Virtual Time

"Virtual Time and Global States of Distributed Systems"

Friedmann Mattern, 1989

The Model: An *asynchronous distributed system* = a set of processes having no shared memory, communicating by message transfer.

Message delay > 0, but is not known in advance.

A global observer – sees the global state at certain points in time. It can be said to "take a snapshot" of the global state.

A local observer – (one of the processes in the system) sees the local state. Because of the asynchrony, a local observer can only gather local views to an ***approximate*** global view.

The lack of a global observer is a hard hazard for many management and control problems: mutual exclusion, deadlock detection, distributed contracts, leader election, load sharing, checkpointing etc.

Solution Approaches

1. Simulating a synchronous system by an asynchronous one. This requires high overhead on global synchronization of each and every step.

2. Simulation of a global state. A snapshot, taken asynchronously, which is not necessarily correct for any specific point in time, but is in a way “consistent” with the local states of all processes.

3. A logical clock which is not global, but can be used to derive useful global information. The system works asynchronously, but the processes make sure to maintain their part of the clock.

Events

An event is a change in the process state.

- An event happens instantly, it does not “take time”.
- A process is a sequence of events
- There are 3 types of events:
  1. *send event* – causes a message to be sent
  2. *receive event* – causes a message to be received
  3. *local event* – only causes an internal change of state

Events correspond to each other as follows:

- All events in the same process happen sequentially, one after the other.
- Each send event has a corresponding receive event

This allows us to define the *happened before* relation among events.

The *Happened Before* Relation

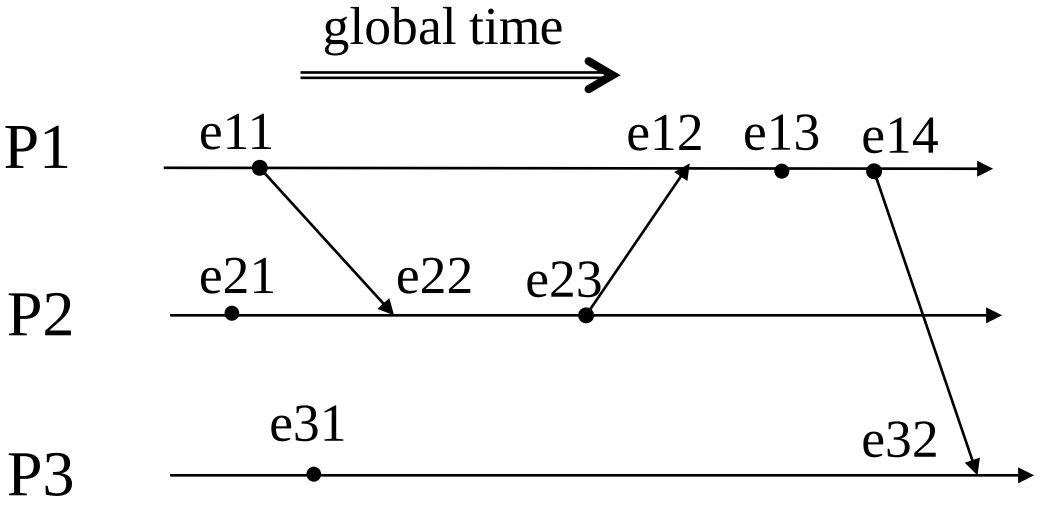
We say that event e *happened before* event e' (and denote it by e → e' or e < e') if one of the following properties holds:

Processor Order: e precedes e' in the same process

Send-Receive: e is a send and e' is the corresponding receive

Transitivity: exists e'' s.t. e < e'' and e''< e'

Example:

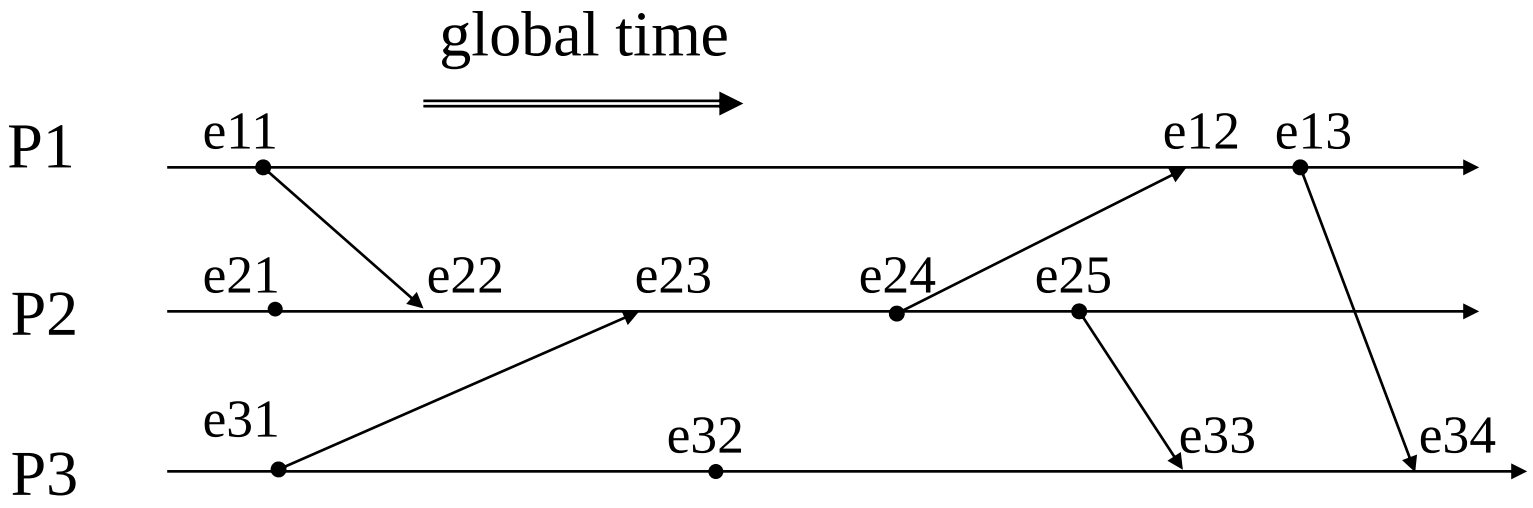


Program order: e13 < e14

Send-Receive: e23 < e12

Transitivity: e21 < e32

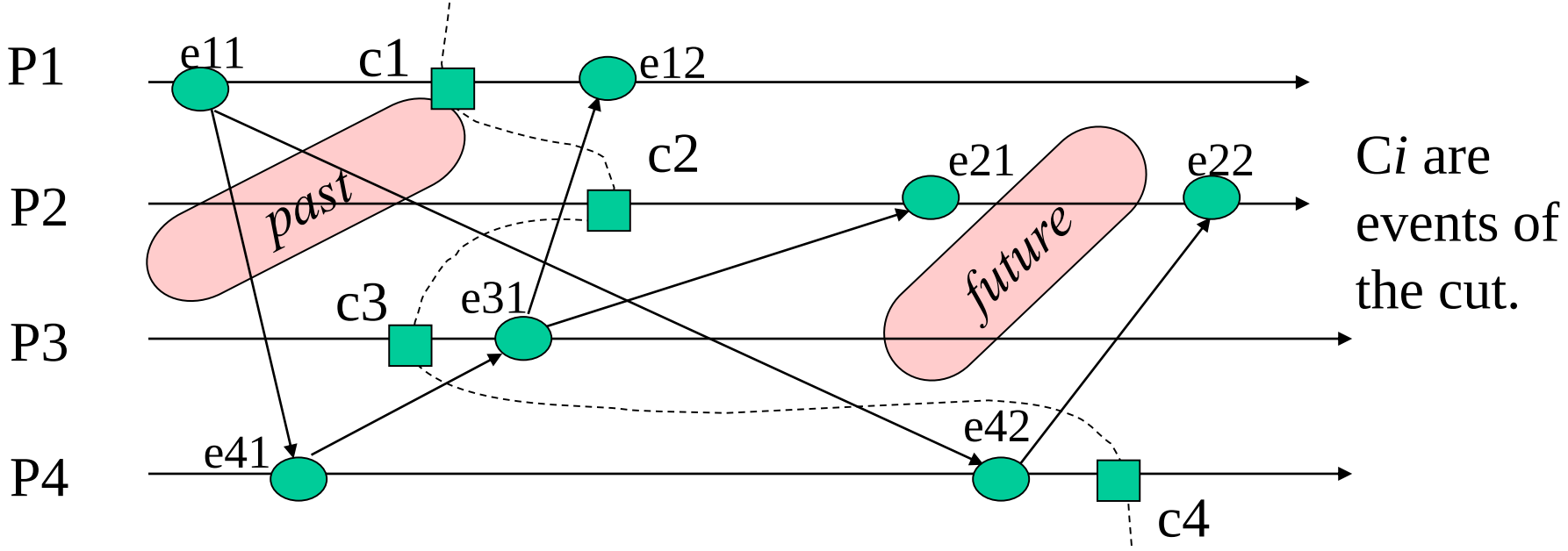
Independent/Concurrent Events



Two such diagrams are called *equivalent* when the happened before relation is the same in both. (When global time differs for certain events, think of processor execution as if it was a *rubber band*).

Two events e, e' are said to be *independent* or *concurrent* (denoted by e || e') if not e < e' and not e' < e.

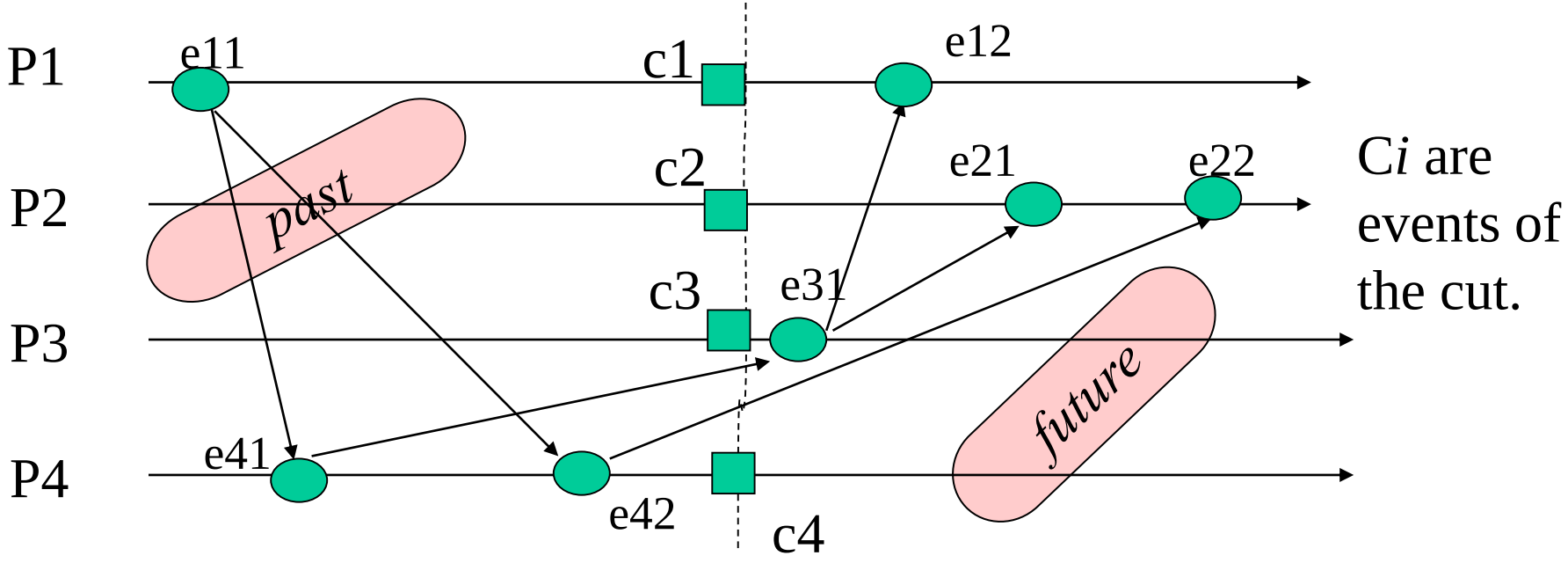
Cuts



A *cut* C of a set of events E is the union set of the cut events set { ci } and all the events e ∈ E which precede any of the ci in program order.

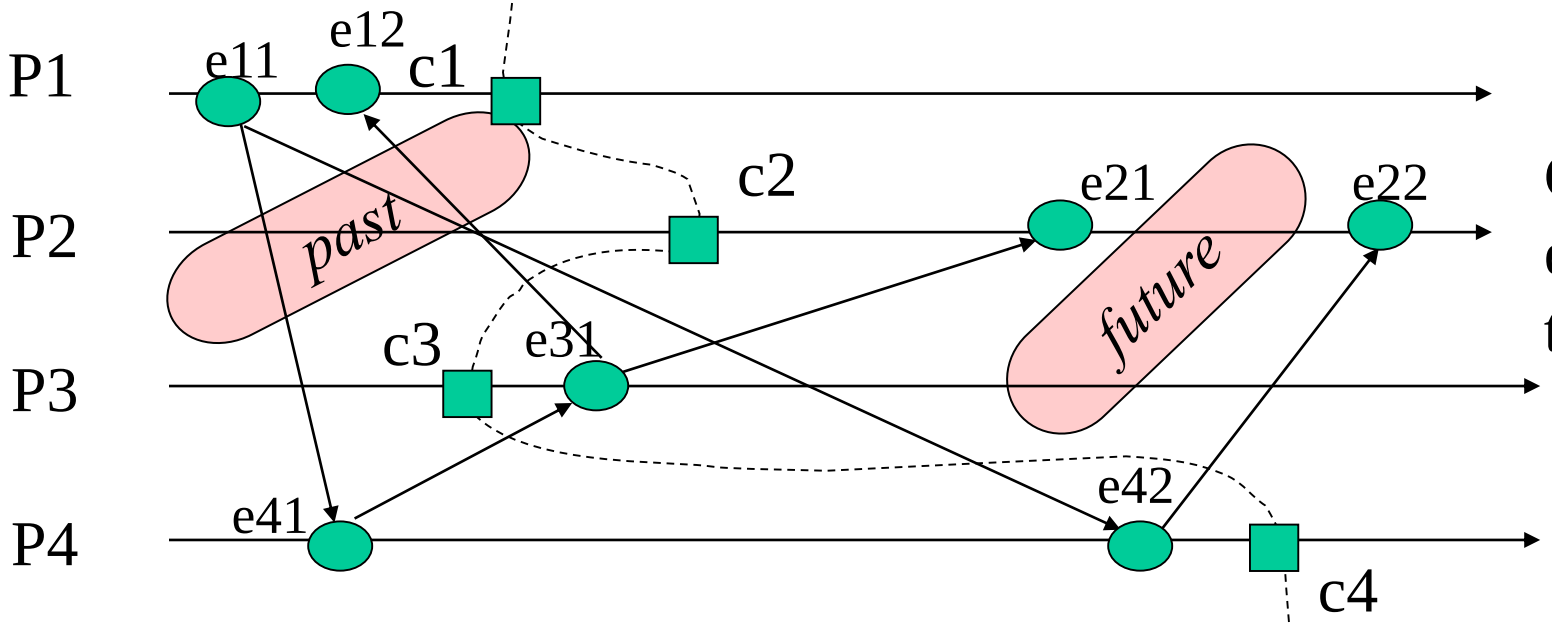
A cut is said to be *consistent* if also: e ∈ C and e'<e always imply e'∈ C.
If a receipt of a message is inside a consistent cut – so is its sending.
The opposite does not necessarily hold.

Consistent Cuts

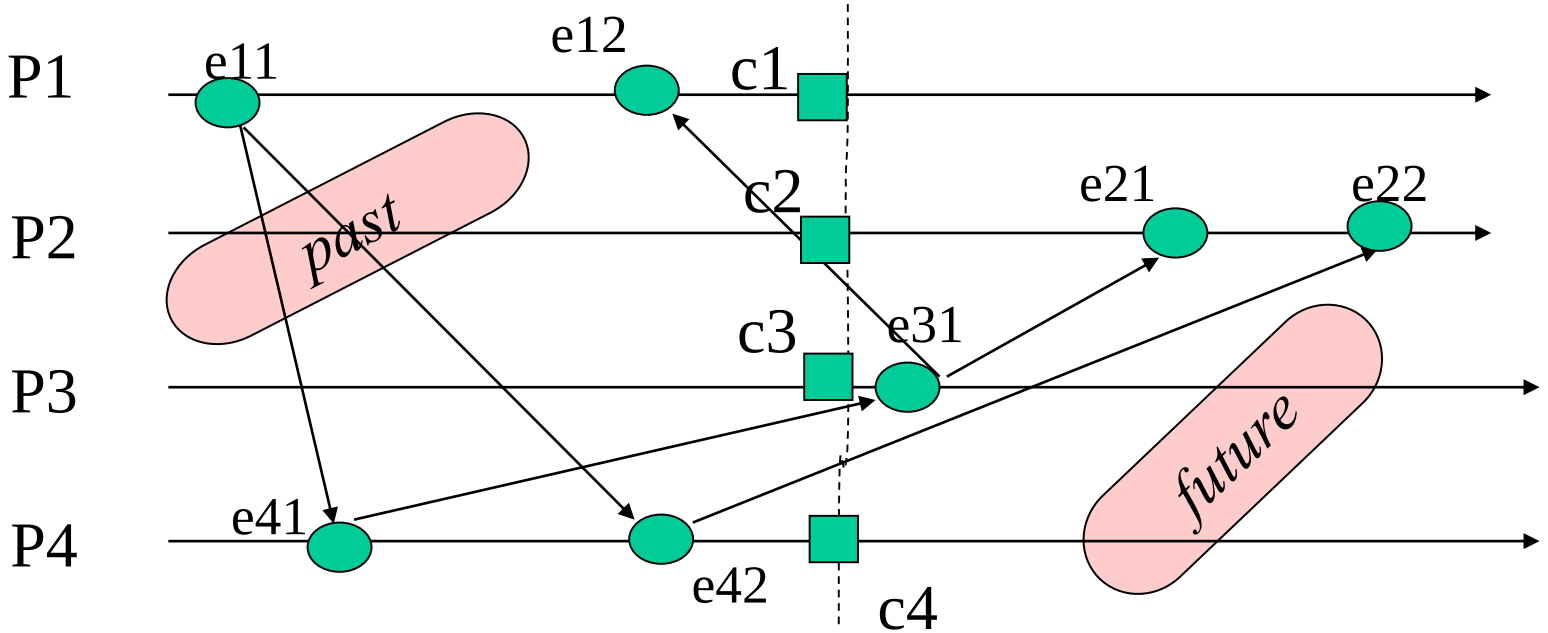


Intuitively, a cut is consistent if when stretching the execution lines so that the cut events form a vertical line, there is no send-receive arrow which crosses the cut right to left.

inconsistent Cut



C*i* are events of the cut.



Global View and Consistent Cuts

• If a cut is consistent then there is an execution of the diagram in which all cut events happen at the same moment.

• Clearly, the view of a global observer at any point in time is necessarily consistent (a message that arrived was necessarily previously sent). Thus:

• A global view determines a consistent cut at any point in time.

Consistent Cuts and Global Snapshots

- Any consistent cut may actually happen at a certain point during a an execution.

- The cut is said to contain the local states of the cut processes and the set of messages that were already sent but not received. This is a “snapshot” of what is happening in the system.

- The Global Snapshot problem: Find an efficient protocol to compute consistent cuts. i.e. collect the local process states and the messages which are “in flight” (or, “in the communication buffers”) at the time of the cut.

Virtual Time (Lamport, 1978)

A *logical clock* is a function C: E(vents) → T(ime)

E – set of events, C(e) timestamp of e

T – a partially ordered set s.t. e<e' ➔ C(e)<C(e')

(the opposite not necessarily true, e.g. concurrent events.)

Commonly, T=*N*, and there is a local clock Ci for every process Pi.

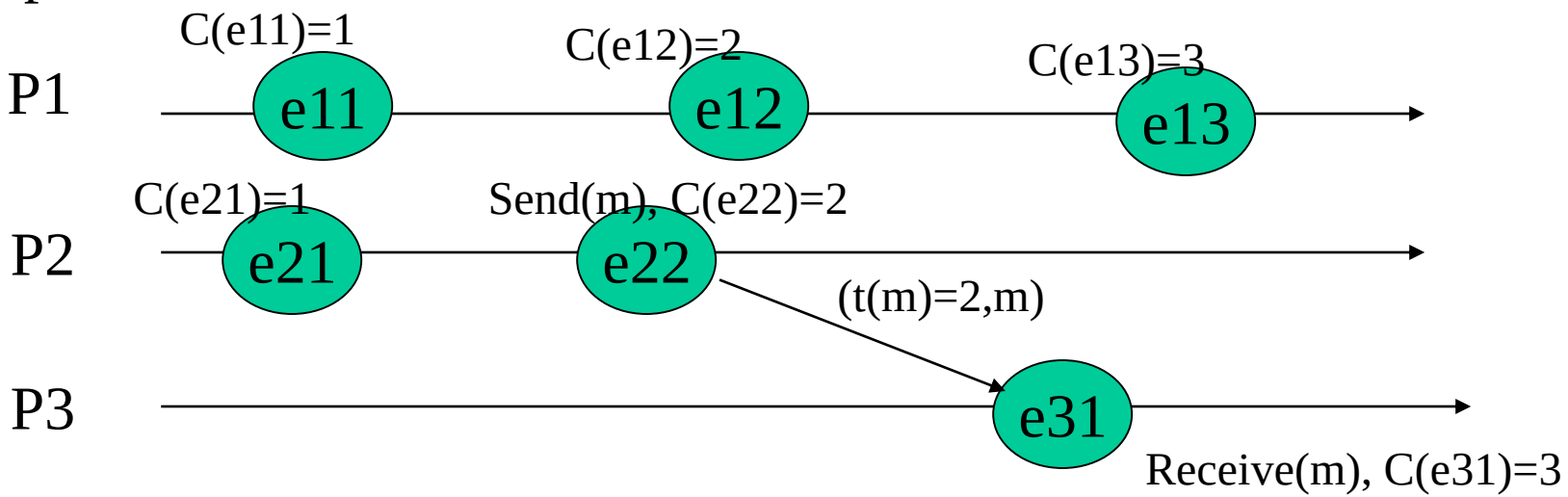
The clocks tick using the following protocol:

1. When a local event e is executed by Pi:
   - new Ci := Ci + d (d>0)
   - C(e) := Timestamp(e) := new Ci
2. A message m, sent on event s = send(m), is time-stamped t(m) = C(s); t(m) is sent together with the m as a pair (t(m),m).
3. When Pi receives (t(m),m):
   - Ci := max{ Ci, t(m) } + d (d >0)

Usually, d=1. However, d may change arbitrarily and dynamically (as long as it is positive), e.g., in an attempt to reflect actual time.

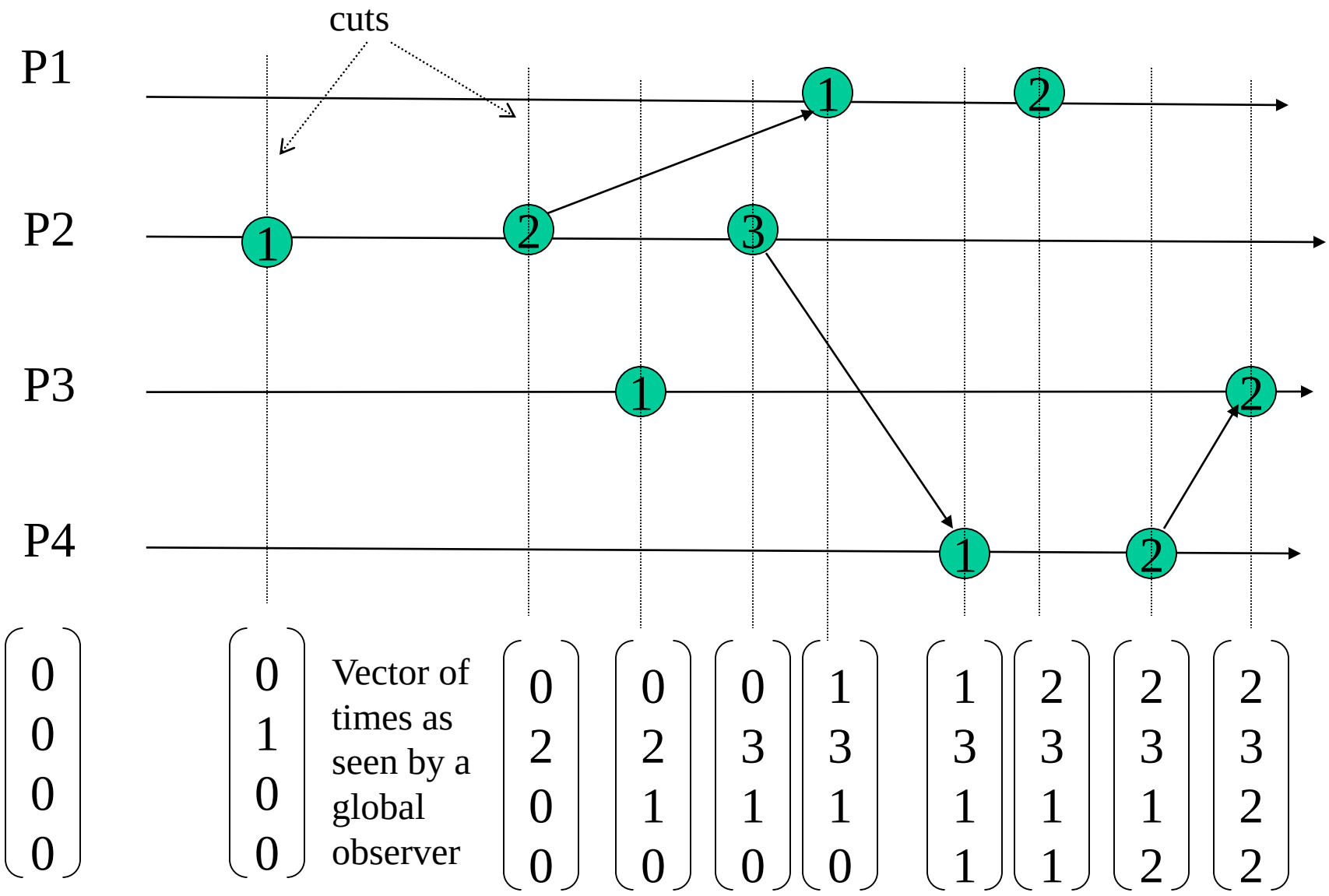
Loss of execution information

Example:



- A problem: Suppose we read execution log where only the timestamps of the events were recorded.
- Suppose C(e) < C(e') . We can infer that e' cannot have happened (Lamport-sense) before e: "not (e'<e)".
- But we cannot distinguish between e < e' and e || e'. Information whether the events are independent is lost.
- Notice: a global observer would know that.
- Question: can we change the clock mechanism to preserve more info?

A Vector Clock for a Global Observer



Approximating Global Vector Time

Idea: let's compute for every process an approximation of the global observer's vector clock (will have to show approx-n is good enough).

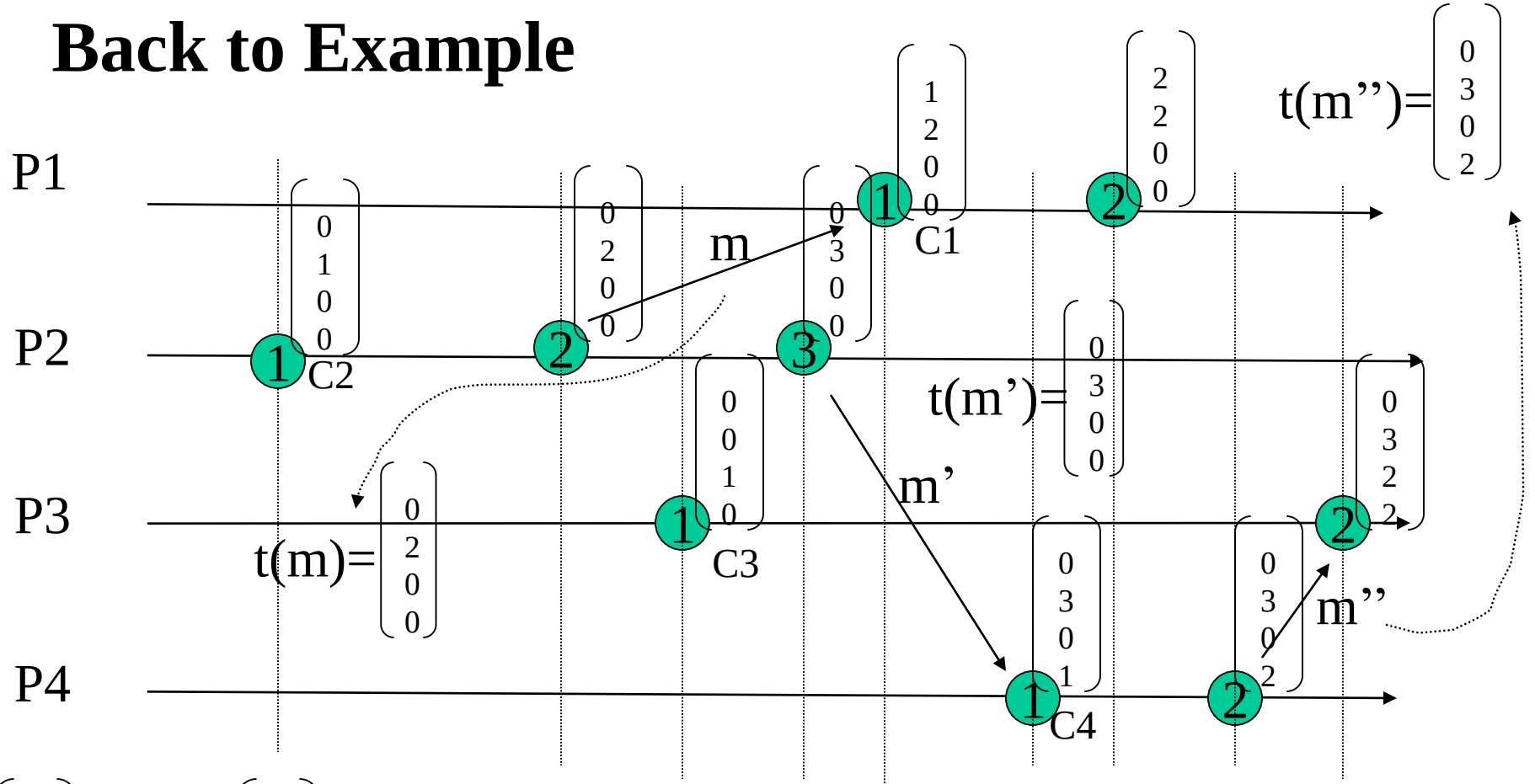
Instead of scalar clocks, process Pi will maintain a *vector clock* Ci[].

Timestamp C(e) of an event e at process Pi is Ci[] when e happens.

1. When executing a local event e in Pi:
   - Ci[i] := Ci[i] + 1 (Ci[] ticks)
   - C(e) :=Ci[]
2. A message m sent by event s = send(m), is time-stamped t(m) = C(s).
   The message is sent together with the timestamp as a pair (t(m),m).
3. Upon receipt of a message m by Pi:
   - for every j: Ci[j] := max{ Ci[j], t(m)[j] }
   - Ci[i] := Ci[i] + 1 (Ci[] ticks)

The time stamp of a received message contains information about other processes states. On receipt of a message, a process combines this information with what it already knows about the system state, and communicates it elsewhere, piggybacking on messages it sends out.

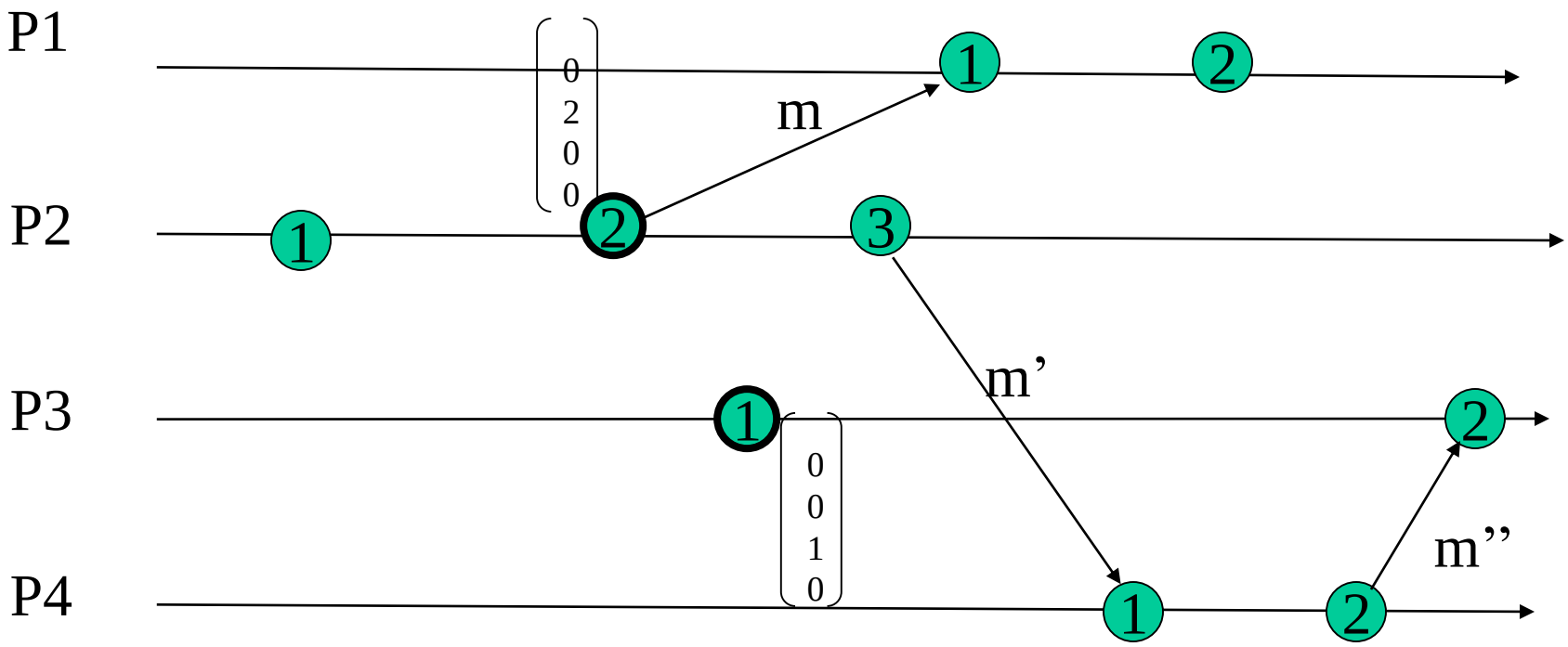
Back to Example



Note 1: P3 receives ticking information from P2 albeit they never communicate directly.

Note 2: Local timestamps approximating global observer times

Actual Order of Independent Events not Important



What is the actual global order?

$\begin{pmatrix}0\\0\\0\\0\end{pmatrix} \rightarrow \begin{pmatrix}0\\1\\0\\0\end{pmatrix} \rightarrow \begin{pmatrix}0\\2\\0\\0\end{pmatrix} \rightarrow \begin{pmatrix}1\\2\\0\\0\end{pmatrix} \rightarrow \begin{pmatrix}2\\3\\0\\0\end{pmatrix} \rightarrow \begin{pmatrix}2\\3\\1\\0\end{pmatrix}$

- Really don't Care!
- Can tell e2,1 and e3,1 are concurrent (vectors independen

$\begin{pmatrix}0\\0\\0\\0\end{pmatrix} \rightarrow \begin{pmatrix}0\\0\\1\\0\end{pmatrix} \rightarrow \begin{pmatrix}0\\1\\1\\0\end{pmatrix} \rightarrow \begin{pmatrix}0\\2\\1\\0\end{pmatrix} \rightarrow \begin{pmatrix}0\\3\\1\\0\end{pmatrix} \rightarrow \begin{pmatrix}2\\3\\1\\0\end{pmatrix}$

- Inconsistent
- global time = is impossible. $\begin{pmatrix}1\\1\\0\\0\end{pmatrix}$

An Invariant (*)

Only Pi can advance his clock: Ci[i] ticks only at Pi.

Thus, Pi always knows the up-to-date time of his local time.

Thus, at any point in time, for any i,j, it holds that:

$$C_i[i] \geq C_j[i]$$

Vector Times can be Compared

u did not happen after *v*:

$$u \leq v \Leftrightarrow \forall i : u[i] \leq v[i]$$

u happened before *v*:

$$u < v \Leftrightarrow u \leq v \,\&\, u \neq v$$

u and *v* concurrent:

$$u \parallel v \Leftrightarrow \neg(u < v) \,\&\, \neg(v < u)$$

➔ With vector times, timestamps can keep more information on order of events.

Need to show: time stamp vector relations and Lamport's happens-before relation of events are equivalent.

Vector Times Relations Represent Order of Events

Claim: for every e,e'

1. $C(e) <_{vectorial} C(e')$ iff $e <_{Lamport} e'$
2. $C(e) \|_{vectorial} C(e')$ iff $e \|_{Lamport} e'$
3. If in addition e is known to have happened in process Pi then for every e' (<>e): $C(e)[i] < C(e')[i]$ iff $e < e'$

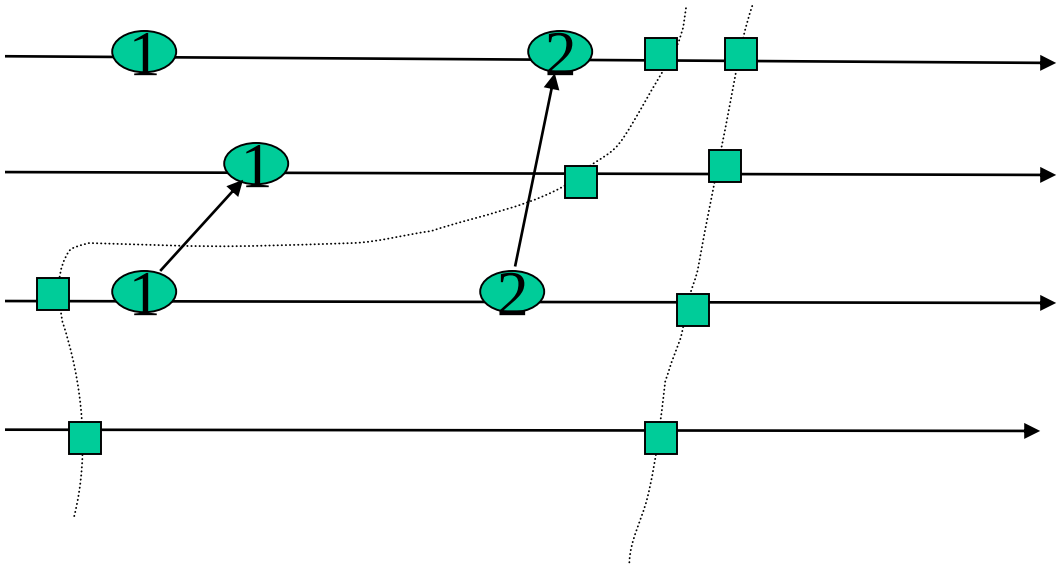
Proof for (1): (Proofs of 2,3 are similar.)

➔ If e' knows the local time of e (i.e., the vector component corresponding to the process where e happened is bigger in e' than in e), then there must exist a sequence of time stamps from e to e' which represents the happened before relation.

➔ If e<e' then there exist a sequence of time stamps from e to e', and these time stamps monotonically grow in each component. The growth is strict in at least one component: that corresponding to e's process.

Cuts Revisited

Let X be a cut, {xi} cut events, a system of n processes.
We say that the *global time* of X is tx = max(C(x1), … , C(xn))
(max is taken per entry).



Example:
Different cuts have same global Time.

$$\begin{pmatrix} 2 \\ 1 \\ 2 \\ 0 \end{pmatrix}$$

However …

Different Consistent Cuts Have Different Global Times

Claim: X is consistent iff tx = [C(x1)[1], … ,C(xn)[n]]

Proof:

➔ If X is consistent then its cut events xi can be viewed as if they occurred at the same moment. By the invariant (*) at that moment the claim holds.

➔ If X is not consistent, then there exists a message that was sent (say) from Pi after xi, and was received by Pj before xj. Let t be the timestamp on the message, then xi[i] < t[i] <= xj[i].

Thus tx > [C(x1)[1], … ,C(xn)[n]].