$\langle clock_i[x], x\rangle$. Consequently, as m_x is not yet to_delivered (assumption), it follows that $(m_y, ts(m_y))$ cannot be the pair with the smallest timestamp in $pending_i$. This contradicts the initial assumption. It follows that the messages that have been to_broadcast and are to_delivered, are to_delivered at each process in the same total order (as defined by their timestamp).

The proof that the total order on timestamps respects the causal precedence order on to_broadcast messages follows from the two following observations.

- A process p_i increases it local clock $clock_i[i]$ each time it invokes to_broadcast(). Hence, if p_i to_broadcasts m before m', we have $ts(m) < ts(m')$.
- Due to line 11, it follows that, after it has received a message TOBC$(m, \langle d, j\rangle)$, we have $clock_i[i] > d$. It follows that, if later p_i to_broadcasts a message m', we necessarily have $ts(m) < ts(m')$. □

7.2 Vector Time

As we have seen, the domain of linear time is the set of non-negative integers, and a set of n logical clocks (one per process) allows us to associate dates with events, in such a way that the dates respect the causal precedence relation $\stackrel{ev}{\longrightarrow}$. However, as we have seen, except for those events that have the same date, linear dates do not allow us to determine with certainty if one event belongs to the causes of another one. The aim of vector time (which is implemented by vector clocks) is to solve this issue. Vector time, and its capture by vector clocks, was simultaneously and independently proposed in 1988 by C.J. Fidge, F. Mattern, and F. Schmuck.

7.2.1 Vector Time and Vector Clocks

Vector Time: Definition A vector clock system is a mechanism that is able to associate dates with events (or local states) in such a way that the comparison of their dates allows us to determine if the corresponding events are causally related or not, and if they are which one belongs to the cause of the other. Vector time is the time notion captured by vector clocks.

More precisely, let $date(e)$ be the date associated with an event e. The aim of a vector clock system is to provide us with a time domain in which any two dates either can be compared ($<, >$), or are incomparable (denoted $||$), in such a way that the following properties are satisfied

- $\forall\, e_1, e_2$: $(e_1 \stackrel{ev}{\longrightarrow} e_2) \Leftrightarrow date(e_1) < date(e_2)$, and
- $\forall\, e_1, e_2$: $(e_1 || e_2) \Leftrightarrow date(e_1) \,||\, date(e_2)$ (the dates cannot be compared).

This means that the dating system has to be in perfect agreement with the causal precedence relation $\stackrel{ev}{\longrightarrow}$. To attain this goal, vector time considers that the time domain is made up of all the integer vectors of size n (the number of processes).

```
when producing an internal event e do
(1)  vc_i[i] ← vc_i[i] + 1;
(2)  Produce event e.  % The date of e is vc_i[1..n].

when sending MSG(m) to p_j do
(3)  vc_i[i] ← vc_i[i] + 1;  % vc_i[1..n] is the sending date of the message.
(4)  send MSG(m, vc_i[1..n]) to p_j.

when MSG(m, vc) is received from p_j do
(5)  vc_i[i] ← vc_i[i] + 1;
(6)  vc_i[1..n] ← max(vc_i[1..n], vc[1..n]).  % vc_i[1..n] is the date of the receive event.
```

Fig. 7.9 Implementation of a vector clock system (code for process p_i)

Vector Clock: Definition To implement vector time, each process p_i manages a vector of non-negative integers $vc_i[1..n]$, initialized to $[0, \ldots, 0]$. This vector is such that:

- $vc_i[i]$ counts the number of events produced by p_i, and
- $vc_i[j]$, $j \neq i$, counts the number of events produced by p_j, as known by p_i.

More formally, let e be an event produced by some process p_i. Just after p_i has produced e we have (where $1(k,i) = 1$ if p_k is p_i, and $1(k,i) = 0$ otherwise):

$$vc_i[k] = \big|\big\{f \,\big|\, (f \text{ has been produced by } p_k) \wedge (f \xrightarrow{ev} e)\big\}\big| + 1(k,i).$$

Hence, $vc_i[k]$ is the number of events produced by p_k in the causal past of the event e. The term $1(k,i)$ is to count the event e, which has been produced by p_i. The value of $vc_i[1..n]$ is the vector date of the event e.

Vector Clock: Algorithm The algorithm implementing the vector clock system has exactly the same structure as the one for linear time (Fig. 7.1). The difference lies in the fact that only the entry i of the local vector clock of p_i (i.e., $vc_i[i]$) is increased each time it produces a new event, and each message m piggybacks the current value of the vector time, which defines the sending time of m. This value allows the receiver to update its local vector clock, so that the date of the receive event is after both the date of the sending event associated with m and the immediately preceding local event produced by the receiver process. The operator max() on integers is extended to vectors as follows (line 5):

$$\begin{aligned}&\max(v1, v2)\\ &\quad = \big[\max\big(v1[1], v2[1]\big), \ldots, \max\big(v1[j], v2[j]\big), \ldots, \max\big(v1[n], v2[n]\big)\big].\end{aligned}$$

Let us observe that, for any pair (i,k), it follows directly from the vector clock algorithm that (a) $vc_i[k]$ never decreases, and (b) at any time, $vc_i[k] \leq vc_k[k]$.

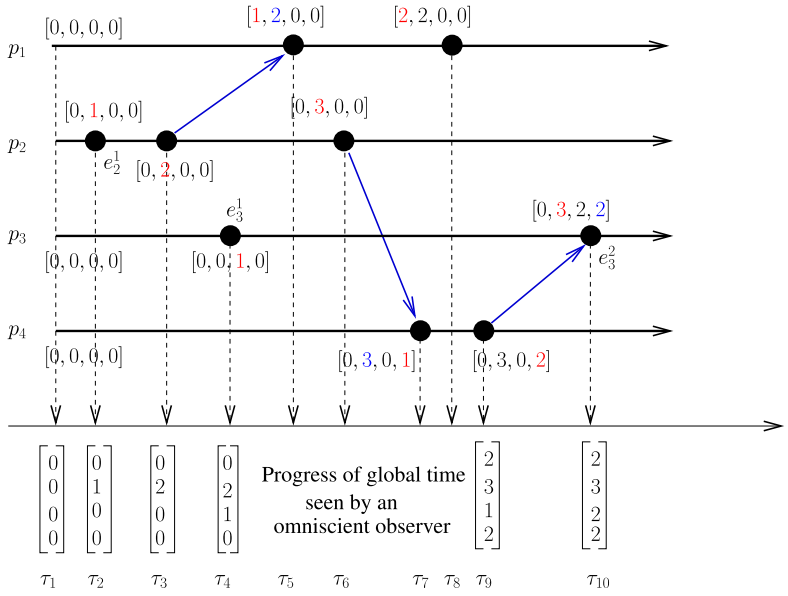

Fig. 7.10 Time propagation in a vector clock system

Example of Time Propagation An example of an execution of the previous algorithm is depicted in Fig. 7.10. The four local vector clocks are initialized to [0, 0, 0, 0]. Then, p_2 produces an internal event dated [0, 1, 0, 0]. Its next event is the sending of a message to p_1, dated [0, 2, 0, 0]. The reception of this message by p_1 is its first event, and consequently the reception of this message is dated [1, 2, 0, 0], etc.

The bottom of the figure shows what could be seen by an omniscient external observer. The global time seen by this observer is defined as follows: At any time τ_x, it sees how many events have been produced by each process up to τ_x. Hence, the current value of this global time is $[vc_1[1], vc_2[2], vc_3[3], vc_4[4]]$. Initially, no process has yet produced events, and the observer sees the global vector time [0, 0, 0, 0] at external time τ_1. Then, at τ_3, it sees the sending by p_2 of a message to p_1 at the global time [0, 2, 0, 0], etc. Some global time instants are indicated on the figure. The value of the last global time seen by the omniscient observer is [2, 3, 2, 2].

The global time increases when events are produced. As the observer is omniscient, this is captured by the fact that the global time values it sees increase in the sense that, if we consider any two global time vectors $GD1$ followed by $GD2$, we have $(\forall k : GD1[k] \leq GD2[k]) \wedge (\exists\, k : GD1[k] < GD2[k])$.

Notation Given two vectors $vc1$ and $vc2$, both of size n, we have

- $vc1 \leq vc2 \stackrel{def}{=} (\forall\, k \in \{1, \ldots, n\} : vc1[k] \leq vc2[k])$.
- $vc1 < vc2 \stackrel{def}{=} (vc1 \leq vc2) \wedge (vc1 \neq vc2)$.

- $vc1||vc2 \stackrel{def}{=} \neg(vc1 \leq vc2) \wedge \neg(vc1 \leq vc2)$.

When considering Fig. 7.10, we have $[0, 2, 0, 0] < [0, 3, 2, 2]$, and $[0, 2, 0, 0] \,||\, [0, 0, 1, 0]$.

7.2.2 Vector Clock Properties

The following theorem characterizes the power of vector clocks.

Theorem 5 *Let $e.vc$ be the vector date associated with event e, by the algorithm of Fig. 7.9. These dates are such that, for any two distinct events e_1 and e_2 we have* (a) $(e_1 \stackrel{ev}{\longrightarrow} e_2) \Leftrightarrow (e_1.vc < e_2.vc)$, *and* (b) $(e_1||e_2) \Leftrightarrow (e_1.vc \,||\, e_2.vc)$.

Proof The proof is made up of three cases.

- $(e_1 \stackrel{ev}{\longrightarrow} e_2) \Rightarrow (e_1.vc < e_2.vc)$. This case follows directly from the fact that the vector time increases along all causal paths (lines 1 and 3–6).
- $(e_1.vc < e_2.vc) \Rightarrow (e_1 \stackrel{ev}{\longrightarrow} e_2)$. Let p_i be the process that produced event e_1. We have $e_1.vc < e_2.vc \Rightarrow e_1.vc[i] \leq e_2.vc[i]$. Let us observe that only process p_i can entail an increase of the entry i of any vector (if p_i no longer increases $vc_i[i] = a$ at line 1, no process p_j can be such that $vc_j[i] > a$). We conclude from this observation, the code of the algorithm (vector time increases only along causal paths), and $e_1.vc[i] \leq e_2.vc[i]$ that there is a causal path from e_1 to e_2.
- $(e_1||e_2) \Leftrightarrow (e_1.vc \,||\, e_2.vc)$. By definition we have $(e_1 \,||\, e_2) = (\neg(e_1 \stackrel{ev}{\longrightarrow} e_2) \wedge \neg(e_2 \stackrel{ev}{\longrightarrow} e_1))$. It follows from the previous items that $\neg(e_1 \stackrel{ev}{\longrightarrow} e_2) \Leftrightarrow \neg(e_1.vc \leq e_2.vc)$, i.e., $\neg(e_1 \stackrel{ev}{\longrightarrow} e_2) \Leftrightarrow (e_1.vc > e_2.vc) \vee (e_1.vc \,||\, e_2.vc)$. Similarly, $\neg(e_2 \stackrel{ev}{\longrightarrow} e_1) \Leftrightarrow (e_2.vc > e_1.vc) \vee (e_2.vc \,||\, e_1.vc)$. Combining the previous observations, and observing that we cannot have simultaneously $(e_1.vc > e_2.vc) \wedge (e_2.vc > e_1.vc)$, we obtain $(e_1 \,||\, e_2) \Leftrightarrow (e_1.vc \,||\, e_2.vc)$. □

The next corollary follows directly from the previous theorem.

Corollary 1 *Given the dates of two events, determining if these events are causally related or not can require up to n comparisons of integers.*

Reducing the Cost of Comparing Two Vector Dates As the cost of comparing two dates is $O(n)$, an important question is the following: Is it possible to add control information to a date in order to reduce the cost of their comparison? If the events are produced by the same process p_i, a simple comparison of the ith entry of their vector dates allows us to conclude. More generally, given an event e produced by a process p_i, let us associate with e a timestamp defined as the pair $\langle e.vc, i \rangle$. We have then the following theorem from which it follows that, thanks to the knowledge of the process that produced an event, the cost of deciding if two events are or not causally related is reduced to two comparisons of integers.

Theorem 6 *Let e_1 and e_2 be events timestamped $\langle e_1.vc, i\rangle$ and $\langle e_2.vc, j\rangle$, respectively, with $i \neq j$. We have $(e_1 \xrightarrow{ev} e_2) \Leftrightarrow (e_1.vc[i] \leq e_2.vc[i])$, and $(e_1 \,||\, e_2) \Leftrightarrow ((e_1.vc[i] > e_2.vc[i]) \wedge (e_2.vc[j] > e_1.vc[j]))$.*

Proof Let us first observe that time increases only along causal paths, and only the process that produced an event entails an increase of the corresponding entry in a vector clock (Observation O). The proof considers each case separately.

- $(e_1 \xrightarrow{ev} e_2) \Leftrightarrow (e_1.vc[i] \leq e_2.vc[i])$.
  If $e_1 \xrightarrow{ev} e_2$, there is a causal path from e_1 to e_2, and we have $e_1.vc \leq e_2.vc$ (Theorem 6), from which $e_1.vc[i] \leq e_2.vc[i]$ follows.
  If $e_1.vc[i] \leq e_2.vc[i]$, it follows from observation O that there is a causal path from e_1 to e_2.
- $(e_1 \,||\, e_2) \Leftrightarrow ((e_1.vc[i] > e_2.vc[i]) \wedge (e_2.vc[j] > e_1.vc[j]))$.
  As, at any time, we have $vc_j[i] \leq vc_i[i]$, p_i increases $vc_i[i]$ when it produces e_1, it follows from the fact that there is no causal path from e_1 to e_2 and observation O that $e_1.vc[i] > e_2.vc[i]$. The same applies to $e_2.vc[j]$ with respect to $e_1.vc$.
  In the other direction, we conclude from $e_1.vc[i] > e_2.vc[i]$ that there is no causal path from e_1 to e_2 (otherwise we would have $e_1.vc[i] \leq e_2.vc[i]$). Similarly, $e_2.vc[j] > e_1.vc[j]$ implies that there is no causal path from e_2 to e_1. □

To illustrate this theorem, let us consider Fig. 7.10, where the first event of p_2 is denoted e_2^1, first event of p_3 is denoted e_3^1, and the second event of p_3 is denoted e_3^2. Event e_2^1 is timestamped $\langle[0, 1, 0, 0], 2\rangle$, e_3^1 is timestamped $\langle[0, 0, 1, 0], 2\rangle$, and e_3^2 is timestamped $\langle[0, 3, 2, 2], 2\rangle$. As $e_2^1.vc[2] = 1 \leq e_3^2.vc[2] = 3$, we conclude $e_2^1 \xrightarrow{ev} e_3^2$. As $e_2^1.vc[2] = 1 > e_3^1.vc[2] = 0$ and $e_3^1.vc[3] = 1 > e_2^1.vc[3] = 0$, we conclude $e_2^1 \,||\, e_3^1$.

7.2.3 On the Development of Vector Time

When considering vector time, messages put restrictions on the development of time. More precisely, let m be a message sent by a process p_i at local time $vc_i[i] = a$ and received by a process p_j at local time $vc_j[j] = b$. This message induces the restriction that no event e can be dated $e.vc$ such that $e.vc[i] < a$ and $e.vc[j] \geq b$. This restriction simply states that there is no event such that there is a message whose reception belongs to its causal past while its sending does not.

To illustrate this restriction created by messages on the vector time domain, let us consider the execution depicted in Fig. 7.11. There are two processes, and each of them produces six events. The vector time date of each event is indicated above or below the corresponding event. Let us notice that the channel from p_1 to p_2 is not FIFO (p_1 sends m_1 before m_2, but p_2 receives m_2 before m_1).

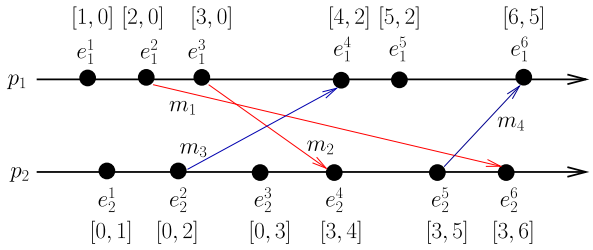

Fig. 7.11 On the development of time (1)

Let us consider the message m_2. As its sending time is $[3, 0]$ and its receive time is $[3, 4]$, it follows that it is impossible for an external observer to see a global time GD such that $GD[1] < 3$ and $GD[2] \geq 4$. The restrictions on the set of possible vector dates due to the four messages exchanged in the computation are depicted in Fig. 7.12. Each message prevents some vector clock values from being observed. As an example, the date $[2, 5]$ cannot exist, while the vector date $[3, 3]$ can be observed by an external observer.

The borders of the area including the vector dates that could be observed by external observers are indicated with dotted lines in the figure. They correspond to the history (sequence of events) of each process.

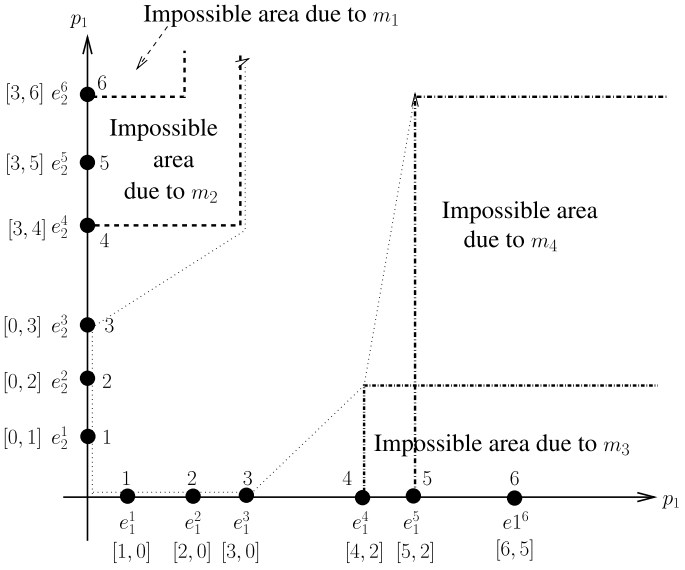

Fig. 7.12 On the development of time (2)

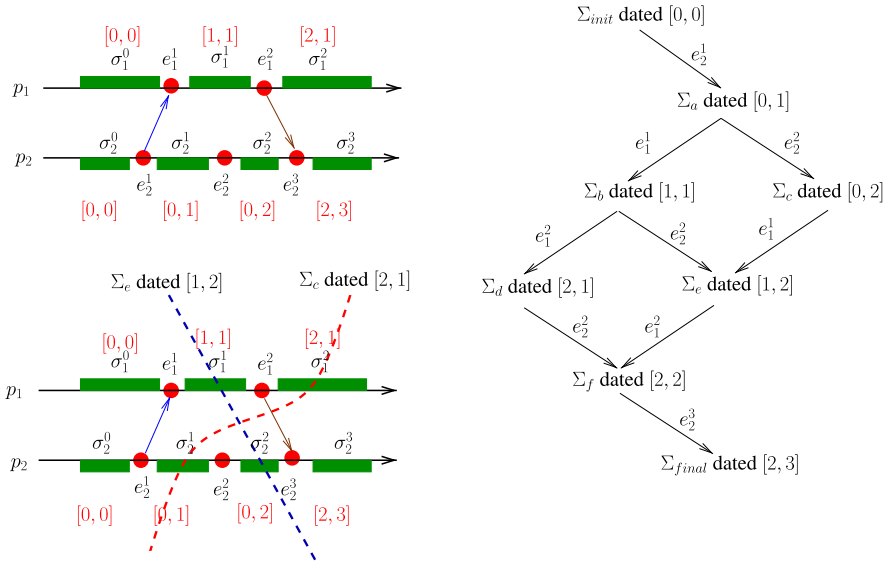

Fig. 7.13 Associating vector dates with global states

7.2.4 Relating Vector Time and Global States

Let us consider the distributed execution described on the left top of Fig. 7.13. Vector dates of events and local states are indicated in this figure. Both initial local states σ_1^0 and σ_2^0 are dated $[0, 0]$. Then, each σ_i^x inherits the vector date of the event e_i^x that generated it. As an example the date of the local state σ_2^2 is $[0, 2]$.

The corresponding lattice of global states is described at the right of Fig. 7.13. In this lattice, a vector date is associated with each global state as follows: the ith entry of the vector is the number of events produced by p_i. This means that, when considering the figure, the vector date of the global state $[\sigma_i^x, \sigma_j^y]$ is $[x, y]$. (This dating system for global states, which is evidently based on vector clocks, was implicitly introduced and used in Sect. 6.3.2, where the notion of a lattice of global states was introduced.) Trivially, the vector time associated with global dates increases along each path of the lattice.

Let us recall that the greatest lower bound (GLB) of a set of vertices of a lattice is their greatest common predecessor, while the least upper bound (LUB) is their least common successor. Due to the fact that the graph is a lattice, each of the GLB and the LUB of a set of vertices (global states) is unique.

An important consequence of the fact that the set of consistent global states is a lattice and the associated dating of global states is the following. Let us consider two global states Σ' and Σ'' whose dates are $[d'_1, \ldots, d'_n]$ and $[d''_1, \ldots, d''_n]$, respectively. Let $\Sigma^- = \mathrm{GLB}(\Sigma', \Sigma'')$ and $\Sigma^+ = \mathrm{LUB}(\Sigma', \Sigma'')$. We have

- $date(\Sigma^-) = date(\mathrm{GLB}(\Sigma', \Sigma'')) = [\min(d'_1, d''_1), \ldots, \min(d'_n, d''_n)]$, and

- $date(\Sigma^+) = date(\text{LUB}(\Sigma', \Sigma'')) = [\max(d'_1, d''_1), \ldots, \max(d'_n, d''_n)]$.

As an example, let us consider the global states denoted Σ_d and Σ_e in the lattice depicted at the left of Fig. 7.13. We have $\Sigma_b = \text{GLB}(\Sigma_d, \Sigma_e)$ and $\Sigma_f = \text{LUB}(\Sigma_d, \Sigma_e)$. It follows that we have $date(\Sigma_b) = [\min(2, 1), \min(1, 2)] = [1, 1]$. Similarly, we have $date(\Sigma_f) = [\max(2, 1), \max(1, 2)] = [2, 2]$. The consistent cuts associated with Σ_d and Σ_e are depicted in the bottom of the left side of Fig. 7.13.

These properties of the vector dates associated with global states are particularly important when one has to detect properties on global states. It allows processes to obtain information on the lattice without having to build it explicitly. At the operational level, processes have to compute vector dates identifying consistent global states which are relevant with respect to the property of interest (see Sect. 7.2.5).

7.2.5 *Vector Clocks in Action: On-the-Fly Determination of a Global State Property*

To illustrate the use of vector clocks, this section presents an algorithm that determines the first global state of a distributed computation that satisfies a conjunction of stable local predicates.

Conjunction of Stable Local Predicates A predicate is *local* to a process p_i if it is only on local variables of p_i. Let LP_i be a predicate local to process p_i. The fact that the local state σ_i of p_i satisfies the predicate LP_i is denoted $\sigma_i \models LP_i$.

Let $\{LP_i\}_{1 \leq i \leq n}$ be a set of n local predicates, one per process. (If there is no local predicate for a process p_x, we can consider a fictitious local predicate $LP_x = true$ satisfied by all local states of p_x.) The predicate $\bigwedge_i LP_i$ is a global predicate, called *conjunction of local predicates*. Let $\Sigma = [\sigma_1, \ldots, \sigma_n]$ be a consistent global state. We say that Σ satisfies the global predicate $\bigwedge_i LP_i$, if $\bigwedge_i (\sigma_i \models LP_i)$. This is denoted $\Sigma \models \bigwedge_i LP_i$.

A predicate is *stable* if, once true, it remains true forever. Hence, if a local state σ_i satisfies a local stable predicate LP_i, all the local states that follow σ_i in p_i's local history satisfy LP_i. Let us observe that, if each local predicate LP_i is stable, so is the global predicate $\bigwedge_i LP_i$.

This section presents a distributed algorithm that computes, on the fly and without using additional control messages, the first consistent global state that satisfies a conjunction of stable local predicates. The algorithm, which only adds control data to application messages, assumes that the application sends "enough" messages (the meaning of "enough" will appear clearly in the description of the algorithm).

On the Notion of a "First" Global State The consistent global state Σ defined by a process p_i from the vector date $first_i[1..n]$ is the *first* global state satisfying $\bigwedge_i LP_i$, in the following sense: There is no global state Σ' such that $(\Sigma' \neq \Sigma) \wedge (\Sigma' \models \bigwedge_i LP_i) \wedge (\Sigma' \xrightarrow{\Sigma} \Sigma)$.

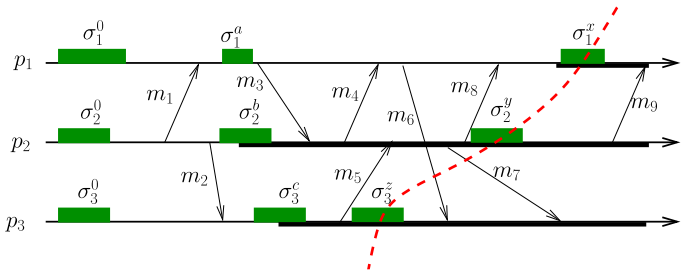

Fig. 7.14 First global state satisfying a global predicate (1)

Where Is the Difficulty Let us consider the execution described in Fig. 7.14. The predicates LP_1, LP_2 and LP_3 are satisfied from the local states σ_1^x, σ_2^b, and σ_3^c, respectively. (The fact that they are stable is indicated by a bold line on the corresponding process axis.) Not all local states are represented; the important point is that σ_2^y is the state of p_2 after it has sent the message m_8 to p_1, and σ_3^z is the state of p_3 after it has sent the message m_5 to p_2. The first consistent global state satisfying $LP_1 \wedge LP_2 \wedge LP_3$ is $\Sigma = [\sigma_1^x, \sigma_2^y, \sigma_3^z]$.

Using causality created by message exchanges and appropriate information piggybacked by messages, the processes can "learn" information related to the predicate detection. More explicitly, we have the following when looking at the figure.

- When p_1 receives m_1, it learns nothing about the predicate detection (and similarly for p_3 when it receives m_2).
- When p_1 receives m_4 (sent by p_2), it can learn that (a) the global state $[\sigma_1^0, \sigma_2^b, \sigma_3^0]$ is consistent and (b) it "partially" satisfies the global predicate, namely, $\sigma_2^b \models LP_2$.
- When p_2 receives the message m_3 (from p_1), it can learn that the global state $[\sigma_1^a, \sigma_2^b, \sigma_3^0]$ is consistent and such $\sigma_2^a \models LP_2$. When it later receives the message m_5 (from p_3), it can learn that the global state $[\sigma_1^a, \sigma_2^b, \sigma_3^c]$ is consistent and such $(\sigma_2^b \models LP_2) \wedge (\sigma_3^c \models LP_3)$. Process p_3 can learn the same when it receives the message m_7 (sent by p_2).
- When p_1 receives the message m_8 (sent by p_2), it can learn that, while LP_1 is not yet locally satisfied, the global state $[\sigma_1^a, \sigma_2^b, \sigma_3^c]$ is the first consistent global state that satisfies LP_2 and LP_3.
- Finally, when p_1 produces the internal event giving rise to σ_1^x, it can learn that the first consistent global state satisfying the three local predicates is $[\sigma_1^x, \sigma_2^y, \sigma_3^z]$. The corresponding consistent cut is indicated by a dotted line on the figure.

Let us recall that the vector date of the local state σ_2^y is the date of the preceding event, which is the sending of m_8, and this date is piggybacked by m_5. Similarly, the date of σ_3^z is the date of the sending of m_5.

Another example is given in Fig. 7.15. In this case, due to the flow of control created by the exchange of messages, it is only when p_3 receives m_3 that it can learn

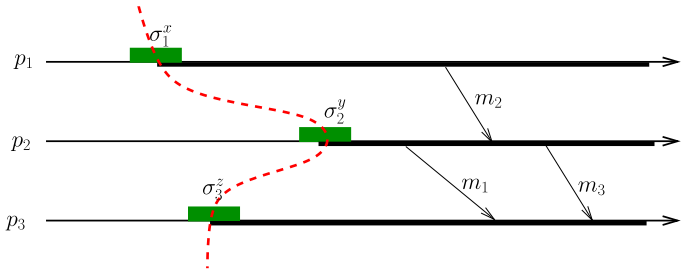

Fig. 7.15 First global state satisfying a global predicate (2)

that $[\sigma_1^x, \sigma_2^y, \sigma_3^z]$ is the first consistent global state satisfying $\bigwedge_i LP_i$. As previously, the corresponding consistent cut is indicated by a dotted line on the figure.

As no control messages are allowed, it is easy to see that, in some scenarios, enough application messages have to be sent after $\bigwedge_i LP_i$ is satisfied, in order to compute the vector date of the first consistent global state satisfying the global predicate $\bigwedge_i LP_i$.

Local Control Variables In order to implement the determination of the first global state satisfying a conjunction of stable local predicates, each process p_i manages the following local variables:

- $vc_i[1..n]$ is the local vector clock.
- sat_i is a set, initially empty, of process identities. It has the following meaning: $(j \in sat_i) \Leftrightarrow (p_i$ knows that p_j has entered a local state satisfying $LP_j)$.
- $done_i$ is Boolean, initialized to *false*, and then set to the value *true* by p_i when LP_i becomes satisfied for the first time. As LP_i is a stable predicate, $done_i$ then keeps that value forever.
- $first_i$ is the vector date of the first consistent global state, known by p_i, for which all the processes in sat_i satisfy their local predicate. Initially, $first_i = [0, \ldots, 0]$. Hence, $[\sigma_1^{first[1]}, \ldots, \sigma_n^{first[n]}]$ is this global state, and we have $\forall\ j \in sat_i : \sigma_j^{first[j]} \models LP_j$.

  As an example, considering Fig. 7.14, after p_1 has received the message m_4, we have $sat_1 = \{2\}$ and $first_1 = [0, b, 0]$. After it has received the message m_8, we have $sat_1 = \{2, 3\}$ and $first_1 = [0, b, c]$.

The Algorithm The algorithm is described in Fig. 7.16. It ensures that if consistent global states satisfy $\bigwedge LP_i$, at least one process will compute the vector date of the first of them. As already indicated, this is under the assumption that the processes send enough application messages.

Before producing a new event, p_i always increases its local clock $vc_i[i]$ (lines 7, 10, and 13). If the event e is an internal event, and LP_i has not yet been satisfied (i.e., $done_i$ is false), p_i invokes the operation check_lp() (line 9). If its current local state σ satisfies LP_i (line 1), p_i adds its identity to sat_i and, as it is the first time that

```
internal operation check_lp(σ) is
(1)  if (σ ⊨ LP_i) then
(2)      sat_i ← sat_i ∪ {i}; first_i[1..n] ← vc_i[1..n]; done_i ← true;
(3)      if (sat_i = {1, ..., n}) then
(4)          first_i[1..n] is the vector date of the first global state satisfying ⋀_j LP_j
(5)      end if
(6)  end if.

when producing an internal event e do
(7)  vc_i[i] ← vc_i[i] + 1;
(8)  Produce event e and move to the next state σ;
(9)  if (¬done_i) then check_lp(σ) end if.

when sending MSG(m) to p_j do
(10) vc_i[i] ← vc_i[i] + 1; move to the next local state σ;
(11) if (¬done_i) then check_lp(σ) end if;
(12) send MSG(m, vc_i, sat_i, first_i) to p_j.

when MSG(m, vc, sat, first) is received from p_j do
(13) vc_i[i] ← vc_i[i] + 1; vc_i ← max(vc_i[1..n], vc[1..n]);
(14) move to the next local state σ;
(15) if (¬done_i) then check_lp(σ) end if;
(16) if (sat ⊈ sat_i) then
(17)     sat_i ← sat_i ∪ sat; first_i ← max(first_i[1..n], first[1..n]);
(18)     if (sat_i = {1, ..., n}) then
(19)         first_i[1..n] is the vector date of the first global state satisfying ⋀_j LP_j
(20)     end if
(21) end if.
```

Fig. 7.16 Detection the first global state satisfying $\bigwedge_i LP_i$ (code for process p_i)

LP_i is satisfied, it defines accordingly $first_i$ as being vc_i (which is the vector date of the global state associated with the causal past of the event e currently produced by p_i, line 2). Process p_i then checks if the consistent global state defined by the vector date $first_i (= vc_i)$ satisfies the whole global predicate $\bigwedge_j LP_j$ (lines 3–4). If it is the case, p_i has determined the first global state satisfying the global predicate.

If the event e is the sending of a message by p_i to p_j, before sending the message, process p_i first moves to its next local state σ (line 10), and does the same as if e was an internal event. The important point is that, in addition to the application message, p_i sends to p_j (line 12) its full current state (from the global state determination point of view), which consists of vc_i (current vector time at p_i), sat_i (processes whose local predicates are satisfied), and $first_i$ (date of the consistent global state satisfying the local predicates of the processes in sat_i).

If the event e is the reception by p_i of a message sent by p_j, p_i updates first its vector clock (line 13), and moves to its next local state (line 14). As in the previous cases, if LP_i has not yet been satisfied, p_i then invokes check_lp() (line 15). Finally, if p_i learns something new with respect to local predicates (test of line 16), it "adds" what it knew before (sat_i and $first_i[1..n]$) with what it learns (sat and $first[1..n]$). The new value of $first_i[1..n]$ is the vector date of the first consistent global state

in which all the local states of the processes in $sat_i \cup sat$ satisfy their local predicates. Finally, p_i checks if the global state defined by $first_i[1..n]$ satisfies all local predicates (lines 18–20).

When looking at the executions described in Figs. 7.14 and 7.15, we have the following. In Fig. 7.14, p_1 ends the detection at line 4 after it has produced the event that gave rise to σ_1^x. In Fig. 7.15, p_3 ends the detection at line 19 after it has received the protocol message MSG$(m_3, [-,-,-], \{1,2\}, [x,y,0])$. Just before it receives this message we have $sat_3 = \{2,3\}$ and $first_3 = [0,y,z]$.

7.2.6 Vector Clocks in Action: On-the-Fly Determination of the Immediate Predecessors

The Notion of a Relevant Event At some abstraction level, only a subset of events are relevant. As an example, in some applications only the modification of some local variables, or the processing of specific messages, are relevant. Given a distributed computation $\widehat{H} = (H, \stackrel{ev}{\longrightarrow})$, let $R \subset H$ be the subset of its events that are defined as relevant.

Let us accordingly define the causal precedence relation on these events, denoted $\stackrel{re}{\longrightarrow}$, as follows

$$\forall e_1, e_2 \in R: \quad (e_1 \stackrel{re}{\longrightarrow} e_2) \Leftrightarrow (e_2 \stackrel{ev}{\longrightarrow} e_1).$$

This relation is nothing else than the projection of $\stackrel{ev}{\longrightarrow}$ on the elements of R. The pair $\widehat{R} = (R, \stackrel{re}{\longrightarrow})$ constitutes an abstraction of the distributed computation $\widehat{H} = (H, \stackrel{ev}{\longrightarrow})$.

Without loss of generality, we consider that the set of relevant events consists only of internal events. Communication events are low-level events which, without being themselves relevant, participate in the establishment of causal precedence on relevant events. If a communication event has to be explicitly observed as relevant, an associated relevant internal event can be generated just before or after it.

An example of a distributed execution with both relevant events and non-relevant events is described in Fig. 7.17. The relevant events are represented by black dots, while the non-relevant events are represented by white dots.

The management of vector clocks restricted to relevant events (Fig. 7.18) is a simplified version of the one described in Fig. 7.9. As we can see, despite the fact that they are not relevant, communication events participate in the tracking of causal precedence. Given a relevant e timestamped $\langle e.vc[1..n], i\rangle$, the integer $e.vc[j]$ is the number of relevant events produced by p_j and known by p_i.

The Notion of an Immediate Predecessor and the Immediate Predecessor Tracking Problem Given two events $e_1, e_2 \in R$, the relevant event e_1 is an *immediate predecessor* of the relevant event e_2 if

$$(e_1 \stackrel{re}{\longrightarrow} e_2) \wedge \big(\nexists e \in R: (e_1 \stackrel{re}{\longrightarrow} e) \wedge (e \stackrel{re}{\longrightarrow} e_2)\big).$$

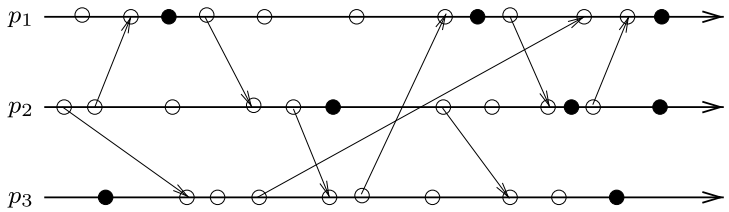

Fig. 7.17 Relevant events in a distributed computation

```
when producing a relevant internal event e do
(1)  vc_i[i] ← vc_i[i] + 1;
(2)  Produce the relevant event e.  % The date of e is vc_i[1..n].

when sending MSG(m) to p_j do
(3)  send MSG(m, vc_i[1..n]) to p_j.

when MSG(m, vc) is received from p_j do
(4)  vc_i[1..n] ← max(vc_i[1..n], vc[1..n]).
```

Fig. 7.18 Vector clock system for relevant events (code for process p_i)

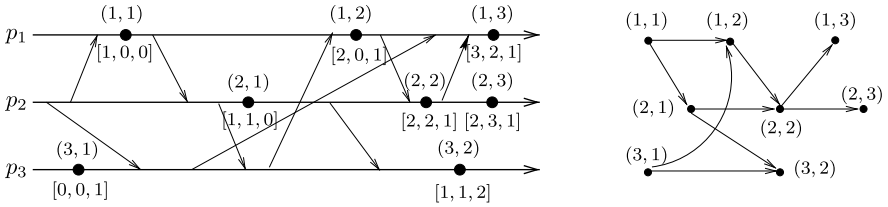

Fig. 7.19 From relevant events to Hasse diagram

The *immediate predecessor tracking* (IPT) problem consists in associating with each relevant event the set of relevant events that are its immediate predecessors. Moreover, this has been done on the fly and without adding control messages. The determination of the immediate predecessors consists in computing the transitive reduction (or Hasse diagram) of the partial order $\widehat{R} = (R, \xrightarrow{re})$. This reduction captures the essential causality of $\widehat{R}$.

The left of Fig. 7.19 represents the distributed computation of Fig. 7.17, in which only the relevant events are explicitly indicated, together with their vector dates and an identification pair (made up of a process identity plus a sequence number). The right of the figure shows the corresponding Hasse diagram.

An Algorithm Solving the IPT Problem: Local Variables The kth relevant event on process p_k is unambiguously identified by the pair (k, vck), where vck is the value of $vc_k[k]$ when p_k has produced this event. The aim of the algorithm is conse-

```
when producing a relevant internal event e do
(1)  vc_i[i] ← vc_i[i] + 1;
(2)  Produce the relevant event e; let e.imp = {(k, vc_i[k]) | imp_i[k] = 1};
(3)  imp_i[i] ← 1;
(4)  for each j ∈ {1, ..., n} \ {i} do imp_i[j] ← 0 end for.

when sending MSG(m) to p_j do
(5)  send MSG(m, vc_i[1..n], imp_i[1..n]) to p_j.

when MSG(m, vc, imp) is received do
(6)  for each k ∈ {1, ..., n} do
(7)     case vc_i[k] < vc[k] then vc_i[k] ← vc[k]; imp_i[k] ← imp[k]
(8)          vc_i[k] = vc[k] then imp_i[k] ← min(imp_i[k], imp[k])
(9)          vc_i[k] > vc[k] then skip
(10)    end case
(11) end for.
```

Fig. 7.20 Determination of the immediate predecessors (code for process p_i)

quently to associate with each relevant event e a set $e.imp$ such that $(k, vck) \in e.imp$ if and only if the corresponding event is an immediate predecessor of e.

To that end, each process p_i manages the following local variables:

- $vc_i[1..n]$ is the local vector clock.
- $imp_i[1..n]$ is a vector initialized to $[0, \ldots, 0]$. Each variable $imp_i[j]$ contains 0 or 1. Its meaning is the following: $imp_i[j] = 1$ means that the last relevant event produced by p_j, as known by p_i, is candidate to be an immediate predecessor of the next relevant event produced by p_i.

An Algorithm Solving the IPT Problem: Process Behavior The algorithm executed by each process p_i is a combined management of both the vectors vc_i and imp_i. It is described in Fig. 7.20.

When a process p_i produces a relevant event e, it increases its own vector clock (line 1). Then, it considers $imp_i[1..n]$ to compute the immediate predecessors of e (line 2). According to the definition of each entry of imp_i, those are the events identified by the pairs $(k, cv_i[k])$ such that $imp_i[k] = 1$ (which indicates that the last relevant event produced by p_k and known by p_i, is still a candidate to be an immediate predecessor of e).

Then, as it produced a new relevant event e, p_i must reset its local array $imp_i[1..n]$ (lines 3–4). It resets (a) $imp_i[i]$ to 1 (because e is candidate to be an immediate predecessor of relevant events that will appear in its causal future) and (b) each $imp_i[j]$ $(j \neq i)$ to 0 (because no event that will be produced in the future of e can have them as immediate predecessors).

When p_i sends a message, it attaches to this message all its local control information, namely, cv_i and imp_i (line 5). When it receives a message, p_i updates it local vector clock as in the basic algorithm so that the new value of vc_i is the component-wise maximum of vc and the previous value of vc_i.

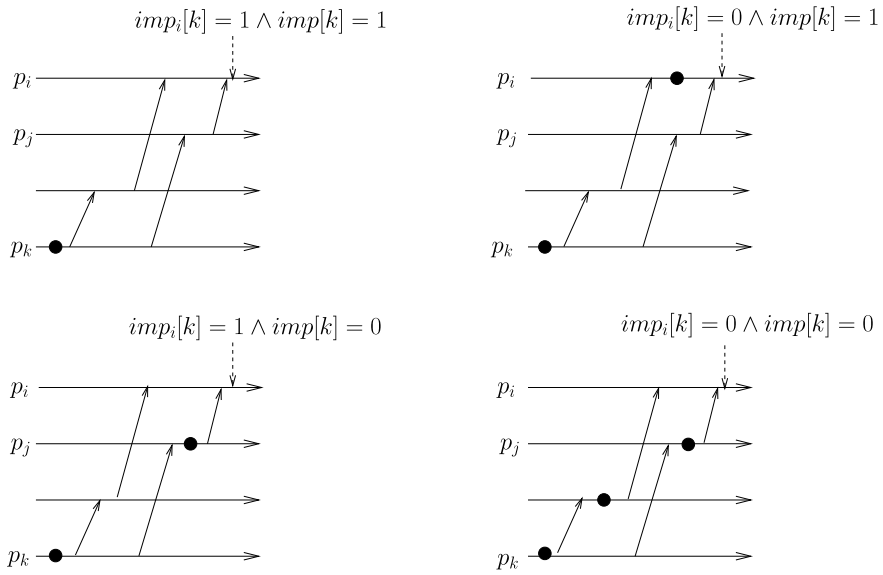

Fig. 7.21 Four possible cases when updating $imp_i[k]$, while $vc_i[k] = vc[k]$

The update of the array imp_i depends on the value of each entry of vc_i.

- If $vc_i[k] < vc[k]$, p_j (the sender of the message) has fresher information on p_k than p_i. Consequently, p_i adopts what is known by p_j, and sets $imp_i[k]$ to $imp[k]$ (line 7).
- If $vc_i[k] = vc[k]$, the last relevant event produced by p_k and known by p_i is the same as the one known by p_j. If this event is still candidate to be an immediate predecessor of the next event produced by p_i from both p_i's point of view (($imp_i[k]$) and p_j's point of view (($imp[k]$), then $imp_i[k]$ remains equal to 1; otherwise, $imp_i[k]$ is set to 0 (line 8). The four possible cases are depicted in Fig. 7.21.
- If $vc_i[k] > vc[k]$, p_i knows more on p_k than p_j. Hence, it does not modify the value of $imp_i[k]$ (line 9).

7.3 On the Size of Vector Clocks

This section first shows that vector time has an inherent price, namely, the size of vector clocks cannot be less than the number of processes. Then it introduces the notion of a *relevant event* and presents a general technique that allows us to reduce the number of vector entries that have to be transmitted in each message. Finally, it presents the notions of *approximation* of the causality relation and approximate vector clocks.