Vector clocks

Independently proposed by Fidge and by Mattern in 1988

- Vector clocks:
  - Assume system contains *n* processes
  - Each process *Pi* has a clock *Ci* , which is an integer vector of length *n*

$$Ci = (Ci\ [1], Ci\ [2], \ldots\ Ci\ [n]\ )$$

  - *Ci* (*a*) is the timestamp (clock value)
  - of event *a* at process *Pi*

- *Ci* [*i*](*a*), entry *i* of of *Ci* , is *Pi* 's logical time
- *Ci* [*k*](*a*), entry *k* of of *Ci* (where *k*¹*i*), is

Pi 's best guess of the logical time at *Pk* More specifically, the time of the occurrence of the last event in *Pk* which "happened before" the current event in *Pi* (based on messages received)

Implementation of vector clocks

Implementation Rules:

[IR1] Clock *Ci* must be incremented between any two successive events in process *Pi* :C [i]:= C [i] + d (d>0, usually d=1)

[IR2] If event a is the event of sending a message m in process Pi , then message m is assigned a vector timestamp tm =Ci (a)

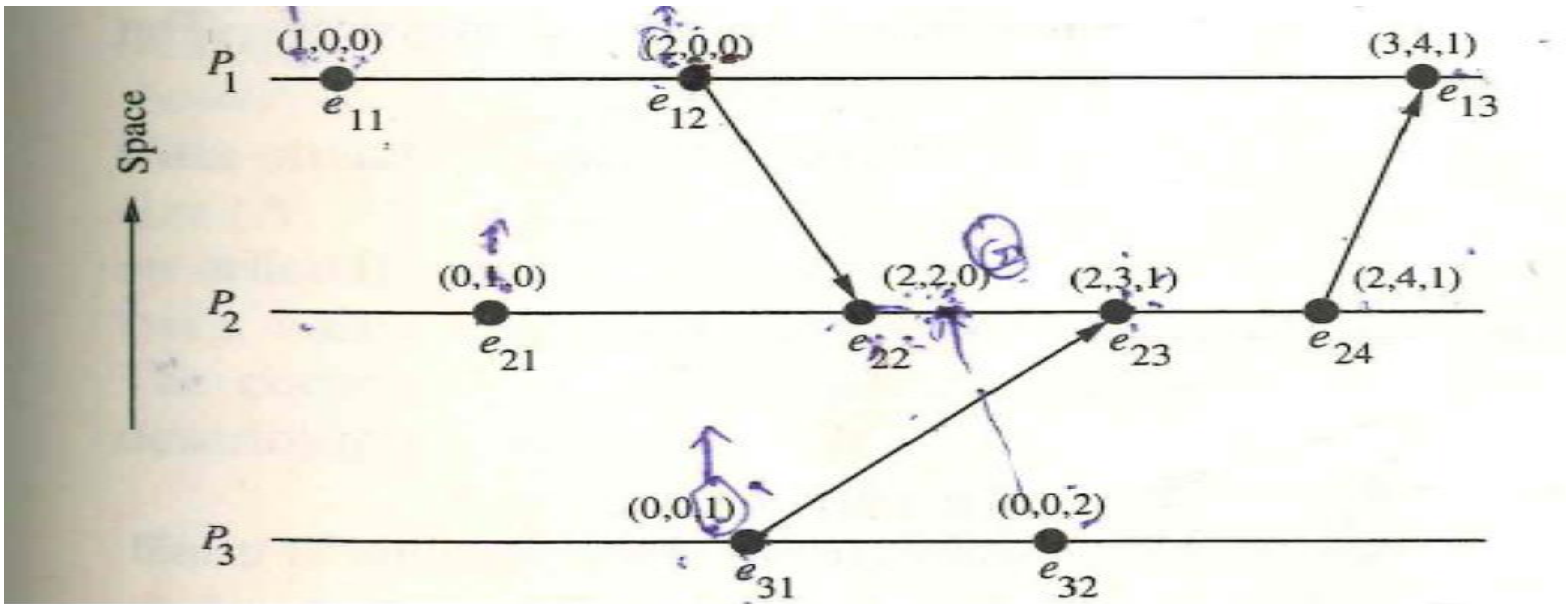
When that same message m is received by a different process Pk , Ck is updated as follows:

"p, Ck [p]:= max(Ck [p], tm [p] + d) (usually d=0 unless

- It can be shown that $\forall i, \forall k : C_i [i] \geq C_k [i]$
- Rules for comparing timestamps can also be established so that if

$t_a < t_b$, then $a \rightarrow b$

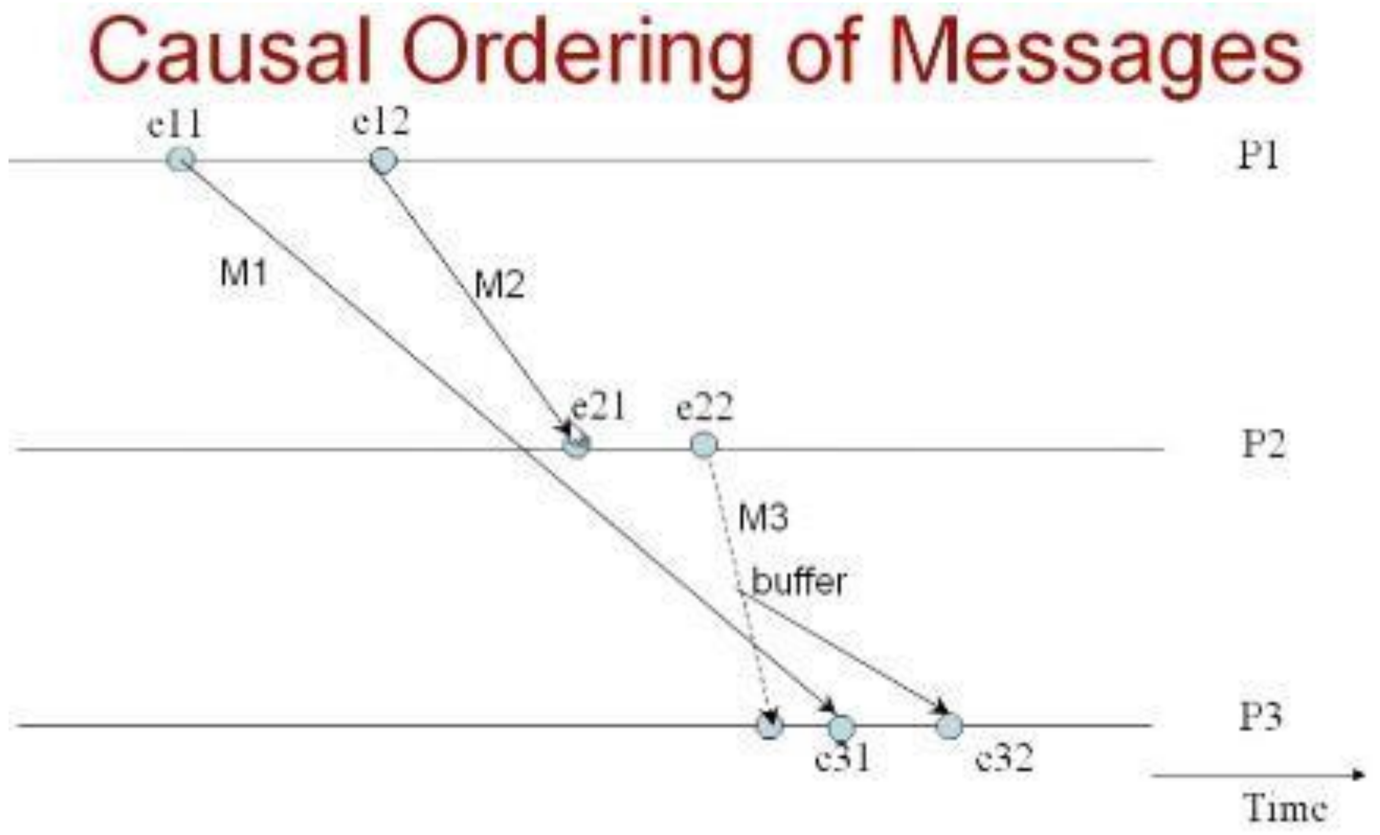
- Solves the problem with Lamport's clocks



Causal Ordering of Messages

The purpose of causal ordering of messages is to insure that the same causal relationship for the "message send" events correspond with "message receive" events. (i.e. All the messages are processed in order that they were created.)

Example



Send (M1) → Send (M2) → Receive (M2) → Send (M3)
So, Send (M1) and Send (M3) are causally related. <u>When two messages sent to a process are causally related, they should be received in the same order.</u>

Algorithms for Casual Ordering of Messages

- BSS: Birman-Scipher-Stephenson Algorithm

-Broadcast-based : Even when a process P1 has to send a message only to another process P2 the message has to be sent to all the processes in the system

More messages, Smaller size for the messages, limited state information

Notation

- n processes
- P_i& P_j
- C_i vector clock associated with process P_i; jth element is $C_i[j]$ and contains P_i's latest value for the current time in process P_j
- t^m vector timestamp for message m (stamped after local clock is incremented)

BSS: Birman-Scipher-Stephenson Algorithm

There are three basic principles to this algorithm:

1. All messages are time stamped by the sending process
2. A message can not be delivered until:
   - All the messages before this one have been delivered locally.
   - All the other messages that have been sent out from the original process has been accounted as delivered at the receiving process.
3. When a message is delivered, the clock is updated.

- This protocol requires that the processes communicate through broadcast messages since this would ensure that only one message could be received at any one time (thus concurrently timestamped messages can be ordered).

Algorithm

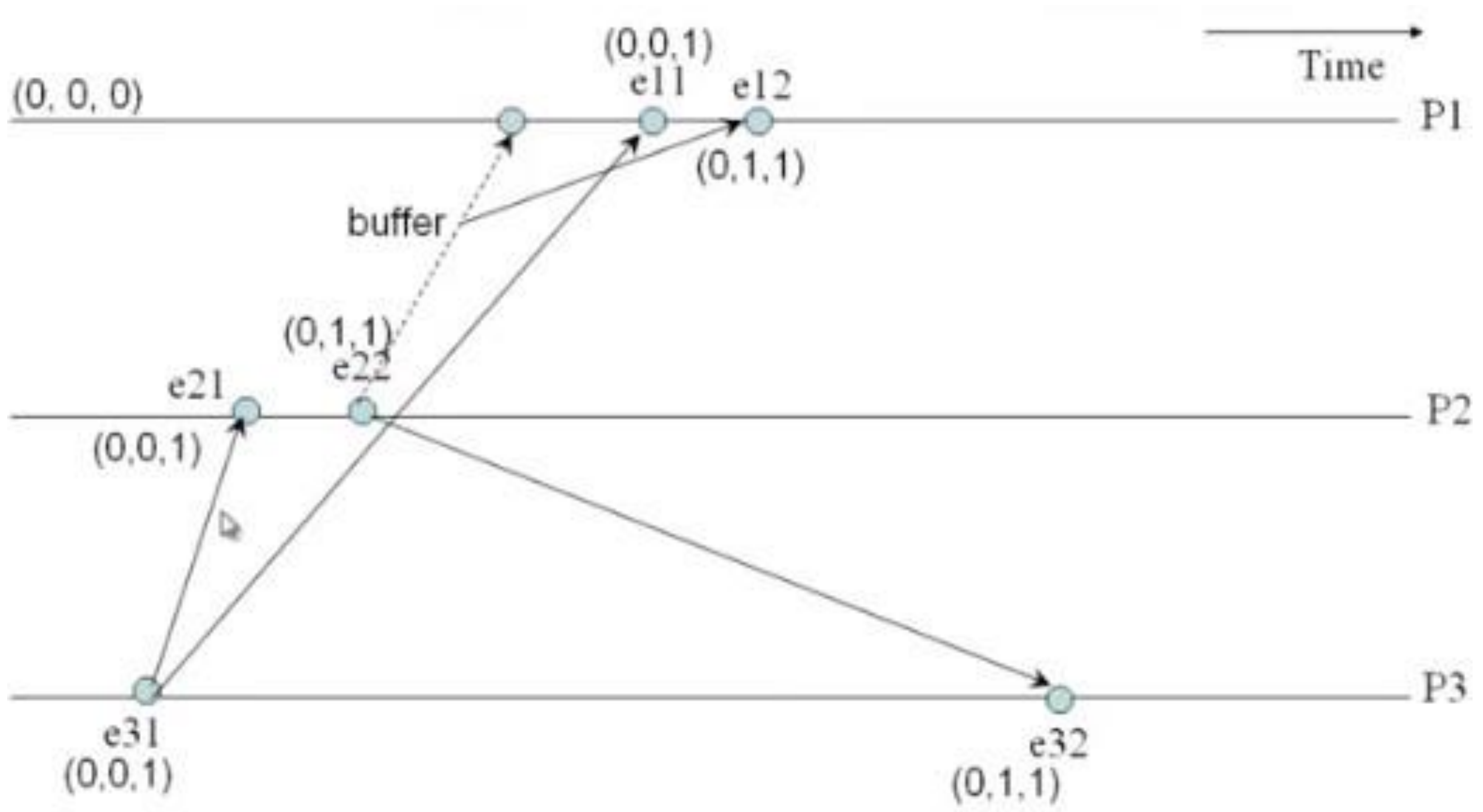
- ***P_i* sends a message to *P_j***

P_i increments $C_i[i]$ and sets the timestamp $t^m = C_i[i]$ for message *m*.

- ***P_j* receives a message from *P_i***

1. When P_j, $j \mathrel{!}= i$, receives *m* with timestamp t^m, it delays the message's delivery until both:
   a. $C_j[i] = t^m[i] - 1$; and
   b. for all $k \leq n$ and $k \mathrel{!}= i$, $C_j[k] \leq t^m[k]$.
2. When the message is delivered to P_j, update P_j's vector clock
3. Check buffered messages to see if any can be delivered.

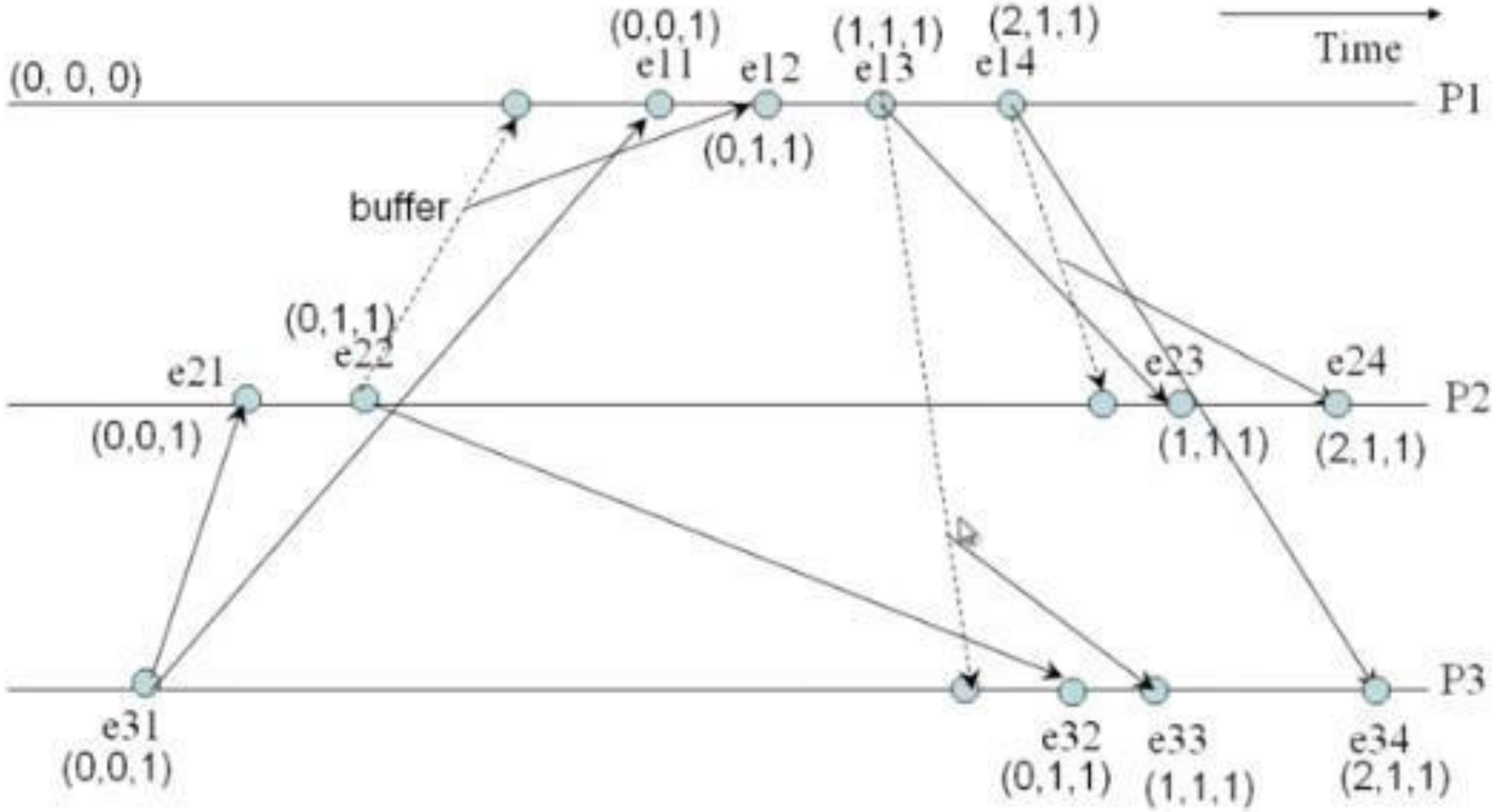
Example



Note: There is NO Clock Increment, upon receiving

Note: There is NO Clock Increment, upon receiving

Example


Time
(0, 0, 0)
(0,0,1)
e11
e12
(1,1,1)
e13
(2,1,1)
e14
P1
(0,1,1)
buffer
(0,1,1)
e22
e21
(0,0,1)
e23
(1,1,1)
e24
(2,1,1)
P2
e31
(0,0,1)
e32
(0,1,1)
e33
(1,1,1)
e34
(2,1,1)
P3

Schiper-Eggli-Sandoz Protocol

- No need for broadcast (works only by unicasting)

- Few messages, Moderate- Larger size for the messages, Lots of state information

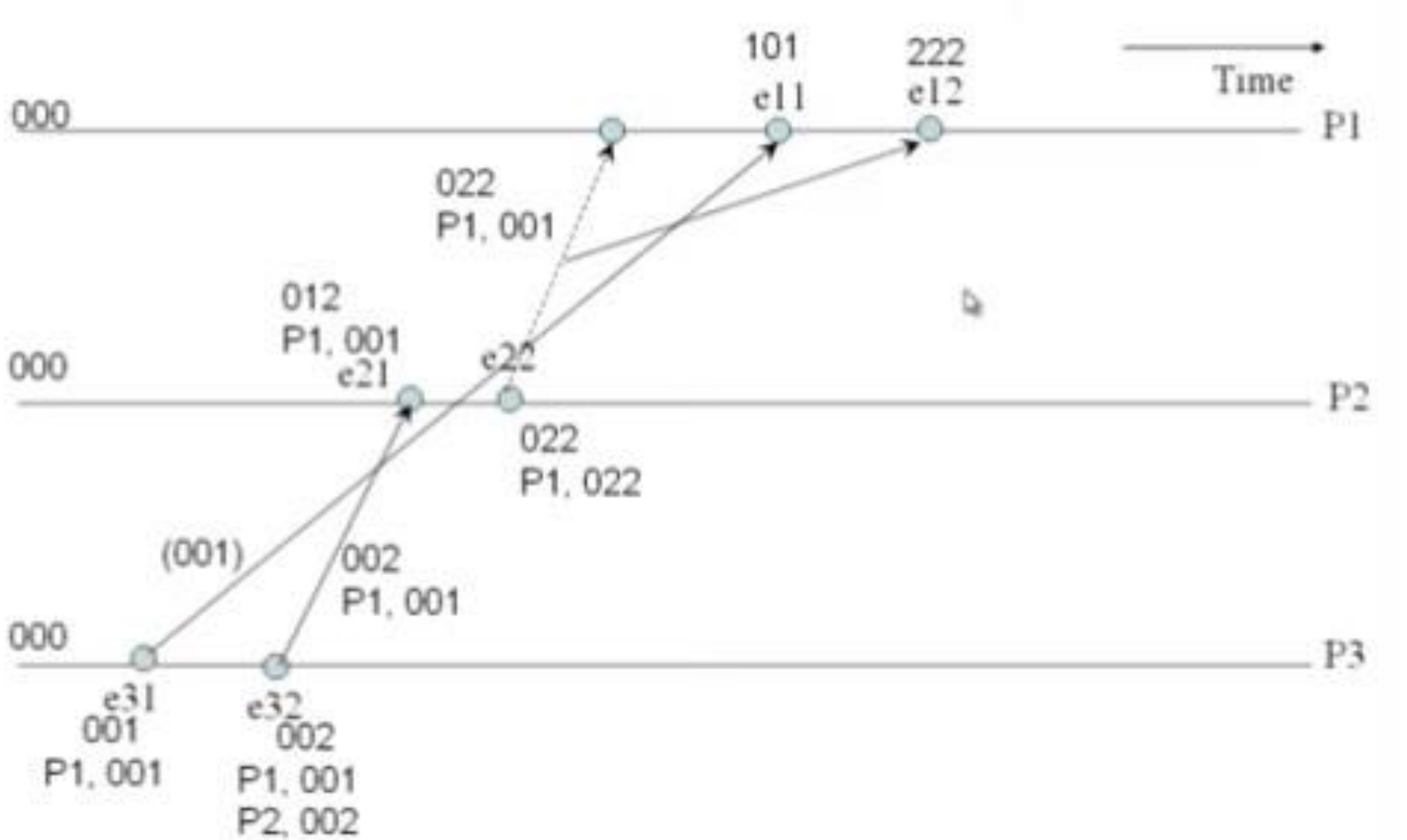
Sending a message:

- All messages are timestamped and sent out with a list of all the timestamps of messages sent to other processes.
- Locally store the timestamp that the message was sent with.

Receiving a message:

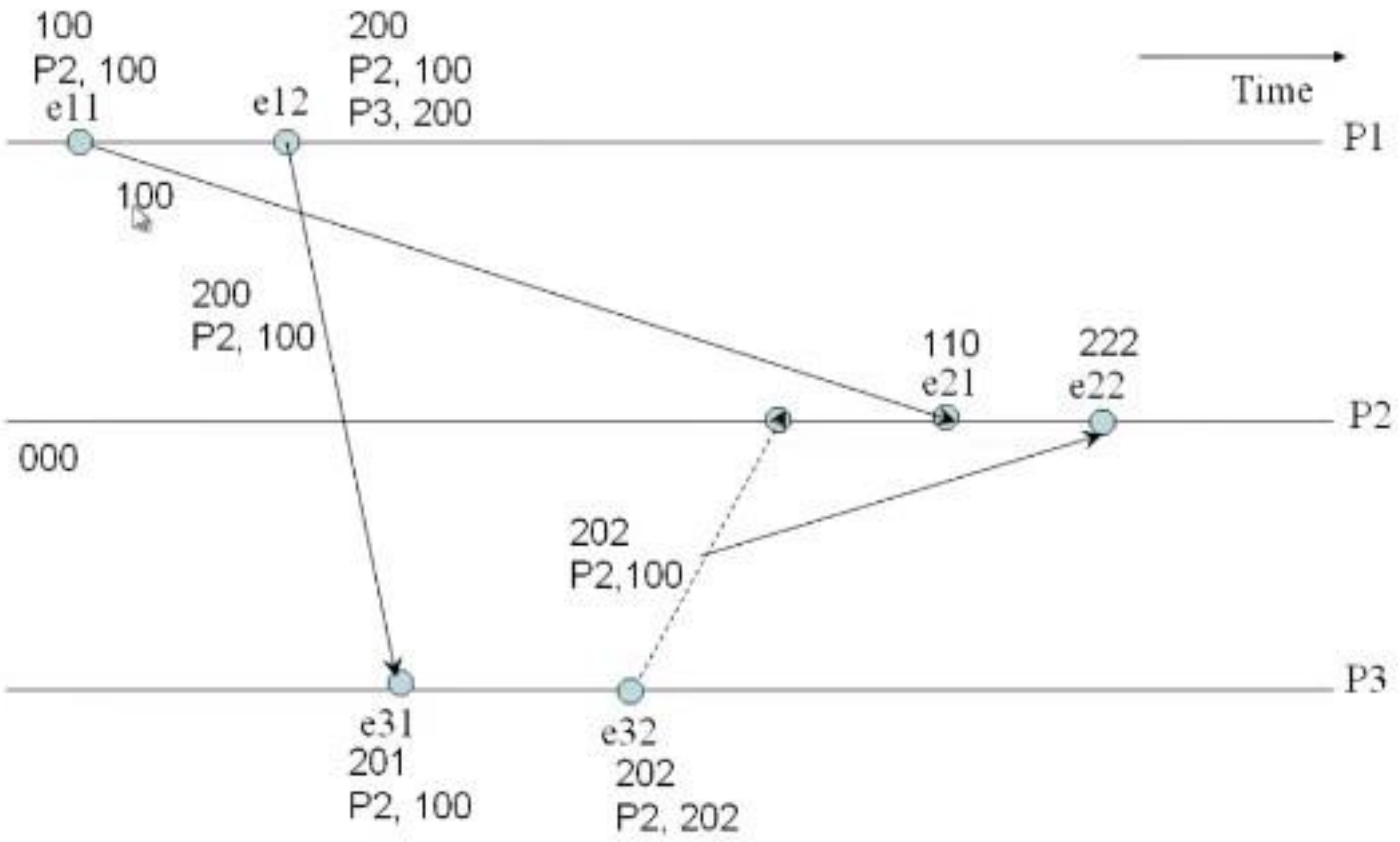
- A message cannot be delivered if there is a message mentioned in the list of timestamps that predates this one.
- Otherwise, a message can be delivered, performing the following steps:
  1. Merge in the list of timestamps from the message:
     a. Add knowledge of messages destined for other processes to our list of processes if we didn't know about any other messages destined for one already.
     b. If the new list has a timestamp greater than one we already had stored, update our timestamp to match.
  2. Update the local logical clock.
  3. Check all the local buffered messages to see if they can now be delivered.

Example



Note: There is <u>Clock Increment</u>, upon receiving

Example -2



Note: There is <u>Clock Increment</u>, upon receiving