s in its causal past. This problem is related to causality, but two levels of predecessors appear in the predicate $\mathcal{P}$. Track "second order knowledge" on the past requires vector of vectors clocks.

Which Clocks? $\mathcal{P}(s,t)$ can be decomposed into two predicates $\mathcal{P}_1(s,u,t)$ and $\mathcal{P}_2(s,u,t)$ in the following way: $\mathcal{P}(s,t) \equiv (\exists u \ : \mathcal{P}_1(s,u,t) \wedge \mathcal{P}_2(s,u,t))$, where $\mathcal{P}_1(s,u,t) \equiv (black(s) \wedge black(u) \wedge black(t))$ and $\mathcal{P}_2(s,u,t) \equiv (s \rightarrow u \wedge u \rightarrow t)$.

$\mathcal{P}_1$ indicates that only the black events are relevant for the predicate detection: so, only black events have to be tracked. This means vector clocks can be managed in the following way: (1) A process P_i increments $VC_i[i]$ only when it produces a black event; (2) The other statements associated with vector clocks are left unchanged.

Let us consider Figure 4.c, where only black events are indicated. We have $\mathcal{P}(s,t_1) = false$, while (due to u or to t_1) $\mathcal{P}(s,t_2) = true$. The underlying idea to solve the problem lies in associating two timestamps with each black event e:
• A vector timestamp $e.VC$ (counting only black events).
• An array of vector timestamps $e.MC[1..n]$ such that $e.MC[j]$ contains the vector timestamp of the last black event of P_j that causally precedes e ($e.MC[j]$ can be considered as a pointer from e to the last event that precedes it on P_j).
When considering Figure 4.c, we have: $t_1.MC[1] = a.VC$, $t_1.MC[2] = b.VC$, $t_1.MC[3] = s.VC$, $t_2.MC[1] = t_1.VC$, $t_2.MC[2] = u.VC$ and $t_2.MC[3] = s.VC$.

Each process P_i has a vector clock $VC_i[1..n]$ and a vector of vector clocks $MC_i[1..n]$. Those variables are managed as described in Figure 5. Let us note that, in statement S3, $MC_i[k]$ and $m.MC[k]$ contain vector timestamps of two black events of P_k. It follows that one of them is greater or equal to the other: $max(MC_i[k], m.MC[k])$ is its timestamp.

(S1) **when** P_i **produces a black event** (e)
$VC_i[i] := VC_i[i] + 1$; % one more black event on P_i %
$e.VC = VC_i; e.MC = MC_i$;
$MC_i[i] := VC_i$ % vector timestamp of P_i's last black event %

(S2) **when** P_i **executes a send event** (e=send m **to** P_j)
$m.VC := VC_i$; $m.MC := MC_i$;
send (m) **to** P_j % m carries $m.VC$ and $m.MC$ %

(S3) **when** P_i **executes a receive event** (e=receive(m))
$VC_i := max(VC_i, m.VC)$; % update of the local vector clock %
$\forall\, k :\ MC_i[k] := max(MC_i[k], m.MC[k])$
% record vector timestamp of the last black predecessor on each P_k %

Fig. 5. Clock Management for Detecting $\mathcal{P}(s,t)$

An Operational Pattern Detection Predicate Let us first remark that, as the protocol considers only black events, the predicate $\mathcal{P}_1$ is trivially satisfied by any triple of events. So, detecting $\mathcal{P}(s,t)$ amounts to only detect $\exists u : \mathcal{P}_2(s,u,t)$. Given s and t with their timestamps (namely, $s.VC$ and $s.MC$ for s; $t.VC$ and $t.MC$ for t), the predicate $(\exists u : \mathcal{P}_2(s,u,t)) \equiv (\exists u : s \rightarrow u \rightarrow t)$ can be

restated in a more operational way using vector timestamps. More precisely: $(\exists u : s \rightarrow u \rightarrow t) \equiv (\exists u : s.VC < u.VC < t.VC)$.

If such an event u does exist, it has been produced by some process P_k and belongs to the causal past of t. Consequently, its vector timestamp is such that: $\exists k : u.VC \leq t.MC[k]$. From this observation, the previous relation translates in: $(\exists u : s \rightarrow u \rightarrow t) \equiv (\exists k \ : \ s.VC < t.MC[k] < t.VC)$.
As $\forall k$, $t.MC[k]$ is the vector timestamp of a black event in the causal past of t, we have $\forall k : t.MC[k] < t.VC$. Consequently, the pattern detection predicate simplifies and becomes: $\boxed{\mathcal{P}(s,t) \equiv (\exists k \ : \ s.VC < t.MC[k])}$.

References

[1] Fidge C.J., Timestamp in Message Passing Systems that Preserves Partial Ordering, *Proc. 11th Australian Computing Conference*, pp. 56-66, 1988.

[2] Garg V.K., *Principles of Distributed Systems*, Kluwer Acad. Press, 274 pages, 1996.

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

[4] Mattern F., Virtual Time and Global States of Distributed Systems, *Proc. "Parallel and Distributed Algorithms" Conference*, North-Holland, pp. 215-226, 1988.

[5] Raynal M. and Singhal S., Logical Time: Capturing Causality in Distributed Systems, *IEEE Computer*, 29(2):49-57, 1996.

[6] Schwarz R. and Mattern F., Detecting Causal Relationships in Distributed Computations: in Search of the Holy Grail. *Distributed Computing*, 7:149-174, 1994.