IEEE distributed systems ONLINE
Expert-authored articles and resources

February 2002

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

Roberto Baldoni • **Universita di Roma, Italy**
Michel Raynal • **IRISA, France**

A distributed computation consists of a set of processes that cooperate to achieve a common goal. A main characteristic of these computations is that the processes do not already share a common global memory and that they communicate only by exchanging messages over a communication network. Moreover, message transfer delays are finite yet unpredictable. This computation model defines what is known as the *asynchronous distributed system model*, which includes systems that span large geographic areas and are subject to unpredictable loads.

A key concept of asynchronous distributed systems is *causality*. More precisely, given two events in a distributed computation, a crucial problem is knowing whether they are causally related. Could the occurrence of one event be a consequence of the other?

Processes produce message sendings, message receives, and internal events. Events that are not causally dependent are concurrent. Fidge [1] and Mattern [2] simultaneously and independently introduced vector clocks to let processes track causality (and concurrency) between the events they produce. A vector clock is an array of n integers (one entry per process), where the entry j counts the number of relevant events that process P_j produces. The timestamp of an event a process produced (or of the local state this event generated) is the current value of the corresponding process's vector clock. So, by associating vector timestamps with events or local states, we can safely decide whether two events or two local states are causally related (see the "A Historical View of Vector Clocks" sidebar).

Here, we present basic vector clock properties, mechanisms, and application examples to help distributed systems engineers solve the casuality problems they face.

A model of distributed execution

A distributed program is made up of *n* sequential local programs that, when executed, can communicate and synchronize only by exchanging messages. A distributed computation describes a distributed program's execution.

Executing a local program gives rise to a sequential process. Let P_1, P_2, ..., P_n be this finite set of processes. We assume that, at runtime, 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. Executing an internal, send, or receive statement produces an internal, send, or receive event. Let

$$e_i^x \left(x \geq 1\right)$$

be the *x*th event process P_i produces. 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 that a distributed computation produces.

This set is structured as a partial order by L. Lamport's "happened-before" relation,[1] denoted "—>" and defined as

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

e —>*f* means that event *e* can affect event *f*. Consequently, ¬(*e* —> *f*) means *e* cannot affect *f*. The partial order

$$\hat{H} = \left(H, \rightarrow\right)$$

constitutes a formal model of the distributed computation with which it is associated.

Figure 1 depicts a distributed computation, where black points denote events.

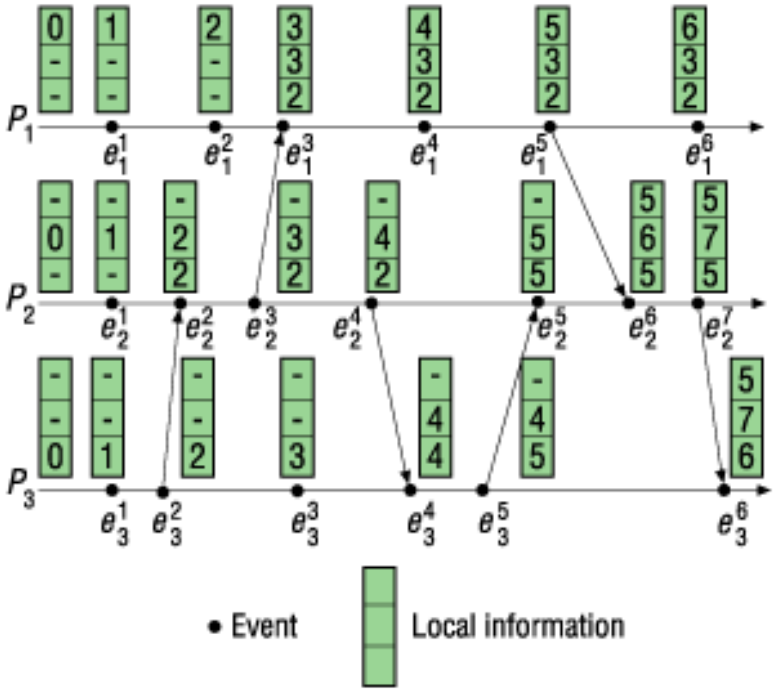

Figure 1. An example of a distributed computation.

Two events *e* and *f* are concurrent (or causally independent) if

$$\neg(e \to f) \wedge \neg(f \to e).$$

The causal past of event *e* is the (partially ordered) set of events *f* such that *f* —> *e*. Similarly, the causal future of event *e* is the (partially ordered) set of events *f* such that *e* —> *f*. For example, in Figure 1, we have

$$e_3^1 \to e_1^3, e_1^3 \parallel e_3^5$$

and the causal past of the event $e_2{}^2$ corresponds to the set

$$\left\{e_2^1, e_3^1, e_3^2\right\}.$$

Vector clocks: A causality tracking mechanism

A vector clock system is a mechanism that associates timestamps with events (local states) such that comparing two events' timestamps indicates whether those events (local states) are causally related (and, if they are, which one comes first).

In the time-stamping system, each process P_i has a vector of integers $VC_i[1..n]$ (initialized to [,0,...,0]) that is maintained as follows:

> (R1) Each time process P_i produces an event (send, receive, or internal), it increments its vector clock entry $VC_i[i]$ ($VC_i[i]$: = $VC_i[i]$+1) to indicate that 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 as
> $\forall x : VC_i[x] := \max\left(VC_i[x], m.VC[x]\right)$ (abbreviated as $VC_i := max(VC_i, m.VC)$.

Note that $VC_i[i]$ counts the number of events that P_i has so far produced. Moreover, for

$$i \neq j,$$

$VC_i[j]$ represents the number of events P_j produced that belong to the current causal past of P_i. When a process P_i produces an event *e,* it can associate with that event a vector timestamp whose value equals the current value of VC_i. Figure 1 shows vector timestamp values associated with events and local states. For example, $e_2^6.VC$ = (5,6,5).

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:[2,3]

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

where *e*.*VC* < *f*.*VC* is an abbreviation for

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

Let P_i be the process that produced e. This additional information lets us simplify the previous relation to [2,3]

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

(See the "An Efficient Implementation of Vector Clocks" sidebar.)

In our discussion of basic vector clock properties, we investigate three problems—causal broadcast, detecting message stability, and detecting an event pattern.

Causal broadcast

Birman and Joseph introduced the *causal broadcast* notion to reduce the asynchrony of communication channels as application processes perceive them.[4] It states that the order in which proccesses deliver messages to application processes cannot violate the precedence order (defined by the —> relation) of the corresponding broadcast events. More precisely, if two broadcast messages m and m' are such that *broadcast*(m) —> *broadcast*(m'), then any process must deliver m before m'. If the broadcasts of m and m' are concurrent, then proccesses are free to deliver m and m' in any order.

This means that when a proccess delivers a message m to a process, all messages whose broadcasts causally precede the broadcast of m have already been delivered to that process. The ISIS system first proposed such a communication abstraction.[4]

Several researchers have proposed vector clock-based implementations of causal broadcast, based on the following idea:[5,6] A receiving process P_i must delay delivering a message m until all the messages broadcast in the causal past of m are delivered to P_i. Consider Figure 2. When m' arrives at P_2, its delivery must be delayed because m' arrived at P_2 before m, and the sending of m causally precedes m'. To this end, each process P_i must manage a vector clock (VC_i) tracking its current knowledge on the number of messages that each process has sent.

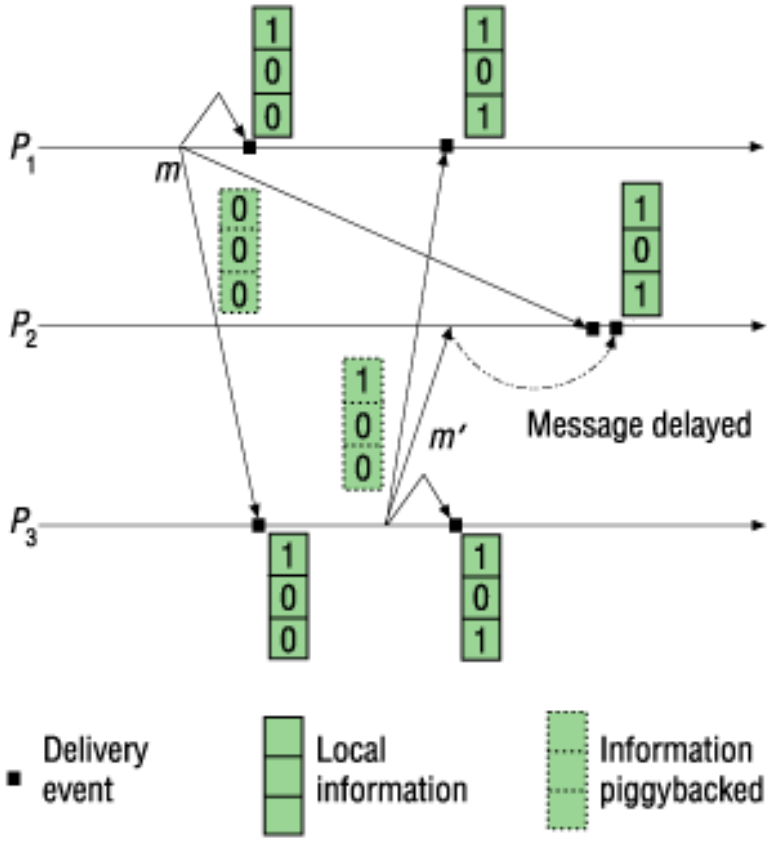

Figure 2. Causal delivery of broadcast messages.

Figure 3 describes a simple broadcast protocol (similar to one presented elsewhere[5]). Broadcast events are a computation's relevant events, and $VC_i[j]$ represents P_i's knowledge of the number of messages that P_j did broadcasts and delivered to P_i. Each message m piggybacks a vector timestamp $m.VC$, revealing how many messages each process has broadcast 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 are 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,

$$\left(\forall x \in \{1,\ldots,n\} : m.VC[x] \leq VC_i[x]\right).$$

Figure 3 describes the resulting causal broadcast protocol (vectors are initialized to [0,...,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.

Detecting message stability

Consider applications in which processes broadcast operations to all the others proccesses, and where each process must eventually receive the same set of operations that correct processes send. This problem abstracts the notions of *reliable broadcasting*[7] and *eventual consistency*,[8] 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 must buffer a copy of every message it sends or receives. If a process P_i fails, any process with a copy of a message m sent by P_i can forward m to any process P_j that detects it has not received m. This can induce a rapid growth of the buffer at each process with the risk of overflowing. Therefore, we need a policy that reduces buffer overflow occurrence. A simpler observation shows that buffering a message that has been delivered to all its intended destinations is not necessary. Such a message is called a *stable* message, and we can safely discard such messages from a process's local buffer.

A *message stability detection* protocol manages the process buffers. Such a protocol can be lazy (stability information piggybacks on application messages), use gossiping (periodic broadcast of control messages propogates stability information), or hybrid (both piggybacking and gossiping propogate stability information).

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, causal delivery is not ensured (that is, each message is delivered on receipt).

Broadcast events are the computation's relevant events. 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 P_i aware of

messages delivered to P_k. More precisely,

$$MC_i[k][\ell]\,(i \neq k, \ell)$$

represents P_i's knowledge of the number of messages that P_k delivered and P_l sent; $MC_i[i][i]$ represents the sequence number of the next message P_i sent. Hence, the minimum value over column j of MC_i —that is,

$$\min_{1 \leq x \leq n} \left(MC_i[x][j] \right)$$

—represents P_i's knowledge of the sequence number of the last message P_j sent that is stable.

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

Two operations update the local buffer ($buffer_i$): `deposit`(m) inserts a message m in the buffer and `discard`(m) removes m from the buffer. A process buffers a message immediately after it receives it and discards it as soon as it becomes stable (that is, when the process learns that all processes have delivered m). We can express the stability predicate for a message m using

$$m.VC[m.sender] \leq \min_{1 \leq x \leq 3} \left(MC_j[x][m.sender] \right),$$

where $m.VC[m.sender]$ represents the sequence number of m. Figure 4 describes the resulting protocol.

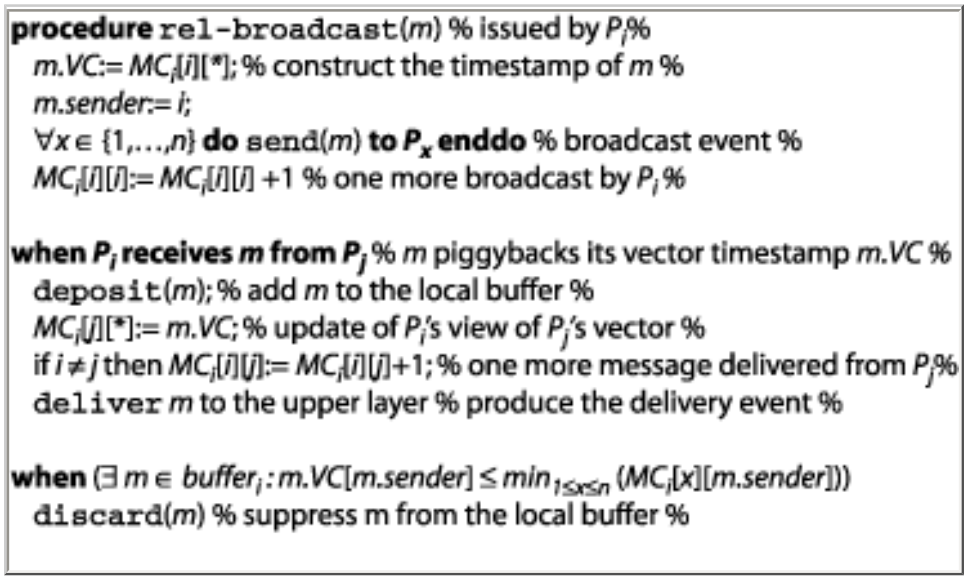

Figure 4. A simple lazy stability tracking protocol.

Figure 5 describes an example of running this stability tracking protocol. P_3 discards m immediately after receiving m', because

$$\min_{1\leq x\leq 3}\left(MC_j[x][m.sender]\right) = 0,$$

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''. To extend this protocol to handle causal delivery, we just need to add a delivery condition, similar to the one in Figure 3 and in the second clause of the protocol in Figure 4.

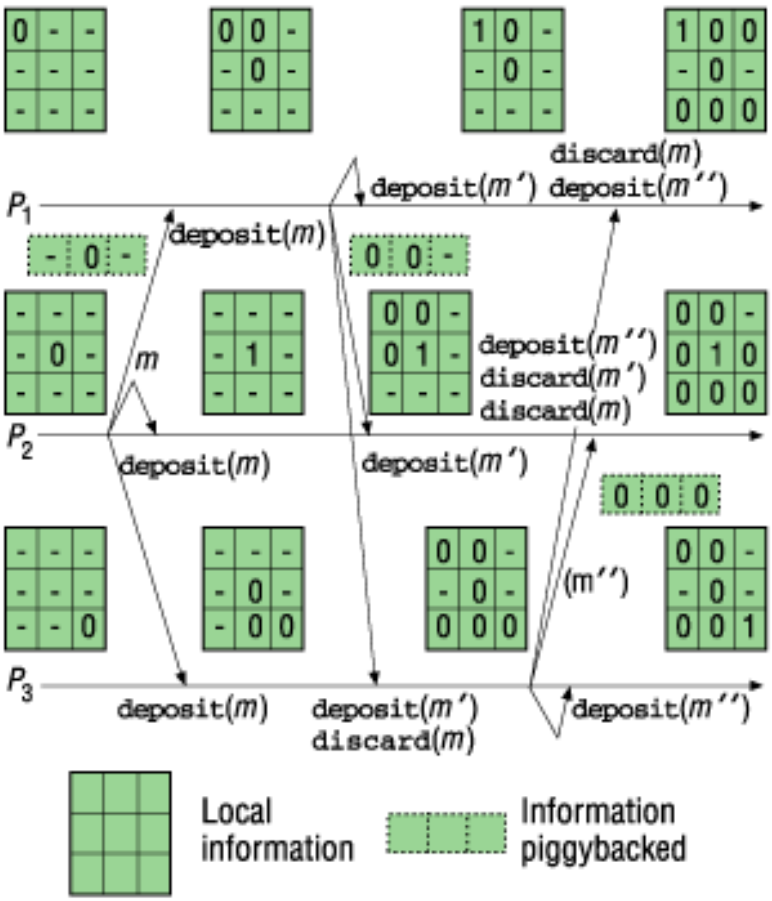

Figure 5. An example of lazy stability tracking.

Detecting an event pattern

Our causal broadcast example showed a simple use of vector clocks: each process managed a simple vector clock, and each message carried a vector timestamp. In our example of message stability detection, each message carried a vector timestamp, but each process had to manage a vector of vector clocks. Detecting an event pattern is a problem that comes from distributed debugging and shows that some problems require not only that each process manage a vector of vector clocks but also that each message carries a vector of vector clocks. In other words, solving causality-related problems is not always tractable with simple vector clocks.[9.10]

Consider a distributed execution that produces two types of internal events: some are tagged *black* and 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 of deciding if there is another black event *u*, such that

$s \rightarrow u \wedge u \rightarrow t.$

Let *black*(*e*) be a predicate indicating whether event *e* is black. More formally, given two events *s* and *t*, the problem consists of deciding if the following predicate $P(s,t)$ is true:

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

Figure 6 shows that vector clocks do not solve this problem. In these two executions, both events *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). However, the right execution satisfies the pattern, while the left one does not. (Note that *s* and *t* will have the same timestamp in both executions, even if vector clocks are incremented only on the occurrence of black events.)

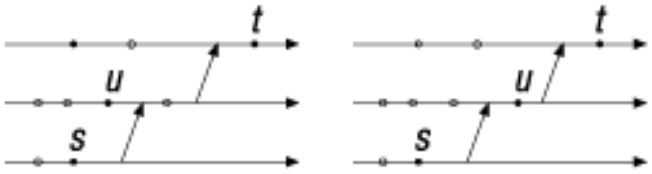

Figure 6. Recognizing a pattern.

Which clocks solve it?

For the predicate $P(s,t)$ to be true, a black event must exist in the causal past of *t*, which has *s* in its causal past. This problem concerns detecting causality, so it requires vector clocks. Moreover, two levels of predecessors appear in the predicate P. Tracking two levels of predecessors requires a vector of vector clocks.

The predicate $P(s,t)$ can be decomposed into two subpredicates $P_1(s,u,t)$ and $P_2(s,u,t)$:

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

with

$$\mathcal{P}_1(s,u,t) \equiv \big(black(s) \wedge black(u) \wedge black(t)\big)$$
$$\mathcal{P}_2(s,u,t) \equiv \big(s \rightarrow u \wedge u \rightarrow t\big).$$

$\mathcal{P}_1$ indicates that only the black events are relevant for the predicate detection. So, detecting $\mathcal{P}(s,t)$ requires only tracking black events. This means we can use vector clocks managed in the following way: A process P_i increments $VC_i[i]$ only when it produces a black event, and the other statements associated with vector clocks are left unchanged. (Actually, black events define the abstraction level at which the distributed computation must be observed to detect $\mathcal{P}$. All the other events—namely, the white events—are not relevant for detecting $\mathcal{P}$).

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, we only count black events in this vector timestamp)
> An array of vector timestamps *e*.*MC*[1..*n*], whose meaning is *e*.*MC*[*j*], contains the vector timestamp of the `last` black event of P_j that causally precedes *e*

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

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

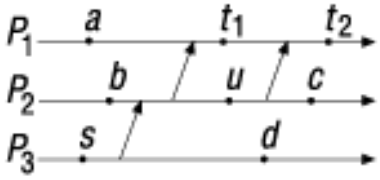

Figure 7. $\mathcal{P}(s,t_2)$ is true; $\mathcal{P}(s,t_1)$ is not.

Managing the clocks

Each process P_i has a vector clock $VC_i[1..n]$ and a vector of vector clocks $MC_i[1..n]$. Figure 8 describes how we manage those variables.

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

(S2) **when P_i executes a send event** (e = `send` m to P_j)
$m.VC := VC_i$; $m.MC := MC_i$; % construct the timestamps of m %
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 timestamps of last black predecessors %

Figure 8. Detection protocol for $P(s,t)$.

As before, the notation $VC := max(VC1, VC2)$ (statement S3 in Figure 8) is an abbreviation for

$$\forall k \in 1..n : VC[k] := \max\left(VC1[k], VC2[k]\right).$$

Moreover, 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 than (or equal to) the other. The result of $max(MC_i[k], m.MC[k])$ is this greatest timestamp. 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, we can deduce the vector clock VC_i from the diagonal of the matrix MC_i. This can reduce the number and size of data structures that processes manage and messages carry.

The pattern detection predicate

As we have seen, $P(s,t)$ is equivalent to

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

Note that, because the protocol considers only black events, the predicate P_1 is trivially satisfied by any triple of events. So, detecting $P(s,t)$ amounts to only detecting

$$\exists u : \mathcal{P}_2(s,u,t).$$

Given *s* and *t* with their timestamps (namely, *s*.*VC* and *s*.*MC* for *s*; *t*.*VC* and *t*.*MC* for *t*), we can state the predicate

$$\left(\exists u : \mathcal{P}_2(s,u,t)\right) \equiv \left(\exists u : s \rightarrow u \rightarrow t\right)$$

in a more operational way using vector timestamps:

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

If such an event *u* does exist, some process P_k produced it, and it 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 into

$$\left(\exists u : s \rightarrow u \rightarrow t\right) \equiv \left(\exists k : s.VC < t.MC[k] < t.VC\right).$$

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

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

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

> The vector timestamp of *u* (*u*.*VC*) is less than or equal to *t*.*MC*[*k*].
> The event *u* belongs to the causal past of *t* (because *t*.*MC*[*k*] is less than *t*.*VC*).
> The event *u* belongs to the causal future of *s* (because *s*.*VC* is less than *u*.*VC* is less than or equal to *t*.*MC*[*k*]).

So, when the system is equipped with the vector clock system we described, we can evaluate the predicate $P(s,t)$ using a simple test—namely,

$$\exists k : s.VC < t.MC[k].$$

Moreover, when we know the identity of the process (say P_i) that produced *s*, we can simplify this test. Using the relation R (presented earlier), the test becomes

$$\exists k : s.VC[i] < t.MC[k][i].$$

Bounded vector clocks

A vector clock system's main drawback is its inability to face scalability problems. To fully capture the causality relation among the events that a distributed computation's processes produce, a vector clock system requires vectors of size *n* (*n* being the number of processes). To circumvent this problem, researchers have introduced two types of bounded vector clocks (whose size is bounded by a constant that is less than *n*): *approximate* [11] and *k-dependency*[12]vector clocks.

Approximate vector clocks use a space-folding approach. We can use this approach when we are only interested in never missing causality between related events (so, we accept that we perceive two events as ordered when they are actually concurrent). *k*-dependency vectors involve a time-folding approach in which an event's bounded timestamp provides causal dependencies that, when recursively exploited, reconstruct the event's vector timestamp.

Approximate vector clocks

In some applications, we are only interested in approximating the causality relation such that

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

(let *e*.*TS* be the timestamp associated with *e*). Such a timestamping never violates causality in the sense that, from *e*.*TS* < *f*.*TS*, we can safely conclude ¬(*f* —> *e*). If we optimistically conclude *e* —>*f*, then we can be wrong, because it is possible that *e* and *f* are not causally related. That is why concluding *e* —>*f* from *e*.*TS* < *f*.*TS* constitutes an approximation of the causality relation.

F.J. Torres and M. Ahamad introduced approximate vector clocks. They provide a simple mechanism that associates approximate vector timestamps with events. Consider vector clocks whose size is bounded by a constant (with $k < n$). So, $TS_i[1..k]$ is P_i's approximate vector clock. Moreover, let f_k be a deterministic function from {1,...,*n*} to {1,...,*k*}. Given a process identity *i*, this function associates with it the entry $f_k(i)$ of all vector clocks $TS[1..k]$—that is, $TS[f_k(i)]$.

Implementing such a time-stamping system is similar to the one described earlier. Each process P_i manages its vector clock $TS_i[1..k]$, initialized to (0,..,0) in the following way:

> (R1) Each time P_i produces a send, receive, or internal event, it 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 as
> $$\forall x \in \{1,\ldots,k\} : TS_i[x] := \max\left(TS_i[x], m.TS[x]\right).$$

Combined with the function f_k, these rules ensure that all processes P_i share the *x*th entry of any vector clock, 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[11]

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

More generally, if $k = n$ and

$$\forall i : f_k(i) = i,$$

then we get classic vector clocks that track full causality. In that case, the vector clock system's entry i is private to P_i in the sense that 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 Lamport's scalar clock system.[1] This scalar clock system is well known for its property

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

Many applications consider the timestamp of an event e that P_i produced as the pair ($e.TS$,i). This provides an easy way to totally order (without violating causal relations) the set of all the events a distributed computation produces. This is the famous total order relation Lamport defined[1]—namely, if P_i and P_j produce e and f, respectively, e is ordered before f if

$$(e.TS < f.TS) \vee \big((e.TS = f.TS) \wedge (i < j)\big).$$

Also, scalar clocks detect some concurrent events—more precisely,

$$(e.TS = f.TS) \Rightarrow (e \text{ and } f \text{ are concurrent}).$$

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[11] show that with $n = 100$ and $2 < k < 5$, the percentage of situations in which e —> f is concluded from $e.TS < f.TS$ (while e and f are concurrent) is less than 10 percent.

Dependency vectors

Given two events e and f of a distributed computation such that $e.TS < f.TS$, approximate

vector clocks can't conclude whether $e \longrightarrow f$ or $e \parallel f$. For such a pair, they can only answer $\neg(f \longrightarrow e)$. So, an important question is,"Does a vector clock system exist with a bounded number of entries, from which we can reconstruct the causality relation—that is, conclude (maybe after some computation) that $e \longrightarrow f$, $f \longrightarrow e$, or $e \parallel f$?" Dependency vectors answer this question.

The following behavior characterizes a k-dependency vector clock system. Each process P_i has a vector clock $DV_i[1..n]$, which is initialized to (0,...,0) and managed in the following way: [12]

> (R1) Each time P_i produces an event, it 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 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 as
> $$\forall x \text{ such that } (x, DV[x]) \in m.TS : DV_i[x] := \max = (DV_i[x], DV[x]).$$

A k-dependency vector clock system provides each process with an 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). Choosing the other $k - 1$ values is left to the user. A good heuristics consists in choosing the last modified $k - 1$ entries of DV_i.[12] 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's assume that P_i produced e. 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 $e \longrightarrow f$ while $e.DV[i] > f.DV[i]$. But, unlike approximate vector clocks, k-dependency vectors can (using additional computation) reconstruct the causality relation (see the "Reconstructing Vector Timestamps from Dependency Timestamps" sidebar). Of course, according to the problem to be solved, we can use k-dependency vectors and approximate vectors simultaneously.

The concept of causality among events is fundamental to designing and analyzing

distributed programs. However, a vector clock system suffers from limitations other than scalability. For example, the system can't cope with hidden channels.[13] This problem arises when a system's processes can communicate through one or more channels that are distinct from the ones application messages use. Hidden channels can causally relate events in distinct processes; the vector clock system doesn't reveal these relations. Shared memory, a database, and 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. A vector clock system also suffers limitations when we consider the computation model at a higher abstraction level where computation atoms are *intervals* (sets of events) instead of events. The "Can Vector Clocks Always Track Precedence Relations?" sidebar briefly addresses this issue.

References

1. L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System," *Comm. ACM*, vol. 21, no. 7, July 1978, pp. 558-565.
2. C. Fidge, "Logical Time in Distributed Computing Systems," *Computer*, vol. 24, no. 8, Aug. 1991, pp. 28-33.
3. F. Mattern, "Virtual Time and Global States of Distributed Systems," *Proc. Parallel and Distributed Algorithms Conf.*, Elsevier Science, 1988, pp. 215–226.
4. K. Birman and T. Joseph, "Reliable Communication in the Presence of Failures,"*ACM Trans. Computer Systems*, vol. 5, no. 1, Feb. 1987, pp. 47–76.
5. K. Birman, A. Schiper, and P. Stephenson, "Lightweight Causal Order and Atomic Group Multicast," *ACM Trans. Computer Systems*, vol. 9, no. 3, Aug. 1991, pp. 282–314.
6. M. Raynal, A. Schiper, and S. Toueg, "The Causal Ordering Abstraction and a Simple Way to Implement it," *Information Processing Letter*, vol. 39, no. 6, Sept. 1991, pp. 343–350.
7. K. Guo et al., *Hierarchical Message Stability Tracking Protocols*, tech. report 1647, Dept. Computer Science, Cornell Univ., 1997.
8. D.B. Terry et al., "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System," *Proc. 15th ACM Symp. Operating System Principles*, ACM Press, New York, 1995, pp. 173–183.
9. C.J. Fidge, "Limitation of Vector Timestamps for Reconstructing Distributed Computations," *Information Processing Letters*, vol. 68, no. 2, Oct. 1998, pp. 87-91.
10. M. Raynal, "Illustrating the Use of Vector Clocks in Property Detection: an Example and a Counter-Example," *Proc. 5th Int. EUROPAR Conf.*, Springer-Verlag, New York, 1999, pp. 806-814.
11. F.J. Torres-Rojas and M. Ahamad, "Plausible Clocks: Constant Size Logical Clocks for Distributed Systems," *Proc. 10th Int'l Workshop Distributed Algorithms*, Springer-Verlag, New York, 1996, pp. 71–88.
12. R. Baldoni and G. Melideo, *Tradeoffs in Message Overhead versus Detection Tim in Causality Tracking*, tech. report 06-01, Dipartimento di Informatica e Sistemistica,

Univ. of Rome, 2000.
13. P. Verissimo, *Real-Time Communication in Distributed Systems* (second edition), ACM Press, New York, 1993.

Roberto Baldoni is an associate professor at the school of engineering of the University of Rome, La Sapienza. He has published more than seventy scientific papers in the fields of distributed computing, dependable middleware, and communication protocols. He received a degree in electronic engineering and a PhD in computer science from the University of Rome, La Sapienza. Contact him at baldoni@dis.uniroma1.it; www.dis.uniroma1.it/~baldoni.

Michel Raynal is a professor of computer science at the University of Rennes, France. At IRISA (CNRS-INRIA-University joint computing laboratory located in Rennes), he leads the ADP
(Distributed Algorithms and Protocols) research group. His main research interest lies in the fundamental concepts, principles, and mechanisms that underly the design and construction of distributed systems. He is currently studying the causality concept and agreement problems.
He is also involved in the implementation of reliable communication primitives, the consistency of distributed data, the design and use of checkpointing protocols, and the set of problems
that can be solved on top of a consensus building block. Contact him at Michel.Raynal@irisa.fr.

A Historical View of Vector Clocks

Researchers have used vector clocks empirically as an ad hoc device to solve specific problems before capturing and defining them as a concept. For example, D.S. Parker and colleagues used similar vectors to detect mutual inconsistencies of the copies of replicated data.[1] B. Ladin and R. Liskov used them to detect obsolete data,[2] and M. Raynal used them to prevent drift among a set of *n* logical scalar clocks.[3] D.B. Johnson and W. Zwaenepoel [4] and R.E. Strom and S. Yemini [5] used them to track causal dependencies between events in their checkpointing protocols. F. Schmuck used them to implement efficient causal broadcast in asynchronous distributed systems.[6]

C.J. Fidge [7] and F. Mattern [8] simultaneously and independently introduced vector clocks as a concept, with their basic properties, in 1988. These works have clearly defined the concept, studied and proved fundamental properties associated with vector clocks, and promoted them as a first-class mechanism to study and solve causality-related problems in distributed systems.

References

1. D.S. Parker et al., "Detection of Mutual Inconsistency in Distributed Systems," *IEEE Trans. Software Eng.*, vol. SE9, no. 3, 1983, pp. 240–246.
2. B. Liskov and R. Ladin, "Highly Available Distributed Services and Fault-Tolerant Distributed Garbage Collection," *Proc. 5th ACM Symp. Principles of Distributed Computing*, ACM Press, New York, 1986, pp. 29–39.
3. M. Raynal, "A Distributed Algorithm to Prevent Mutual Drift Between *n* Logical Clocks," *Information Processing Letters*, vol. 24, no. 3, Feb. 1987, pp. 199–202.
4. D.B. Johnson and W. Zwaenepoel, "Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing," *Proc. 7th ACM Symp. Principles of Distributed Computing*, ACM Press, New York, 1988, pp. 171–181.
5. R.E. Strom and S. Yemini, "Optimistic Recovery in Distributed Systems," *ACM Trans. Computer Systems*, vol. 3, no. 3, Aug. 1985, pp. 204–226.
6. F. Schmuck, *The Use of Efficient Broadcast in Asynchronous Distributed Systems*, doctoral dissertation, Dept. Computer Science, Cornell University, 1988, p. 124.
7. C.J. Fidge, "Timestamp in Message Passing Systems that Preserves Partial Ordering," *Proc. 11th Australian Computing Conf.*, 1988, pp. 56–66.
8. F. Mattern, "Virtual Time and Global States of Distributed Systems," *Proc. Parallel and Distributed Algorithms Conf.*, Elsevier Science, 1988, pp. 215–226.

An Efficient Implementation of Vector Clocks

A drawback of a vector clock system is that each message must carry an array of *n* integers (where *n* is the size of the application—the total number of processes).[1]

To face this problem, M. Singhal and A. Kshemkalyani proposed a simple technique that, for the average case, reduces the size of the vector timestamps that messages piggyback.[2] This technique is based on the empirical observation that few processes are likely to frequently interact (this is especially true when the number of processes is high). Between two successive sending events from process P_i to process P_j, only a few of the vector clock's entries are expected to change.

In such a case, there is no point in attaching to each outgoing message from P_i to P_j a whole vector clock. It suffices to piggyback only the information relative to the entries that changed. This corresponds to a set of tuples (*proc_id*, *VC*[*proc_id*]). Therefore, we expect this technique will save communication bandwidth at the cost of local memory overhead,

because a process must keep track of the last values sent to each process (one vector for each process) to select the set of tuples to piggyback on each message.

Figure A shows the progress of vector clocks using Singhal-Kshemkalyani's technique. Note that for this technique to work, communication channels must be first-in first-out.

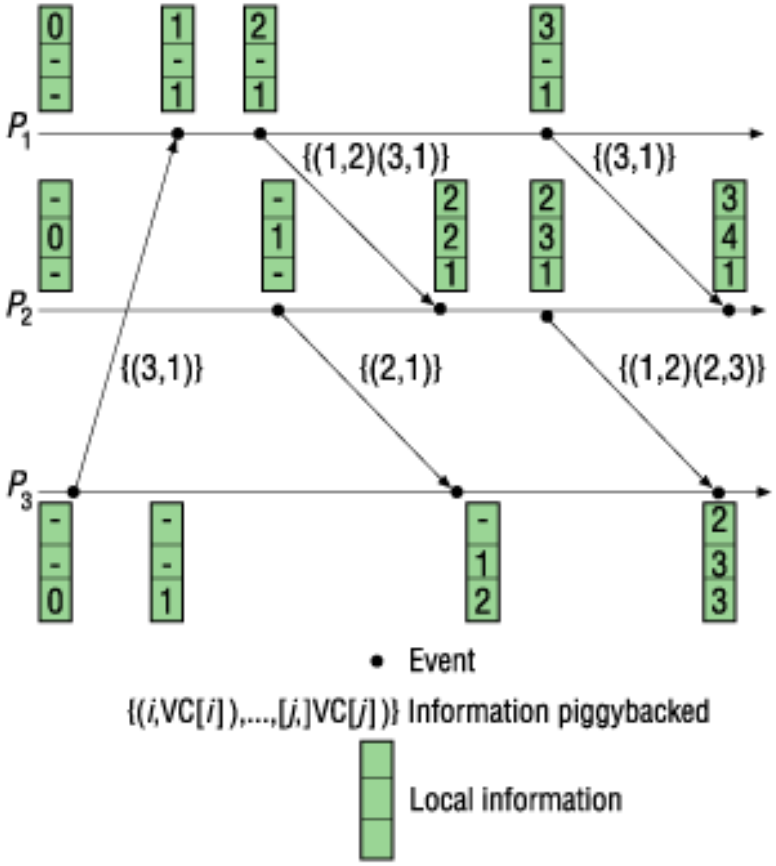

Figure A. The progress of vector clocks using Signhal-Kshemkalyani's technique.

Another practical problem that vector clock systems must face is the overflow of their vector entries. A general solution to solve this problem appears elsewhere.[3,4]

References

1. B. Charron-Bost, "Concerning the Size of Logical Clocks in Distributed Systems," *Information Processing Letters*, vol. 39, no. 12, July 1992, pp. 11-16.
2. M. Singhal and A. Kshemkalyani, "An Efficient Implementation of Vector Clocks," *Information Processing Letters*, vol. 43, no. 10, Aug. 1992, pp. 47-52.

3. L.-Y. Yen and T.-L. Huang, "Resetting Vector Clocks in Distributed Systems," *J. Parallel and Distributed Systems*, vol. 43, no. 1, May 1997, pp. 15-20.
4. R. Baldoni, "A Positive Acknowledgment Protocol for Causal Broadcasting," *IEEE Trans. Computers*, vol. 47, no. 12, Dec. 1998, pp. 1341-1350.

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 that has not yet been deposited in the corresponding queue, it waits until it has received the required information. The algorithm the checker process executes to compute the vector timestamp associated with an event *e* operates iteratively (defined in Figure B). The function $\max(V_1,V_2)$ is defined as

$$\forall \ell \in \{1..n\} : \max(V_1V_2)[\ell] = \max\left(V_1[\ell], V_2[\ell]\right).$$

```
procedure Vector_Timestamp(var e : event)
    e.V := e.DV;
    repeat
        old_V := e.V;
        for each x ∈ {1,...,n} do e.V := max(e.V, e_x^old_V[x].DV) enddo
    until (e.V = old_V);
let e.VC = e.V;
```

Figure B. The algorithm the checker process executes to compute the vector timestamp associated with an event *e*.

The **repeat** statement computes the causal precedence relation's transitive closure. The inner loop moves the current dependency timestamp of *e* forward 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*, so *e*.*V* is the vector timestamp of *e*—namely, *e*.*VC*.

Several researchers have investigated such a reconstruction of a vector timestamp from dependency vectors.[1-3] Others consider dependency vectors with $k = 1$.[4,5] They use them to define consistent global checkpoints and causal breakpoints, respectively.

References

1. P. Baldy et al., "Efficient Reconstruction of the Causal Relationship in Distributed Systems," *Proc. 1st Canada-France Conf. Parallel and Distributed Computing*, Springer Verlag, New York, 1994, pp. 101-113.
2. V.K. Garg, *Principles of Distributed Systems*, Kluwer Academic Press, Dordrecht, Netherlands, 1996.
3. R. Schwarz and F. Mattern, "Detecting Causal Relationship in Distributed Computations: In Search of the Holy Grail," *Distributed Computing*, vol. 7, no. 3, 1994, pp. 149-174.
4. R. Baldoni et al., "Direct Dependency-Based Determination of Consistent Global Checkpoints," *Int'l J. Computer Systems Science and Eng.*,vol. 16, no. 1, Jan. 1999.
5. J. Fowler and W. Zwaenepoel, "Causal Distributed Breakpoints," *Proc. 10th Int'l IEEE Conf. Distributed Computing Systems*, IEEE Press, Piscataway, N.J., 1990, pp. 134-141.

Can Vector Clocks Always Track Precedence Relations?

Assume sequences of events, namely *intervals*, as the base abstraction for the computation's model. In this case, each event belongs to an interval and each process is made of a sequence of nonempty intervals. Messages establish relations between intervals in distinct processes.[1] For example, Figure C 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*.

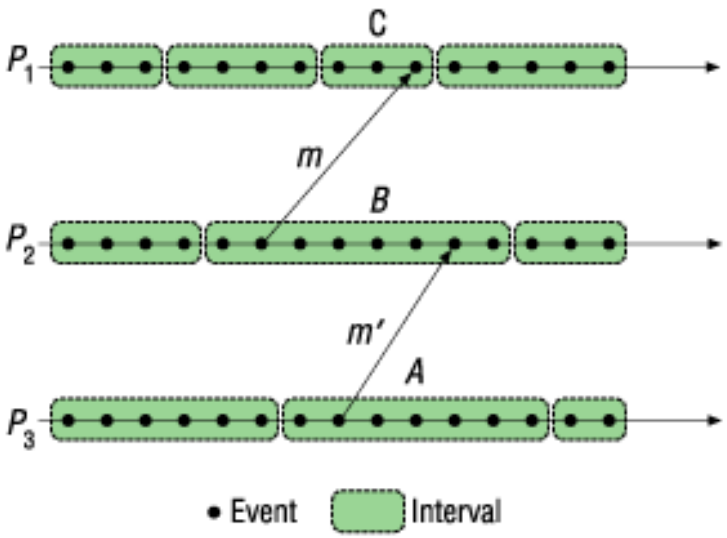

Figure C. An interval-based model of a computation.

A vector clock system cannot capture such dependencies, because a part of them can be noncausal. For example, the dependency established between *A* and *C* is noncausal, because message *m* was sent from P_2 before the receipt of *m´* thus a vector systems can't track this dependence. The dependency among intervals is usually called *zigzag* dependency [2], due to the presence of such noncausal relations. When each interval is made of exactly one event, vector clocks are sufficient to track all precedence relations because they cannot be noncausal.

This interval-based model is used, for example, in the context of rollback recovery. It defines *consistent global checkpoints*[3], 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.

References

1. R. Baldoni, J.-M. Helary, and M. Raynal, "Consistent Records in Asynchronous Computations," *Acta Informatica*, vol. 35, no. 6, June 1998, pp. 441–455.
2. R.H.B. Netzer and J. Xu, "Necessary and Sufficient Conditions for Consistent Global Snapshots," *IEEE Trans. Parallel and Distributed Systems*, vol. 6, no. 2, Feb. 1995, pp. 165–169.
3. K.M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Trans. Computer Systems*, vol. 3, no. 1, Feb. 1985, pp. 63–75. 1985.