Fundamentals of Distributed Computing: A Practical Tour of Vector Clock Systems

R. Baldoni[†], M. Raynal[*]

[†] DIS, Universitá di Roma *La Sapienza*, Via Salaria 113, Roma, Italy
[*] IRISA, Campus de Beaulieu, 35042 Rennes-cedex, France
baldoni@dis.uniroma1.it raynal@irisa.fr

Abstract

A *vector clock* system is a timestamping mechanism that tracks *causality* among events produced by a distributed computation. This paper is a practical introduction to vector clock systems. It reviews their fundamental protocols, properties and uses. The paper also discusses approximate vector clocks and dependency vectors. Concepts and mechanisms presented in this paper are of primary importance to distributed computing systems engineers. They will provide them with a solid background to better understand causality related distributed computing problems. Such an experience can be used to solve a given problem with the appropriate protocols.

Keywords: Asynchronous Distributed Systems, Causality Tracking, Direct Dependency, Distributed Computing, Timestamps, Vector Clocks.

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 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*. This model 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.

This paper considers a key concept of asynchronous distributed systems, namely, *causality*. 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 *could be* a consequence of the occurrence of the other. Events produced by processes are message sendings, message receives or internal events. 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. More precisely, a vector clock is an array of n integers (one entry per process) where the entry j counts the number of relevant events produced by the process P_j. The timestamp of an event produced by a process (or of the local state generated by this event) is the current value of the vector clock of the corresponding process. In that way, by associating vector timestamps with events or local states it becomes possible to safely decide whether two events or two local states are causally related or not.

This paper constitutes a practical and critical advanced introduction to vector clocks. It presents their basic properties and examples of problems they can solve. To this end three problems are investigated: causal broadcast, detection of message stability and detection of an event pattern. These problems are of increasing difficulty. The first one requires simple vector clocks, while the last one requires vectors of vector clocks. This increasing difficulty should help the reader better understand causality related problems and the way they can be solved [5, 9].

The main drawback of a vector clock system lies in its inability to face scalability problems. To fully capture the causality relation among the events produced by the processes of a distributed computation, a vector clock system requires vectors of size n (n being the number of processes). To circumvent this problem, two types of bounded vector clocks (vector clocks whose size is bounded by a constant $k < n$) have been introduced, namely, approximate vector clocks [12] and k-dependency vectors [1]. So, the paper surveys them and presents their main properties.

The concepts and mechanisms presented in this paper, together with the illustrative examples, can help distributed systems engineers to get a deeper insight into the causality related problems they have to cope with. The basic protocols that are presented should help them to select the appropriate mechanism to solve their problems efficiently.

2 A Model of Distributed Execution

Distributed Program and Distributed Computation
A distributed program is made up of n sequential local programs which, when executed, can 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 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 c_{ij} through which P_i can send messages to P_j. Message transmission delays are finite but unpredictable. 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.

A Distributed Computation as a Partial Order of Events
Execution of an internal/send/receive statement produces an internal/send/receive event. Let e_i^x ($x \geq 1$) 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 "happened-before" relation [7], denoted "$\rightarrow$" and defined as follows:

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

$e \rightarrow f$ means that it is *possible* for the event e to affect the event f. As a consequence $\neg(e \rightarrow f)$ means e cannot affect f. The partial order $\widehat{H} = (H, \rightarrow)$ constitutes a formal 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 $e \rightarrow f$. As an example,

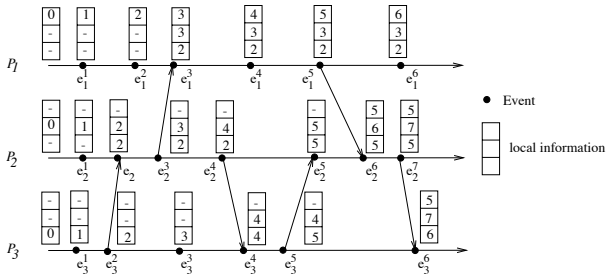


Figure 1: An example of a Distributed Computation

in Figure 1 we have $e_3^1 \rightarrow e_1^3$, $e_1^3 \| e_3^5$ and the *causal past* of the event e_2^2 corresponds to the set $\{e_2^1, e_3^1, e_3^2\}$.

Sidebar 1: A Historical View of Vector Clocks

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, Parker *et al.* used similar vectors to detect mutual inconsistencies of the copies of a replicated data [e]. Ladin and Liskov used them to detect obsolete data [c]. Raynal used them to prevent drift among a set of n logical scalar clocks [f]. Johnson and Zwaenepoel [b] on one side, and Strom and Yemini [h] on an other side, used them to track causal dependencies between events in their checkpointing protocols. Schmuck used them to implement efficient causal broadcast in asynchronous distributed systems [g].
As a concept, with the associated theory, vector clocks have been introduced in 1988, simultaneously and independently by Fidge [a] and by Mattern [d]. These works have provided clear definitions of the concept, have studied and proved fundamental properties associated with vector clocks and have promoted them as a first class mechanism to study and solve causality related problems in distributed systems.

[a] Fidge C.J., Timestamp in Message Passing Systems that Preserves Partial Ordering, *Proc. 11th Australian Computing Conference*, pp. 56-66, 1988.
[b] Johnson D.B. and Zwaenepoel W., Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing, *Proc. 7th ACM Symposium on Principles of Distributed Computing*, pp. 171-181, 1988.
[c] Liskov B. and Ladin R., Highly Available Distributed Services and Fault-Tolerant Distributed Garbage Collection, *Proc. 5th ACM Symposium on Principles of Distributed Computing*, 1986, pp.29-39.
[d] Mattern F., Virtual Time and Global States of Distributed Systems, *Proc. "Parallel and Distributed Algorithms" Conference*, (Cosnard, Quinton, Raynal, Robert Eds), North-Holland, pp. 215-226, 1988.
[e] Parker D.S. *et al.*, Detection of Mutual Inconsistency in Distributed Systems, *IEEE Transactions on Software Engineering*, SE9(3):240-246, 1983.
[f] Raynal M., A Distributed Algorithm to Prevent Mutual Drift Between n Logical Clocks. *Information Processing Letters*, 24:199-202, 1987.
[g] Schmuck F., *The Use of Efficient Broadcast in Asynchronous Distributed Systems.* Doctoral Dissertation, Tech. Report TR88-928, Dept Computer Science, Cornell University, 124 p., 1988.
[h] Strom R.E. and Yemini S. Optimistic Recovery in Distributed Systems, *ACM Transactions on Computer Systems*, 3(3):204-226, 1985.

3 Vector Clocks: A Causality Tracking Mechanism

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 are 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 $VC_i[1..n]$ (initialized to $[, 0, \ldots, 0]$) that is maintained in the following manner (see Sidebar 1 for a historical view of vector clocks, and Sidebar 2 for an efficient implementation):

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

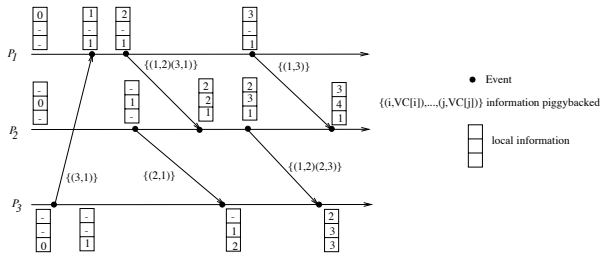
R2 When a process P_i sends a message m, it attaches to it the current value of VC_i. Let $m.VC$ denote this value.

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 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, for $i \neq j$ $VC_i[j]$ represents the number of events produced by P_j that belong to the current causal past of P_i. When a process P_i produces an event e it can associate with them a vector timestamp whose value is equal to the current value of VC_i. Figure 1 shows vector timestamp values associated with events and local states. As an example we have: $e_2^6.VC = (5, 6, 5)$.

Sidebar 2: An Efficient Implementation of Vector Clocks

A major drawback of a vector clock system lies in the fact that each message has to carry an array of n integers (where n is the size of the application, *i.e.*, the total number of processes). It has been shown that, in the worst case, this is a necessary requirement [b]. To face this problem, Singhal and Kshemkalyani have proposed a simple technique that, in the average case, reduces the size of vector timestamps piggybacked by messages [c]. This technique is based on the empirical observation that only few processes are likely to interact frequently (this is especially true when the number of processes is high) and, then, between two successive sending events from process P_i to process P_j only a few entries of the vector clock are expected to change. In such a case, there is no point to attach to each outgoing message from P_i to P_j a whole vector clock (see rule 2 of Section 3). It suffices to piggyback only the information relative to the entries which are changed, this actually corresponds to a set of tuples $(proc\_id, VC[proc\_id])$. Therefore, this technique is expected to save communication bandwidth at the cost of a local memory overhead as a process needs to keep track of the last values sent to each process (*i.e.*, one vector for each process) in order to select the set of tuples to piggyback on each message. The Figure describes an example of the progress of vector clocks using Singhal-Kshemkalyani's technique. It is important to note that for this technique to work correctly, communication channels have to be first-in first-out.
Another practical problem that vector clock systems have to face, is the overflow of their vector entries. A general solution to solve this problem is described in [d]. In the context of causal ordering, another solution can be found in [a].



[a] Baldoni R., A Positive Acknowledgment Protocol for Causal Broadcasting. *IEEE Transactions on Computers*, 47(12): 1341-1350, 1998.
[b] Charron-Bost B., Concerning the Size of Logical Clocks in Distributed Systems. *Information Processing Letters*, 39:11-16, 1991.
[c] Singhal M. and Kshemkalyani A., An Efficient Implementation of Vector Clocks. *Information Processing Letters*, 43:47-52, 1992.
[d] Yen L.-Y. and Huang T.-L., Resetting Vector Clocks in Distributed Systems. *Journal of Parallel and Distributed Systems*, 43:15-20, 1997.

Properties of Vector Clocks

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 [4, 8]:

$$\forall (e, f) : e \neq f : ((e \rightarrow f) \Leftrightarrow (e.VC < f.VC)$$

where $e.VC < f.VC$ is an abbreviation for $(\forall k : \ e.VC[k] \leq f.VC[k]) \wedge (\exists k : \ e.VC[k] < f.VC[k])$. Let P_i be the process that has produced e. This additional information allows to simplify the previous relation that reduces to [4, 8]:

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

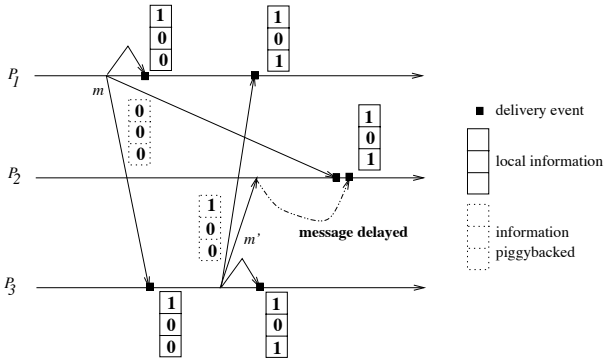


Figure 2: Causal Delivery of Broadcast Messages

4 Illustration 1: Causal Broadcast

This section and the next two sections describe three causality related problems whose solutions can be vector clock-based. The aim of these sections is to show the reader how vector clocks are the appropriate tool to solve causality related problems.

The *causal broadcast* notion has been introduced by Birman and Joseph [2] in order to reduce the asynchrony of communication channels as perceived by application processes. It states that the order in which messages are delivered to application processes cannot violate the precedence order (defined by the $\rightarrow$ relation) of the corresponding broadcast events. More precisely, if two broadcast messages m and m' are such that $broadcast(m) \rightarrow broadcast(m')$, then any process must deliver m before m'. If the broadcasts of m and m' are concurrent, then m and m' can be delivered in some order at a process and in a different order at another process.

Practically, this means that when a message m is delivered to a process, all messages whose broadcasts causally precede the broadcast of m, have already been delivered to that process. The ISIS system has been the first to propose such a communication abstraction [2].

A Vector Clock-Based Implementation of Causal Broadcast
Several vector clock-based implementations of causal broadcast have been proposed (*e.g.*, [3, 10]). They are based on the following simple idea: it consists for a receiving process P_i to delay the delivery of a message m until all the messages broadcast in the causal past of m have been delivered to P_i. Let us consider Figure 2. When m' arrives at P_2, its delivery has to be delayed because (1) m' arrived at P_2 before m and (2) the sending of m causally precedes m'. To this end, each process P_i has to manage a vector clock (VC_i) tracking its current knowledge on the number of messages that have been sent by each process.

A simple broadcast protocol (similar to the one presented in [3]) is described in Figure 3. Broadcast events are the relevant events of a computation, and $VC_i[j]$ represents P_i's knowledge of the number of messages broadcast by P_j and delivered to P_i. Each message m, piggybacks a vector timestamp $m.VC$ revealing how many messages have been broadcast by each process in the causal past of m's broadcast. Then, when a process P_i receives a message m, it delays its delivery until all the messages "that belong to its causal past" have been delivered. This is expressed by a simple condition involving the vector clock of the receiving process P_i and the vector timestamp ($m.VC$) of the received message m, namely: $(\forall x \in \{1, \ldots, n\} : m.VC[x] \leq VC_i[x])$. The resulting causal broadcast protocol is described in Figure 3 (vectors are initialized to $[0, \ldots, 0]$).

```
procedure broadcast(m) % issued by P_i %
        m.VC := VC_i; % construct the timestamp of m %
        ∀x ∈ {1, ..., n} do send(m) to P_x enddo % broadcast event %
        VC_i[i] := VC_i[i] + 1 % one more broadcast by P_i %

when P_i receives m from P_j % m piggybacks its vector timestamp m.VC %
        delay the delivery until (∀x ∈ {1, ... n} : m.VC[x]≤VC_i[x]);
        if i ≠ j then VC_i[j] := VC_i[j] + 1; % update control variable %
        deliver m to the upper layer % produce the delivery event %
```

Figure 3: A Simple Causal Broadcast Protocol

5 Illustration 2: Message Stability Tracking

Let us consider applications where processes broadcast operations to all the others, and where each process has to eventually receive the same set of operations sent by correct processes. This problem abstracts the notion of *reliable broadcasting* [6] and *eventual consistency* [11], just to name a few. In the context of reliable broadcasting, operations correspond to messages and, to meet the problem requirements in the presence of sender process failures and network partitions, each process has to buffer a copy of every message it sends or receives. If a process P_i fails, any process which has a copy of a message m sent by P_i can forward m to a process P_j that detected it has not received m. This can induce a rapid growth of the buffer at each process with the risk of overflowing. Therefore, one needs some policy to reduce buffer overflow occurrence. A simpler observation shows that it is not necessary to buffer a message that has been delivered to all its intended destinations. Such a message is called a *stable* message. Stable messages can be safely discarded from the local buffer of a process.

A *message stability detection* protocol is responsible for managing the process buffers. Such a protocol can be lazy (stability information is piggybacked on application messages), use gossiping (stability information is propagated by periodic broadcast of control messages) or hybrid (stability information is propagated by using both piggybacking and gossiping).

A Vector Clock-Based Implementation of a Lazy Stability Protocol
In order to concentrate on the buffer management actions, we consider the simple case where communication channels are first-in first-out, and we assume there is no failure. Moreover no causal delivery is ensured (i.e, each message is delivered as soon as it is received).

Broadcast events are the relevant events of the computation. Each process P_i has a vector (MC_i) of vector clocks. This vector of vectors is such that the vector $MC_i[k]$ keeps track of messages delivered to P_k to P_i's knowledge. More precisely, $MC_i[k][\ell]$ $(i \neq k, \ell)$ represents P_i's knowledge of the number of messages delivered by P_k and sent by P_ℓ; and $MC_i[i][i]$ represents the sequence number of the next message sent by P_i. Hence, the minimum value over column j of MC_i (*i.e.*, $min_{1\leq x\leq n}(MC_i[x][j])$) represents P_i's knowledge of the sequence number of the last message sent by P_j that is stable.

To propagate stability information, each message m sent by P_i piggybacks the identity of its sender ($m.sender$) and a vector timestamp $m.VC$ indicating how many messages have been delivered by P_i from each other process P_ℓ (*i.e.*, $m.VC$ corresponds to the vector $MC_i[i][*]$).

A local buffer ($buffer_i$) is updated by two operations: deposit(m) which inserts a message m in the buffer and discard(m) which removes m from the buffer. A message is buffered by a process immediately after its receipt and it is discarded as soon as it becomes stable

(*i.e.*, when the process learns that m has been delivered by all the processes). The stability predicate for a message m can be expressed by the following condition : $m.VC[m.sender] \leq min_{1 \leq x \leq n}(MC_i[x][m.sender])$ where $m.VC[m.sender]$ represents the sequence number of m. The resulting protocol is described in Figure 4.

procedure rel-broadcast(m) % issued by P_i%
  $m.VC := MC_i[i][*]$; % construct the timestamp of m %
  $m.sender := i$;
  $\forall x \in \{1, \ldots, n\}$ **do** send(m) **to** P_x **enddo** % broadcast event %
  $MC_i[i][i] := MC_i[i][i] + 1$ % one more broadcast by P_i %

when P_i **receives** m **from** P_j % m piggybacks its vector timestamp $m.VC$ %
  deposit(m); % add m to the local buffer %
  $MC_i[j][*] := m.VC$; % update of P_i's view of P_j's vector %
  if $i \neq j$ then $MC_i[i][j] := MC_i[i][j] + 1$; % one more message delivered from P_j%
  deliver m to the upper layer % produce the delivery event %

when $(\exists\ m \in buffer_i :\ m.VC[m.sender] \leq min_{1 \leq x \leq n}(MC_i[x][m.sender]))$
  discard(m) % suppress m from the local buffer %

Figure 4: A Simple Lazy Stability Tracking Protocol

Figure 5 describes an example of running this stability tracking protocol. P_3 discards m immediately after the receipt of m' as $min_{1 \leq x \leq 3}(MC_j[x][m.sender])$ is equal to zero which corresponds to the sequence number of m. At the end of the example, P_1's and P_3's buffers contain m' and m'', while P_j's buffer contains only m''. Note that to extend this protocol to handle causal delivery we need just to add a delivery condition, similar to the one of Figure 3, in the second clause of protocol of Figure 4.

6 Illustration 3: Recognition of a Simple Pattern

The first illustration (causal broadcast) has shown a simple use of vector clocks: each process manages a simple vector clock and each message carries a vector timestamp. In the second illustration (message stability detection) each message carries a vector timestamp but each process has to manage a vector of vector clocks. The problem studied in this section, coming from

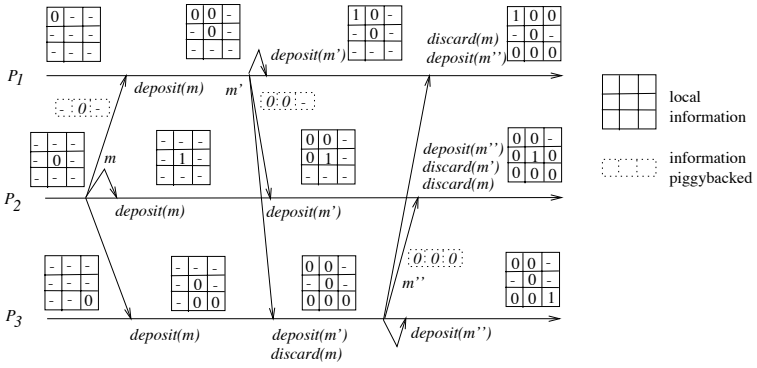


Figure 5: Lazy Stability Tracking: an Example

distributed debugging, shows that some problems require that not only each process manages a vector of vector clocks, but also that each message carries a vector of vector clocks. This series of illustrations with increasing difficulty shows that the solving of causality related problems is not always tractable with simple vector clocks (see [5, 9] for a list of such problems).

Let us consider a distributed execution that produces two types of internal events: some internal events are tagged *black* while the others are tagged *white*. All communication events are tagged *white*. (As an example, in a distributed debugging context, an internal event is tagged *black* if the associated local state satisfies a given local predicate, otherwise it is tagged *white*.) Given two black 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:

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

To show vector clocks do not allow to solve this problem let us consider Figure 6. 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, the right execution satisfies the pattern, while the left one does not! (Remark also that s -and also t- will have the same timestamp in both executions, even if vector clocks are incremented only upon the occurrence of black events!)

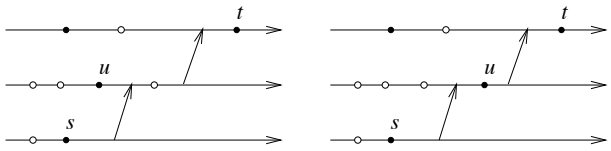


Figure 6: Recognize a Pattern

Which Clocks to Solve It?

Observe that for the predicate $\mathcal{P}(s,t)$ to be true, there must exist a black event in the causal past of t which has s in its causal past. This problem concerns the detection of causality: so, vector clocks are required. Moreover, two levels of predecessors appear in the predicate $\mathcal{P}$. Tracking two levels of predecessors requires a vector of vector clocks.

The predicate $\mathcal{P}(s,t)$ can be decomposed into two sub-predicates $\mathcal{P}_1(s,u,t)$ and $\mathcal{P}_2(s,u,t)$ in the following way:

$$\mathcal{P}(s,t) \equiv (\exists u \ : \mathcal{P}_1(s,u,t) \wedge \mathcal{P}_2(s,u,t))$$

with:

$$\mathcal{P}_1(s,u,t) \equiv (black(s) \wedge black(u) \wedge black(t))$$

$$\mathcal{P}_2(s,u,t) \equiv (s \rightarrow u \wedge u \rightarrow t)$$

$\mathcal{P}_1$ indicates that only the black events are relevant for the predicate detection. So, the detection of $\mathcal{P}(s,t)$ requires only to track black events. This means we can use vector clocks 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. (Actually,

black events define the abstraction level at which the distributed computation has to be observed to detect $\mathcal{P}$. All the other events -namely, the white events- are not relevant for detecting $\mathcal{P}$).

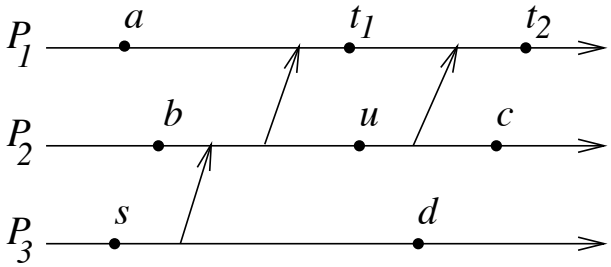


Figure 7: $\mathcal{P}(s, t_2)$ is true while $\mathcal{P}(s, t_1)$ is not

Let us consider Figure 7, where only black events are indicated. We have $\mathcal{P}(s, t_1) = false$, while $\mathcal{P}(s, t_2) = true$. The underlying idea to solve the problem lies in associating two timestamps with each black event e:

- A vector timestamp $e.VC$ (as indicated, only black events are counted in this vector timestamp),
- An array of vector timestamps $e.MC[1..n]$ whose meaning is the following: $e.MC[j]$ contains the vector timestamp of the last black event of P_j that causally precedes e.

Note that $e.MC[j]$ can be considered as a pointer from e to the last event that precedes it on P_j. When considering Figure 7, we have:

$$t_1.MC[1] = a.VC \qquad t_1.MC[2] = b.VC \qquad t_1.MC[3] = s.VC$$
$$t_2.MC[1] = t_1.VC \qquad t_2.MC[2] = u.VC \qquad t_2.MC[3] = s.VC$$

Managing the Clocks

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 8.

As before, the notation $VC := max(VC1, VC2)$ (statement S3) is an abbreviation for $\forall k \in 1..n : VC[k] := max(VC1[k], VC2[k])$. Moreover, 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. The result of $max(MC_i[k], m.MC[k])$ is this greatest timestamp.

Remark. Let us finally note that $MC_i[i][i] = VC_i[i] - 1$, and $\forall k \neq i : \ MC_i[i][k] = VC_i[k]$. So, the vector clock VC_i can be deduced from the diagonal of the matrix MC_i. This can be used to reduce the number and the size of data structures managed by processes and carried by messages.

The Pattern Detection Predicate

As we have seen, $\mathcal{P}(s, t)$ is equivalent to $\exists u : \mathcal{P}_1(s, u, t) \wedge \mathcal{P}_2(s, u, t)$. Let us 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:

$$(\exists u : s \rightarrow u \rightarrow t) \equiv (\exists u : s.VC < u.VC < t.VC)$$

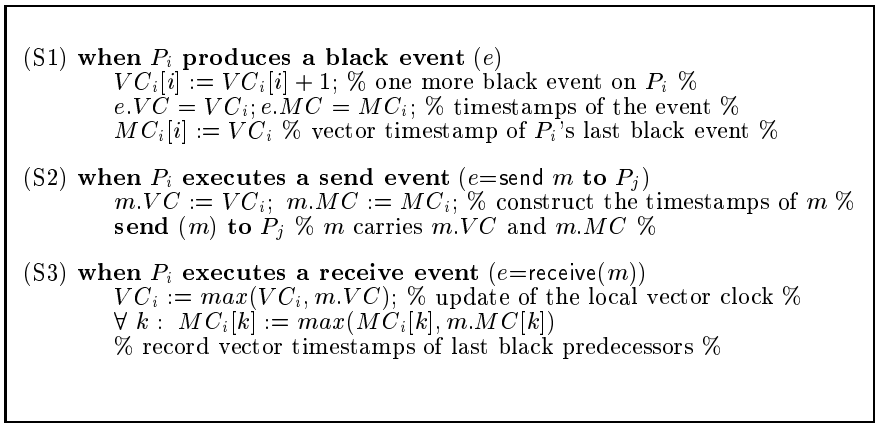


Figure 8: Detection Protocol for $\mathcal{P}(s,t)$

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:

$$\mathcal{P}(s,t) \equiv (\exists k\ :\ s.VC < t.MC[k])$$

To summarize, when this condition is true, it means that it does exist a process P_k that has produced a black event u such that:

- The vector timestamp of u ($u.VC$) is $\leq t.MC[k]$
- The event u belongs to the causal past of t (because $t.MC[k] < t.VC$)
- The event u belongs to the causal future of s (because $s.VC < u.VC \leq t.MC[k]$)

So, when the system is equipped with the vector clock system described previously, the predicate $\mathcal{P}(s,t)$ can be evaluated by a simple test, namely, $\exists k : s.VC < t.MC[k]$. Moreover, when the identity of the process (say P_i) that has produced s is known, then this test can be simplified; using the relation (R) (Section 3), the test becomes: $\exists k : s.VC[i] < t.MC[k][i]$.

7 Bounded Vector Clocks

The main drawback of a system of vector Clocks is scalability. This section presents two ways to limit the size of vector clocks. The first one (approximate vector clocks [12]) can be seen as a *space folding* approach. It can be used when one is only interested in never missing causality between related events (so, it is accepted that two events be perceived as ordered while they are actually concurrent). The second (k-dependency vectors [1]) can be seen as a *time folding* approach. In that case, the bounded timestamp of an event provides causal dependencies that, when recursively exploited, allow to reconstruct the vector timestamp of the event.

Approximate Vector Clocks

In some applications we are only interested in an approximation of the causality relation such that (let $e.TS$ be the timestamp associated with e):

$$(e \rightarrow f) \Rightarrow (e.TS \ < \ f.TS)$$

Such a timestamping never violates causality in the sense that, from $e.TS \ < \ f.TS$ we can safely conclude that $\neg(f \rightarrow e)$. If we optimistically conclude $e \rightarrow f$, then we can be wrong, as it is possible that e and f are not causally related. That is why, concluding $e \rightarrow f$ from $e.TS \ < \ f.TS$, constitutes an approximation of the causality relation.

Approximate vector clocks have been introduced by Torres and Ahamad [12]. They provide a simple mechanism that allow to associate approximate vector timestamps with events. Let us consider vector clocks whose size is bounded by a constant k (with $k < n$). So, $TS_i[1..k]$ is the approximate vector clock of P_i. Moreover, let f_k be a deterministic function from $\{1, \ldots, n\}$ to $\{1, \ldots, k\}$. Given a process identity i, this function associates with it the entry $f_k(i)$ of all vector clocks $TS[1..k]$ (*i.e.*, $TS[f_k(i)]$).

The implementation of such a timestamping system is similar to the one described in Section 3. Each process P_i manages its vector clock $TS_i[1..k]$ (initialized to $(0, \ldots, 0)$) in the following way:

R1 Each time it produces a send/receive/internal event, P_i updates its vector clock TS_i to indicate it has progressed: $TS_i[f_k(i)] := TS_i[f_k(i)] + 1$.

R2 When a process P_i sends a message m, it attaches to it the current value of TS_i, let $m.TS$ be this value.

R3 When P_i receives a message m, it updates its vector clock in the following way: $\forall \ x \in \{1, \ldots, k\} : \ TS_i[x] := max(TS_i[x], m.TS[x])$ (*i.e.*, $TS_i := max(TS_i, m.TS)$).

Combined with the function f_k, these rules ensure the x-th entry of any vector clock is shared by all processes P_i such that $f_k(i) = x$. Such an entry sharing makes the vector clocks "approximate" as far as causality tracking is concerned. These approximate vector clocks are characterized by the following properties [12]:

$$(e \rightarrow f) \Rightarrow (e.TS < f.TS)$$

$$(e.TS < f.TS) \Rightarrow ((e \rightarrow f) \vee (e||f))$$

More generally, we have:

- If $k = n$ and $\forall \ i : \ f_k(i) = i$, then we get classic vector clocks that track full causality. In that case, the entry i of the vector clock system is "private" to P_i in the sense only P_i can entail its increase.

- If $k = 1$, then $\forall \ i : \ f_k(i) = 1$ and all processes share the unique entry of the (degenerated) vector. The resulting clock system is nothing else than Lamport's scalar clock system [7]. This scalar clock system is well known for its property $(e \rightarrow f) \Rightarrow (e.TS \ < \ f.TS)$.

  A lot of applications actually consider the timestamp of an event e produced by P_i as the pair $(e.TS, i)$. This provides an easy way to totally order (without violating causal relations) the set of all the events produced by a distributed computation. This is the famous total order relation defined by Lamport [7], namely, e and f being produced by P_i and P_j, respectively, e is ordered before f if $(e.TS < f.TS) \vee ((e.TS = f.TS) \wedge (i < j))$. Let us also note that scalar clocks allow to detect some concurrent events, more precisely, $(e.TS \ = \ f.TS) \Rightarrow (e$ and f are concurrent).

- If $1 < k < n$ then all processes P_i such that $f_k(i) = x$ share the same entry x of the vector clock system. This sharing adds "false" causality detections that make this vector clock system approximate. Experimental results [12] show that with $n = 100$ and $2 < k < 5$, the percentage of situations in which $e \rightarrow f$ is concluded from $e.TS < f.TS$, while actually e and f are concurrent, is less than 10%.

Dependency Vectors

Given two events e and f of a distributed computation such that $e.TS < f.TS$, approximate vector clocks are unable to conclude whether $e \rightarrow f$ or $e||f$. For such a pair, they can only answer $\neg(f \rightarrow e)$. So, an important question, is the following one: "Does it exist a vector clock system with a bounded number of entries, from which it is possible to reconstruct the causality relation, *i.e.*, to conclude (maybe after some computation) that $e \rightarrow f$, or $f \rightarrow e$, or $e||f$?". This section shows that dependency vectors [1] answer this question.

A k-dependency vector clock system is characterized by the following behavior. Each process P_i has a vector clock ($DV_i[1..n]$ which is initialized to $(0, \ldots, 0)$) that is managed in the following way:

R1 Each time it produces an event, P_i updates its dependency vector DV_i to indicate it has progressed: $DV_i[i] := DV_i[i] + 1$.

R2 When a process P_i sends a message m, it attaches to it a set of k pairs $(x, DV_i[x])$. This set always includes the pair $(i, DV_i[i])$. Let $m.TS$ denote the set piggybacked by m.

R3 When P_i receives a message m, it updates its dependency vector in the following way: $\forall\ x$ such that $(x, DV[x]) \in m.TS$: $DV_i[x] := max(DV_i[x], DV[x])$.

Sidebar 3: Reconstructing Vector Timestamps from Dependency Timestamps

To reconstruct the vector timestamp associated with an event we add a checker process to the computation. Each time a process executes a relevant event e, it sends the checker process the corresponding dependency vector $e.DV$. The checker has n queues, one for each process, where it stores the timestamps received from the corresponding process. If the checker requires a timestamp which has not yet been deposited in the corresponding queue, it waits until it has received the required information. The algorithm executed by the checker process to compute the vector timestamp associated with an event e operates iteratively. It is described below. The function $\mathsf{max}(V_1, V_2)$ is defined in the following way: $\forall \ell \in \{1 \dots n\} \ : \ \mathsf{max}(V_1, V_2)[\ell] = max(V_1[\ell], V_2[\ell])$.

procedure Vector_Timstamp(var e : event)
    $e.V := e.DV$;
    repeat
        $old\_V := e.V$;
        for each $x \in \{1, \dots, n\}$ **do** $e.V := \mathsf{max}(e.V, e_x^{old\_V[x]}.DV)$ **enddo**
    until $(e.V = old\_V)$;
let $e.VC = e.V$;

The **repeat** statement actually computes the transitive closure of the causal precedence relation. The inner loop moves forward the current dependency timestamp of e by incorporating the new dependencies revealed by events belonging to old_V and not taken into account by $e.V$. When $(e.V = old\_V)$ there are no more dependencies to be incorporated in $e.V$, and then $e.V$ is the vector timestamp of e namely $e.VC$.
Such a reconstruction of a vector timestamp from dependency vectors has been investigated in [a,d,e]. References [b] and [c] consider dependency vectors with $k = 1$. They use them to define consistent global checkpoints and causal breakpoints, respectively.

[a] Baldy P., Dicky H., Medina R., Morvan M. and Vilarem J.-M., Efficient Reconstruction of the Causal Relationship in Distributed Systems. *Proc. 1st Canada-France Conference on Parallel and Distributed Computing*, LNCS #805, pp. 101-113, 1994.
[b] Baldoni R., Cioffi G., Hélary J.-M. and Raynal M., Direct Dependency-Based Determination of Consistent Global Checkpoints, *Proc. 3rd Int. Conference on Principles of Distributed Systems (OPODIS'99)*, Hanoi, 1999.
[c] Fowler J. and Zwaenepoel W., Causal Distributed Breakpoints. *Proc. 10th Int. IEEE Conference on Distributed Computing Systems*, pp. 134-141, 1990.
[d] Garg V.K., *Principles of Distributed Systems*, Kluwer Academic Press, 274 pages, 1996.
[e] Schwarz R. and Mattern F., Detecting Causal Relationship in Distributed Computations: In Search of the Holy Grail, *Distributed Computing*, 7(3):149-174, 1994.

A k-dependency vector clock system provides each process with a n size vector, but each message carries only a subset of size k. This subset always includes the current value of $DV_i[i]$ (where P_i is the sender process). The choice of the other $k-1$ values is left to the user. A good heuristics [1] consists in choosing the last modified $k-1$ entries of DV_i. It is easy to see that $k = n$ provides classical vector clocks.

Let us consider two events e and f timestamped $e.DV$ and $f.DV$, respectively. Moreover, let us assume that e has been produced by P_i. The k-dependency vector protocol ensures the following property:

$$(e.DV[i] \leq f.DV[i]) \Rightarrow (e \rightarrow f)$$

Note that the implication is in one direction only. this means that it is possible that actually $e \rightarrow f$ while $e.DV[i] > f.DV[i]$. But, differently from approximate vector clocks, k-dependency vectors allow (using additional computation) to reconstruct the causality relation (see Sidebar

3). Of course, according to the problem to be solved, k-dependency vectors and approximate vectors can be used simultaneously.

8 Conclusion

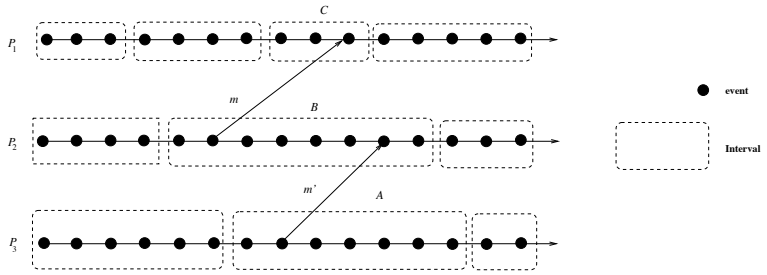
The concept of causality among events is fundamental to the design and analysis of distributed programs. A *vector clock* system is a timestamping mechanism that tracks *causality* among events produced by a distributed computation. This paper has provided a practical introduction to vector clock systems. It has reviewed their fundamental protocols, their properties and has discussed problems that have vector clock-based solutions. Approximate vector clocks and dependency vectors have also been presented. Moreover, the limitation of simple vector clocks to solve some causality related problems has been investigated (those problems require vectors of vector clocks).

The concepts and mechanisms presented in this paper are of primary importance to distributed computing systems engineers. They will provide them with a solid background that will help them better understand causality related distributed problems and solve those problems with appropriate protocols.

Let us finally remark that, as well as scalability, a vector clock system suffers other pitfalls and limitations both from practical and theoretical side. It is, for example, not able to cope with hidden channels [13]. This problem comes out when processes of a system can communicate through one or more channels which are distinct from the ones used by application messages as defined in Section 2 (hence the name "hidden channels"). Hidden channels can causally relate events in distinct processes and these relations are not reaveled by the vector clock system. A shared memory, a database, a shared file are examples of hidden channels. Moreover vector clocks can be difficult to adapt to dynamic systems such as systems of multithreaded processes system. From a theoretical side, a vector clock system suffers also limitations when we consider the computation model at a higher abstraction level where "computation atoms" are *intervals* (sets of events) instead of events. Sidebar 4 briefly addresses this issue.

Sidebar 4: Are Vector Clocks Always Able to Track Precedence Relations in Distributed Computations?

Vector clocks have been introduced to track causality (a precedence relation among events) in an *event-based* model (see Section 2). Let us now assume sequences of events, namely *intervals*, as the base abstraction for the model of the computation. In this case, each event belongs to an interval and each process is made of a sequence of non-empty intervals. Messages establish relations between intervals in distinct processes [a]. For example, the Figure shows an interval-based model of a computation where the interval A precedes B (due to message m'), B precedes C (due to message m) and then by transitivity A precedes C.



Such dependencies *cannot* be captured by a vector clock system as a part of them can be non-causal. As an example the dependency established between A and C is non-causal as message m has been sent from P_2 before the receipt of m' so this dependence cannot be tracked by a vector clock system. The dependency among intervals is usually called *zigzag* dependency [d], due to the presence of such non-causal relations. When each interval is made of exactly one event vector clocks are sufficient to track all precedence relations as they cannot be non-causal.

This interval-based model is used, for example, in the context of rollback recovery, to define *consistent global checkpoints* [b] where an interval is the set of events between two successive checkpoints, a local checkpoint is a dump of a local state of the process onto stable storage and a global checkpoint is a set of local checkpoints, one from each process. The reader can refer to the below references for more details.

[a] Baldoni R., Hélary J.-M. and Raynal M., Consistent Records in Asynchronous Computations, *Acta Informatica*, 35:441-455, 1998.

[b] Chandy K.M. and Lamport L., Distributed Snapshots: Determining Global States of Distributed Systems, *ACM Transactions on Computer Systems*, 3(1):63-75, 1985.

[c] Hélary J.-M., Netzer R.H.B. and Raynal M., Consistency Criteria for Distributed Checkpoints, *IEEE Transactions on Software Engineering*, 2(2):274-281, 1999.

[d] Netzer R.H.B. and Xu J., Necessary and sufficient conditions for Consistent Global Snapshots, *IEEE Transactions on Parallel and Distributed Systems*, 6(2):165-169, 1995.

References

[1] Baldoni R. Melideo G., k-dependency vectors: A scalable causality tracking protocol, *Technical Report, Dipartimento di Informatica e Sistemistica, #21.00, Univ. of Rome "La Sapienza"*, 2000.

[2] Birman K. and Joseph T., Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, 1987.

[3] Birman K., Schiper A., and Stephenson P., Lightweight Causal Order and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):282–314, 1991.

[4] Fidge C., Logical Time in Distributed Computing Systems, *IEEE Computer*, 24(8):28-33, 1991.

[5] Fidge C.J., Limitation of Vector Timestamps for Reconstructing Distributed Computations, *Information Processing Letters*, 68:87-91, 1998.

[6] Guo K., van Renesse R., Vogels W. and Birman K., *Hierarchical Message Stability Tracking Protocols*, Technical Report #1647, Dept Computer Science, Cornell University, 1997.

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

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

[9] Raynal M., Illustrating the Use of Vector Clocks in Property Detection: an Example and a Counter-Example. *Proc. the 5th Int. EUROPAR Conference*, Toulouse (France), Springer-Verlag LNCS Series 1685, p. 806-814, 1999.

[10] Raynal M., Schiper A., Toueg S., The Causal Ordering Abstraction and a Simple Way to Implement it. *Information Processing Letter* 39(6): 343-350 (1991).

[11] Terry D.B., Theimer M., Petersen K., Demers A.J., Spreitzer M., Hauser C.H., Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. *Proc. 15th ACM Symposium on Operating System Principles*, pp 173-183, 1995.

[12] Torres-Rojas F.J. and Ahamad M., Plausible Clocks: Constant Size Logical Clocks for Distributed Systems, *Proc. 10th Int. Workshop on Distributed Algorithms*, Springer-Verlag LNCS 1151 (Babaoglu O. and Marzullo K. Eds), pp. 71-88, 1996.

[13] Verissimo P., Real-Time Communication in Distributed Systems (second edition), S. Mullender Editor, ACM press, 1993.