# LECTURE 4

# VECTOR CLOCK SYNCHRONIZATION

Prof. D. S. Yadav
Department of Computer Science
IET Lucknow

# PROBLEMS WITH TOTAL ORDERING

- **A linearly ordered structure of time is not always adequate for distributed systems**
    - captures dependence of events
    - loses independence of events - artificially enforces an ordering for events that need not be ordered.

- **Mapping partial ordered events onto a linearly ordered set of integers it is *losing information***
    - Events which may happen simultaneously may get different timestamps as if they happen in some definite order.

- **A partially ordered system of *vectors* forming a *lattice* structure is a natural representation of time in a distributed system**

# WHY DO WE NEED GLOBAL CLOCKS?

- For causally ordering events in a distributed system
  - Example:
    - Transaction T transfers Rs 10,000 from S1 to S2
    - Consider the situation when:
      - State of S1 is recorded after the deduction and state of S2 is recorded before the addition
      - State of S1 is recorded before the deduction and state of S2 is recorded after the addition
- Should not be confused with the clock-synchronization problem

# VECTOR TIME (FIDGE/MATTERN/SCHMUCK)

- The system of vector clocks was developed independently by Fidge, Mattern and Schmuck.
- In the system of vector clocks, the time domain is represented by a set of n-dimensional non-negative integer vectors.
- Each process has a clock $C_i$ consisting of a vector of length $n$, where $n$ is the total number of processes.
- VTPi[1..n], where VTPi [j ] is the local logical clock of Pi and describes the logical time progress at process Pj .

# ORDERING OF EVENTS

Lamport's *Happened Before* relationship:

For two events a and b, $a \rightarrow b$ if

- *a and b are events in the same process and a occurred before b, or*
- *a is a send event of a message m and b is the corresponding receive event at the destination process, or*
- *$a \rightarrow c$ and $c \rightarrow b$ for some event c*

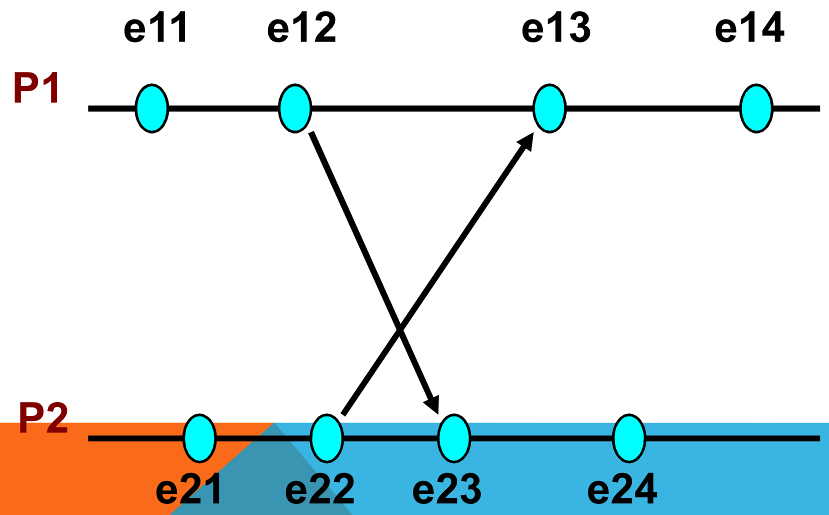
# CAUSALLY RELATED VERSUS CONCURRENT

**Causally related events:**

- Event a causally affects event b if a $\rightarrow$ b

**Concurrent events:**

- Two distinct events a and b are said to be concurrent ( denoted by a||b ) if a $\not\rightarrow$ b and b $\not\rightarrow$ a




e11 and e21 are concurrent

e14 and e23 are concurrent

e22 causally affects e14

A space-time diagram

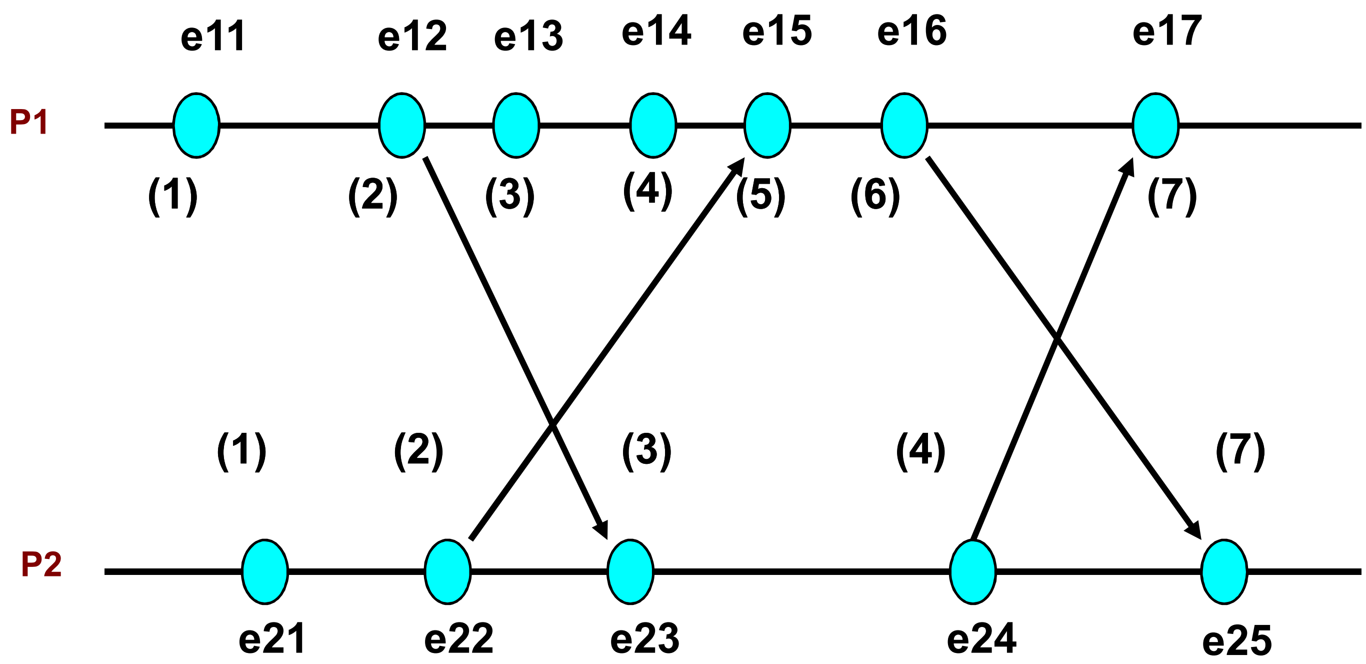
# LAMPORT'S LOGICAL CLOCK

***Each process i keeps a clock $C_i$***

- Each event a in i is time-stamped $C_i(a)$, the value of $C_i$ when a occurred
- $C_i$ is incremented by 1 for each event in i
- In addition, if a is a send of message m from process i to j, then on receive of m,

$$C_j = \max(C_j, C_i(a)+1)$$

# HOW LAMPORT'S CLOCKS ADVANCE



# IN LAMPORT'S CLOCK

- if a $\rightarrow$ b, then C(a) < C(b)
- $\rightarrow$ is a partial order
- Total ordering possible by arbitrarily ordering concurrent events by process numbers

# LIMITATION OF LAMPORT'S CLOCK

$a \longrightarrow b$ implies $C(a) < C(b)$

**BUT**

$C(a) < C(b)$ doesn't imply $a \longrightarrow b$ !!

*So not a true clock !!*

# VECTOR TIME: THE PROTOCOL

- A process $P_i$ ticks by incrementing its own component of its clock

$$C_i[i] = C_i[i] + 1$$

- The timestamp C(e) of an event e is the clock value after ticking.
- Each message gets a piggybacked timestamp consisting of the vector of the local clock.
  - The process gets some knowledge about the other process' time approximation.

# LOGICAL CLOCKS : VECTOR CLOCK

- Vector Clock uses a vector of Integers of size N, where N is number of processes in system.
- Process $P_i$ maintains a vector clock $VT_i$.
- $VT_i$ [ i ] is process $P_i$'s own logical time.
- $VT_i$ [ j ] is process $P_i$'s best knowledge of time at process $P_j$.
- Proposed by Fidge and Mattern and based on Lamport's scalar clocks

# VECTOR CLOCKS : UPDATE RULES

**Each process $P_i$ has a clock $C_i$, which is a vector of size n**

**The clock $C_i$ assigns a vector $C_i(a)$ to any event *a* at $P_i$**

**Update rules (Internal & Message Events)**

[IR1] : Clock $C_i$ is incremeted beween two successive events in process $P_i$

$$C_i[i] = C_i[i] + d$$

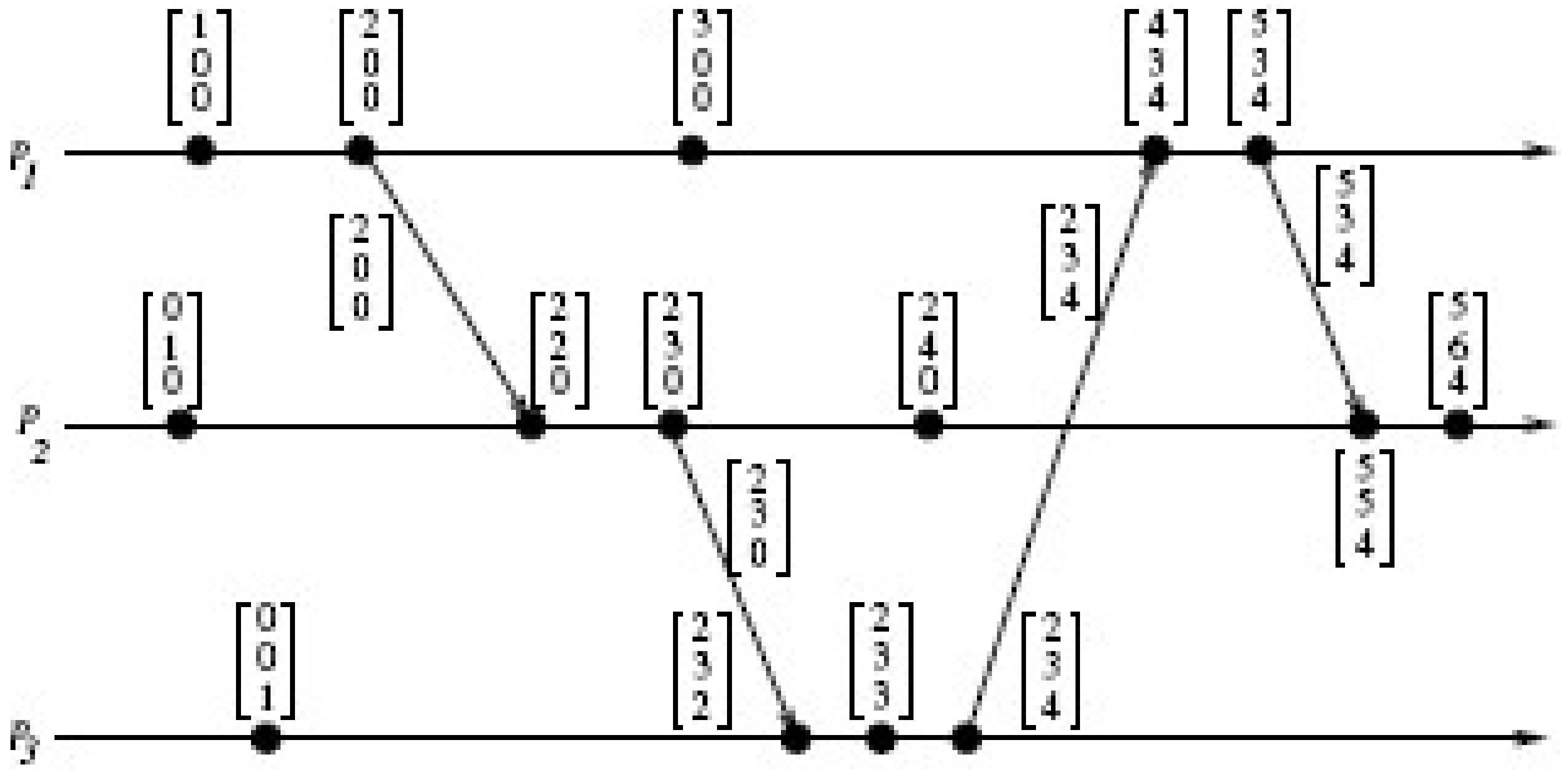
[IR2]When $P_i$ sends a message m to Process $P_j$ , it piggybacks a logical timestamp t which equals the time of the send event : $t(m) = C_i$

When executing a receive event at $P_j$ where a message with timestamp *t* is received, the clock is advanced

- $C_J[J] = C_i[J] + d$

  $C_j[k] = \max(C_j[k], t_m[k])$ **for all k**

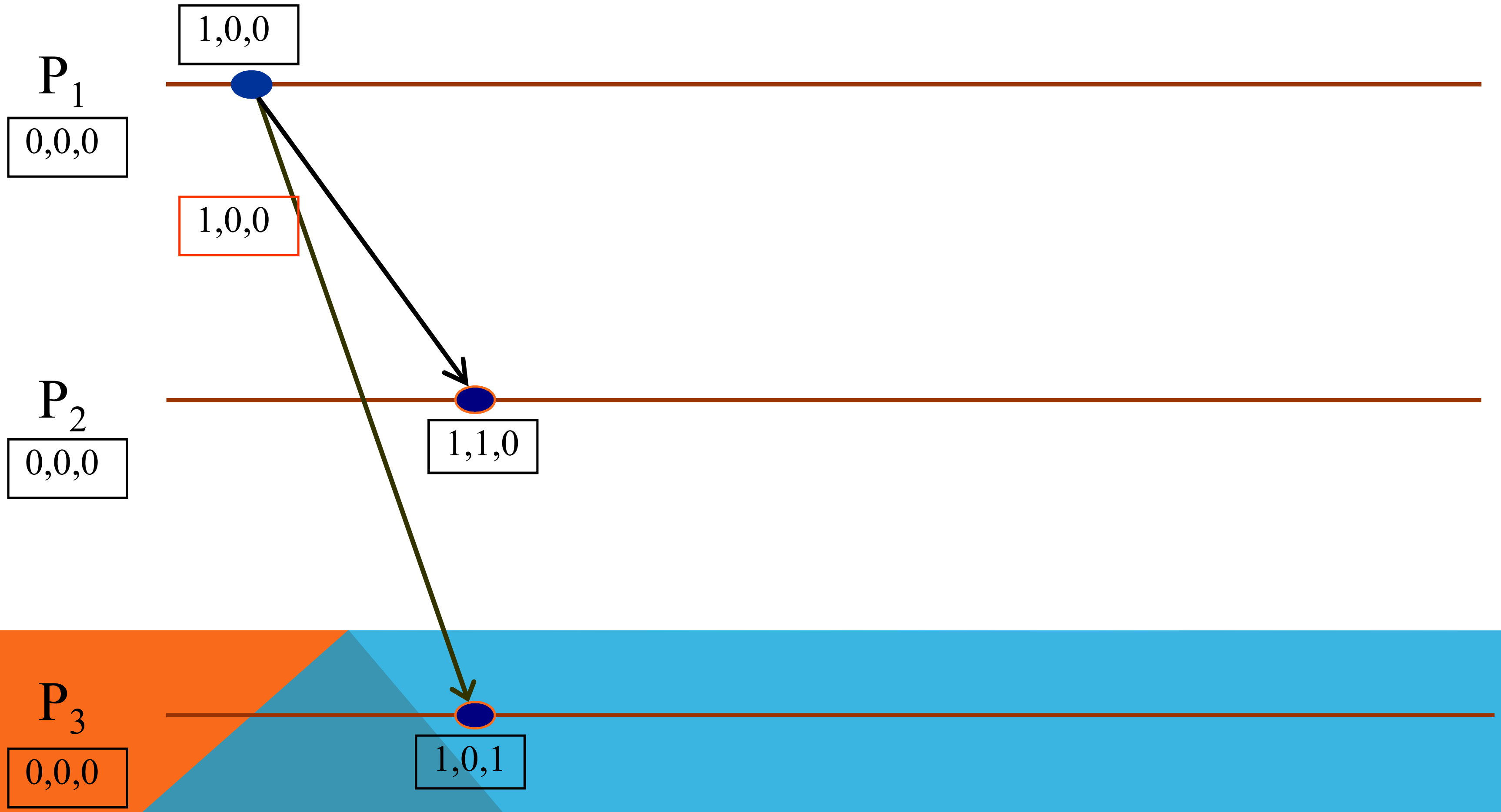
# VECTOR CLOCKS EXAMPLE



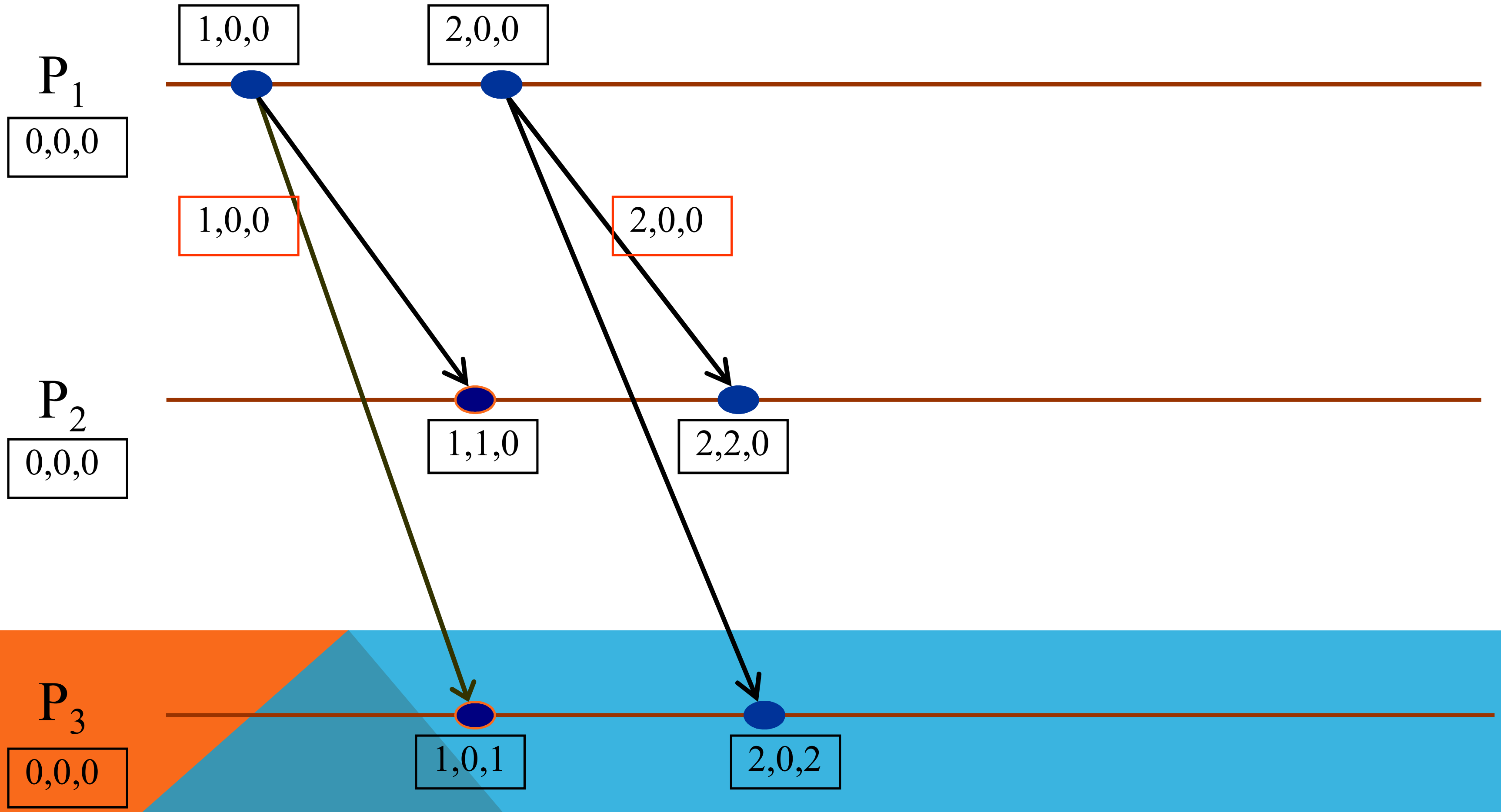

Evolution of vector time.

From A. Kshemkalyani and M. Singhal (Distributed Computing)

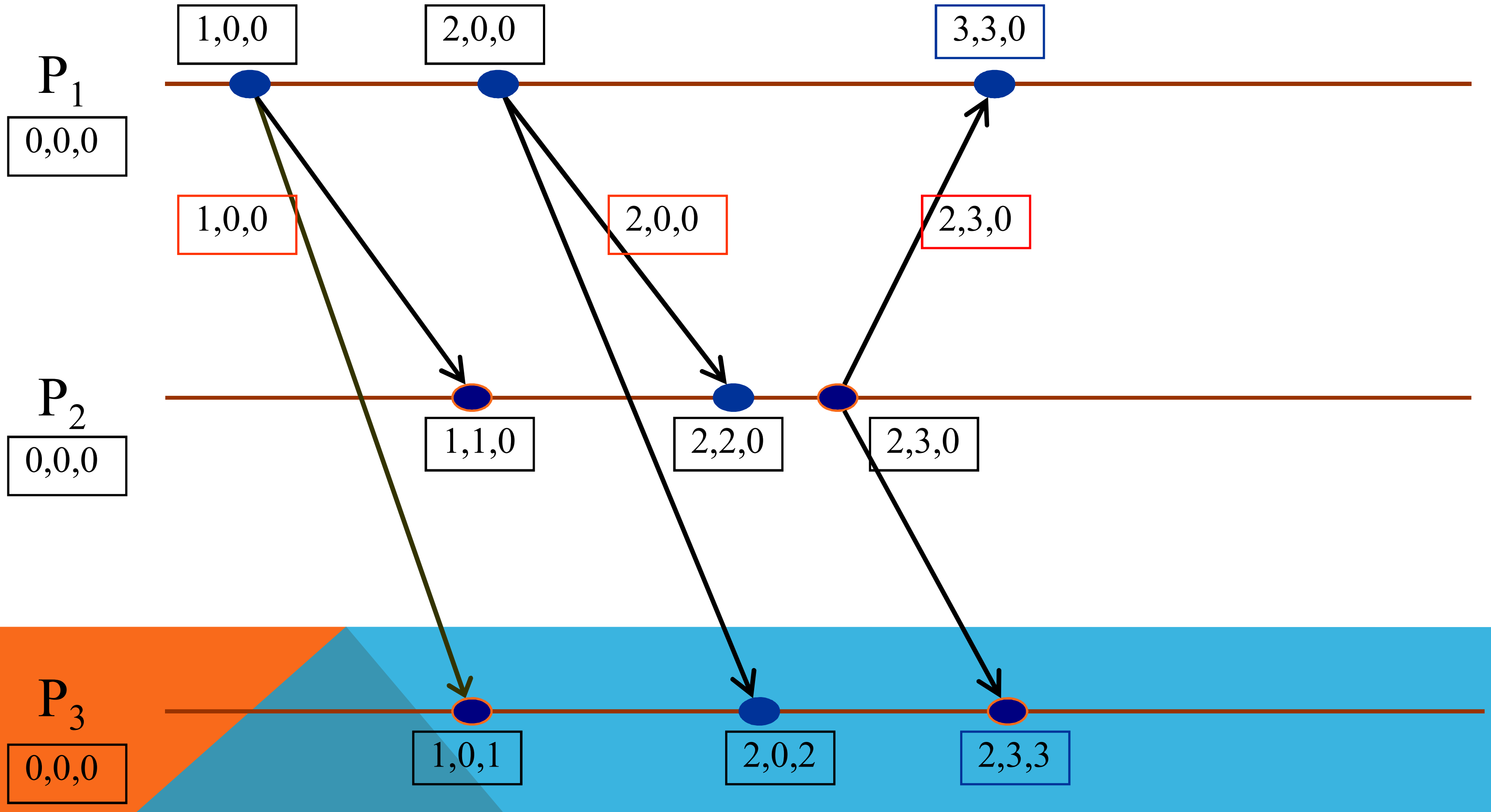
# APPLYING VECTOR CLOCKS
## TO BROADCAST SYSTEM



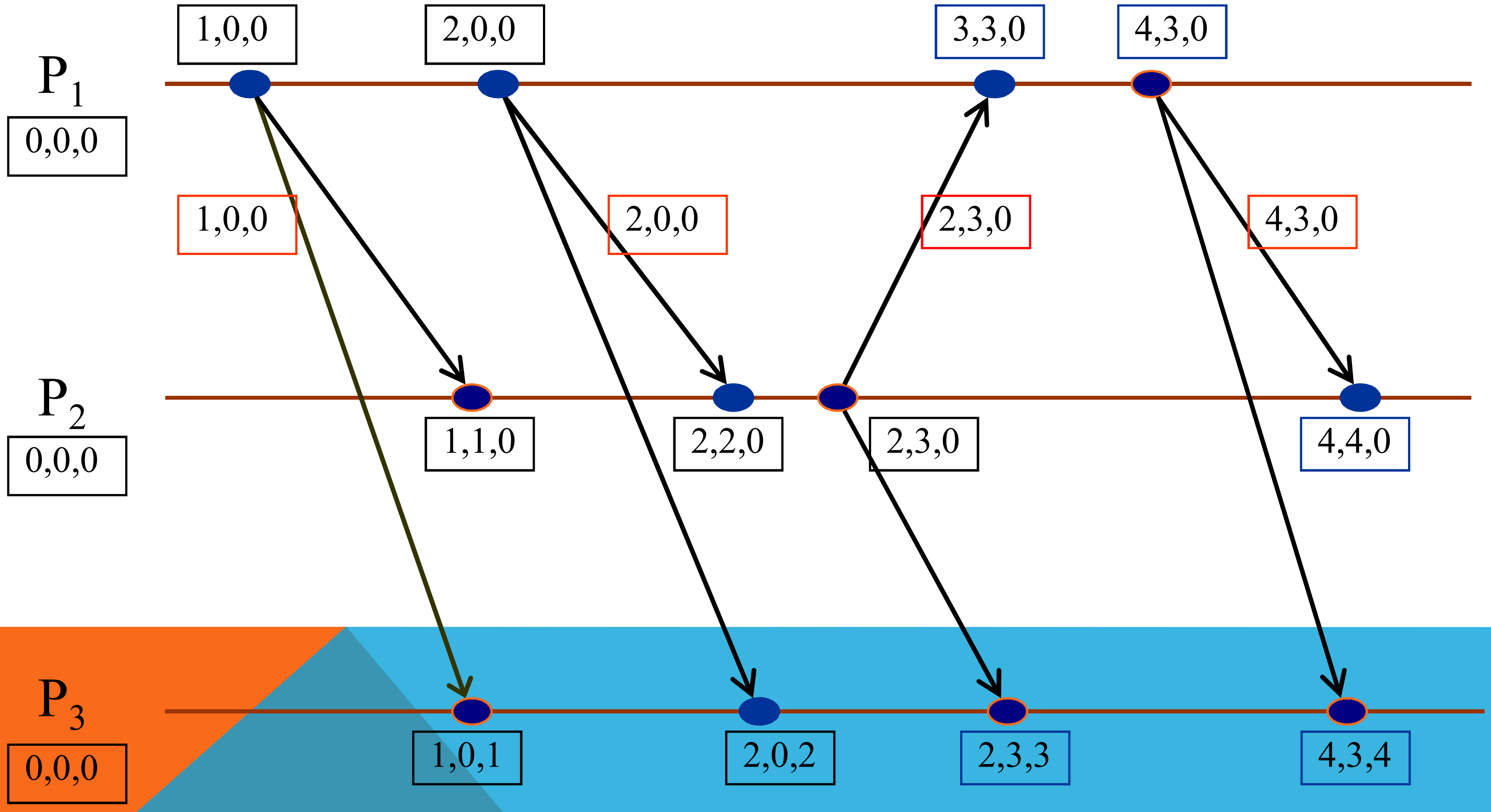
# APPLYING VECTOR CLOCKS
## TO BROADCAST SYSTEM



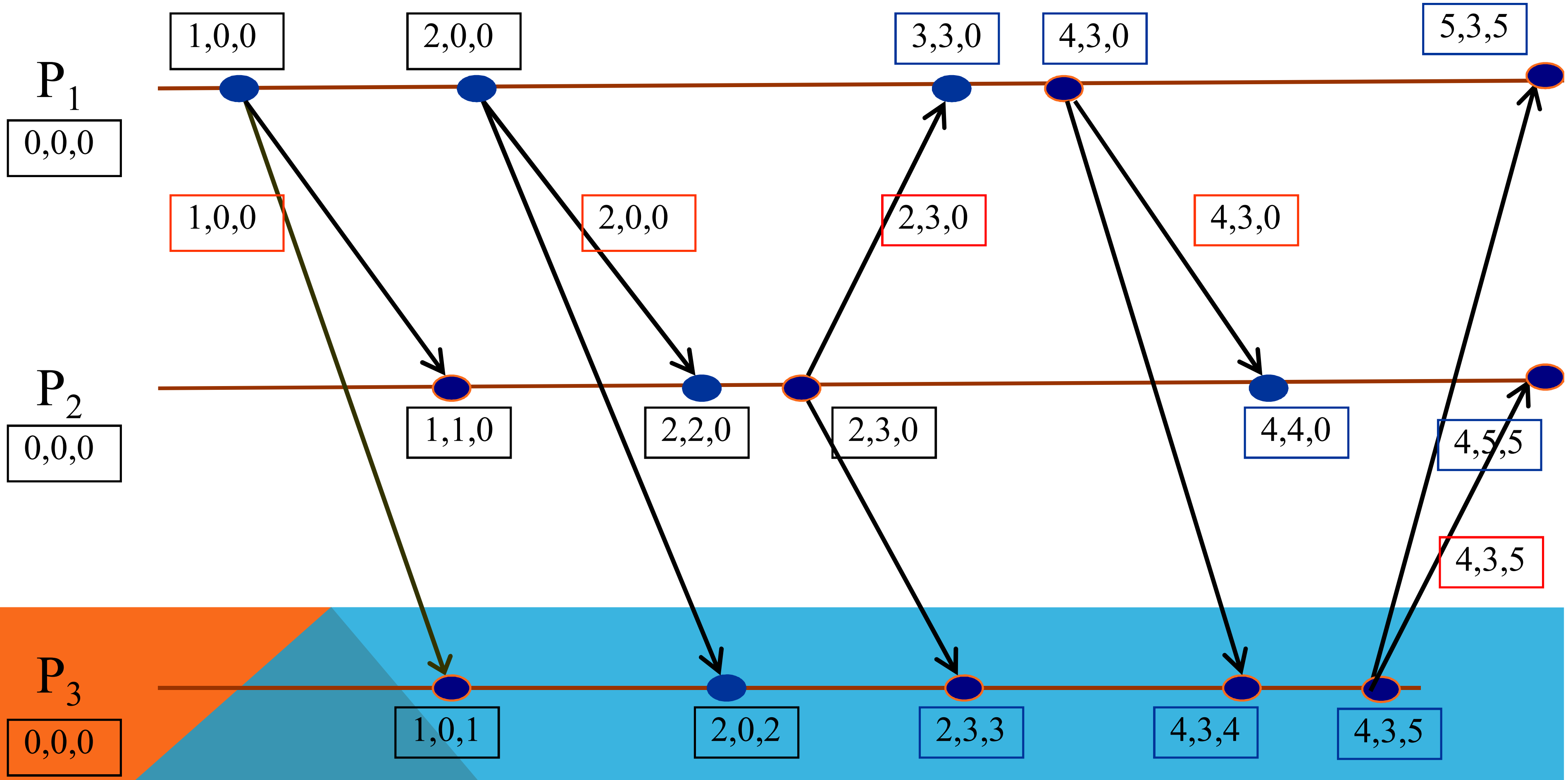
# APPLYING VECTOR CLOCKS
## TO BROADCAST SYSTEM



# APPLYING VECTOR CLOCKS
## TO BROADCAST SYSTEM



# APPLYING VECTOR CLOCKS
## TO BROADCAST SYSTEM



# PARTIAL ORDER BETWEEN TIMESTAMPS

For events a and b with vector timestamps $t^a$ and $t^b$,

- Equal: $t^a = t^b$ iff $\forall i, t^a[i] = t^b[i]$
- Not Equal: $t^a \neq t^b$ iff $\exists i, t^a[i] \neq t^b[i]$
- Less or equal: $t^a \leq t^b$ iff $\forall i, t^a[i] \leq t^b[i]$
- Not less or equal: $t^a \nleq t^b$ iff $\exists i, t^a[i] > t^b[i]$
- Less than: $t^a < t^b$ iff $(t^a \leq t^b$ and $t^a \neq t^b)$
- Not less than: $t^a \nless t^b$ iff $\neg(t^a \leq t^b$ and $t^a \neq t^b)$
- Concurrent: $t^a \| t^b$ iff $(t^a \nless t^b$ and $t^b \nless t^a)$

# CAUSAL ORDERING

- $a \longrightarrow b$ iff $t_a < t_b$
- Events a and b are causally related iff $t_a < t_b$ or $t_b < t_a$, else they are concurrent
- Note that this is still not a total order

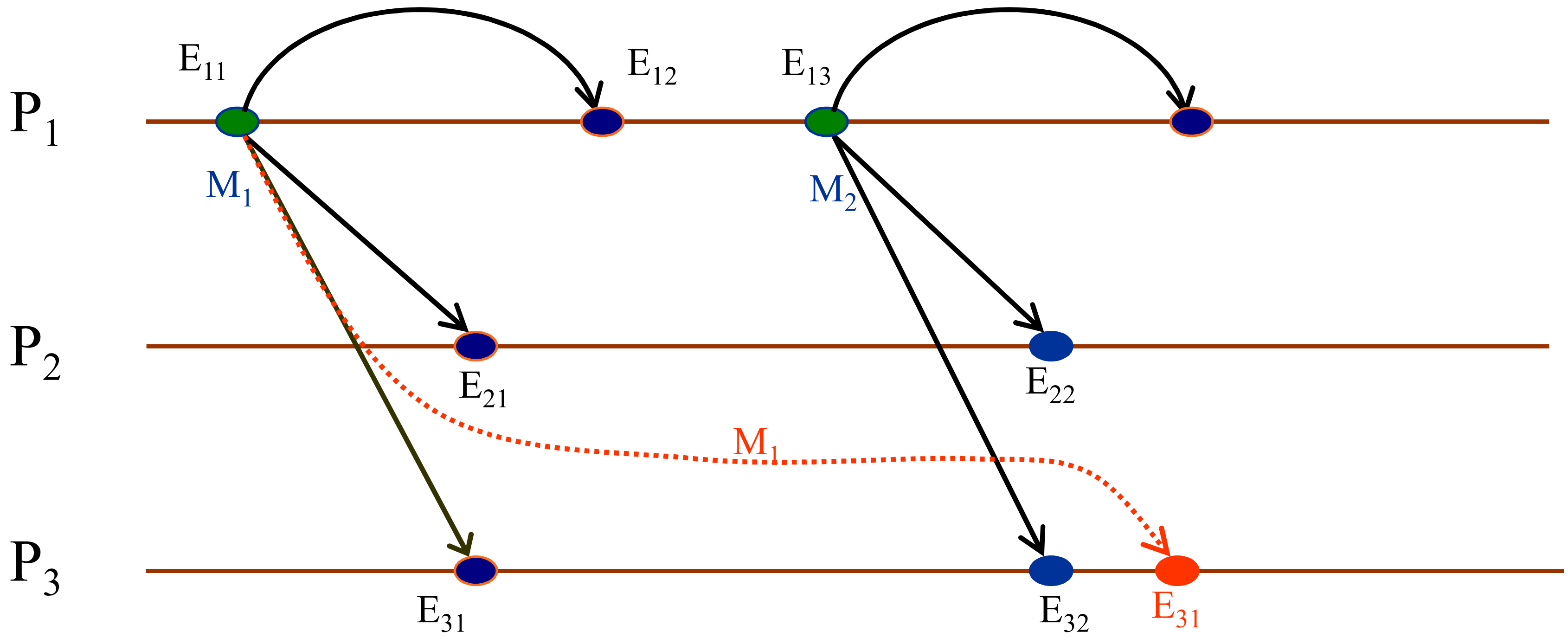
# USE OF VECTOR CLOCKS IN CAUSAL ORDERING OF MESSAGES

- If send(m1) → send(m2), then every recipient of both message m1 and m2 must “deliver” m1 before m2.

    – “deliver” – when the message is actually given to the application for processing

# FIFO ORDER



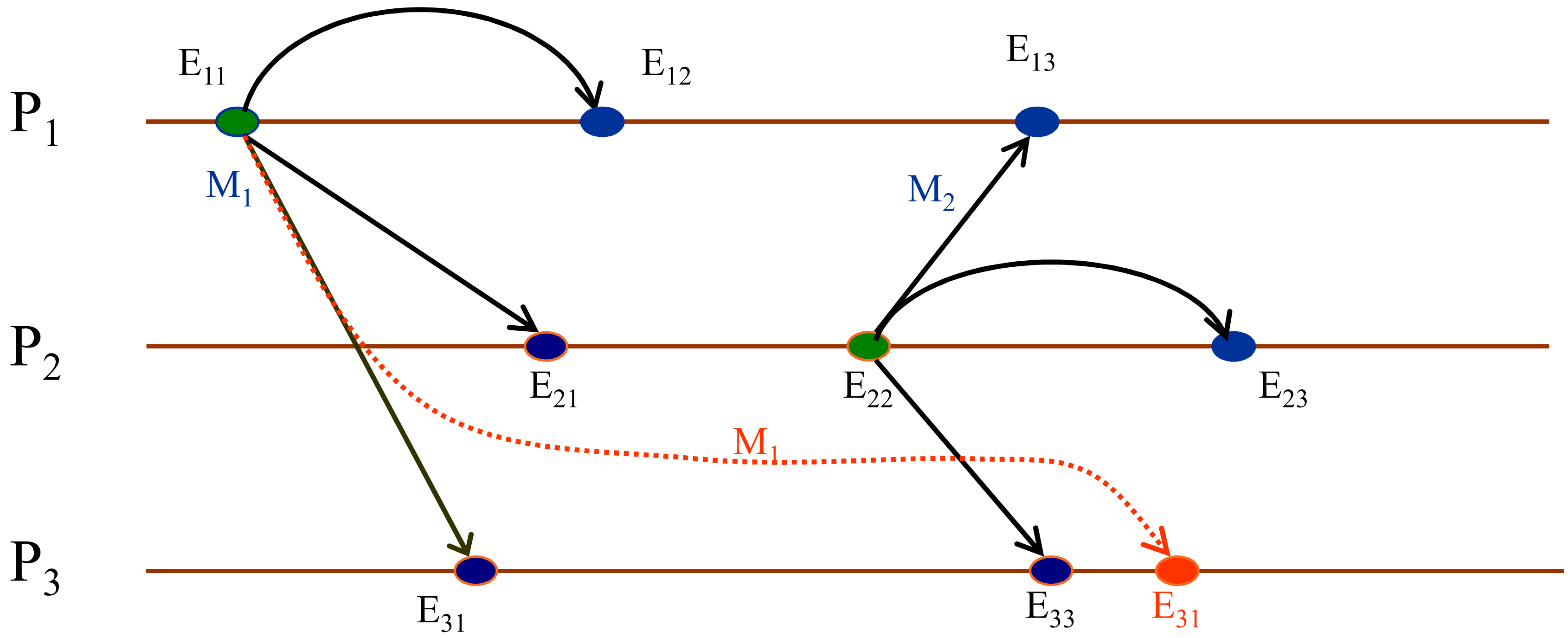



If a particular process broadcasts a message $M_1$ before it broadcasts a message $M_2$ , then each recipient process delivers $M_1$ before $M_2$.

- Message $M_1$ ( ) shows violation of FIFO order.

# Local Order





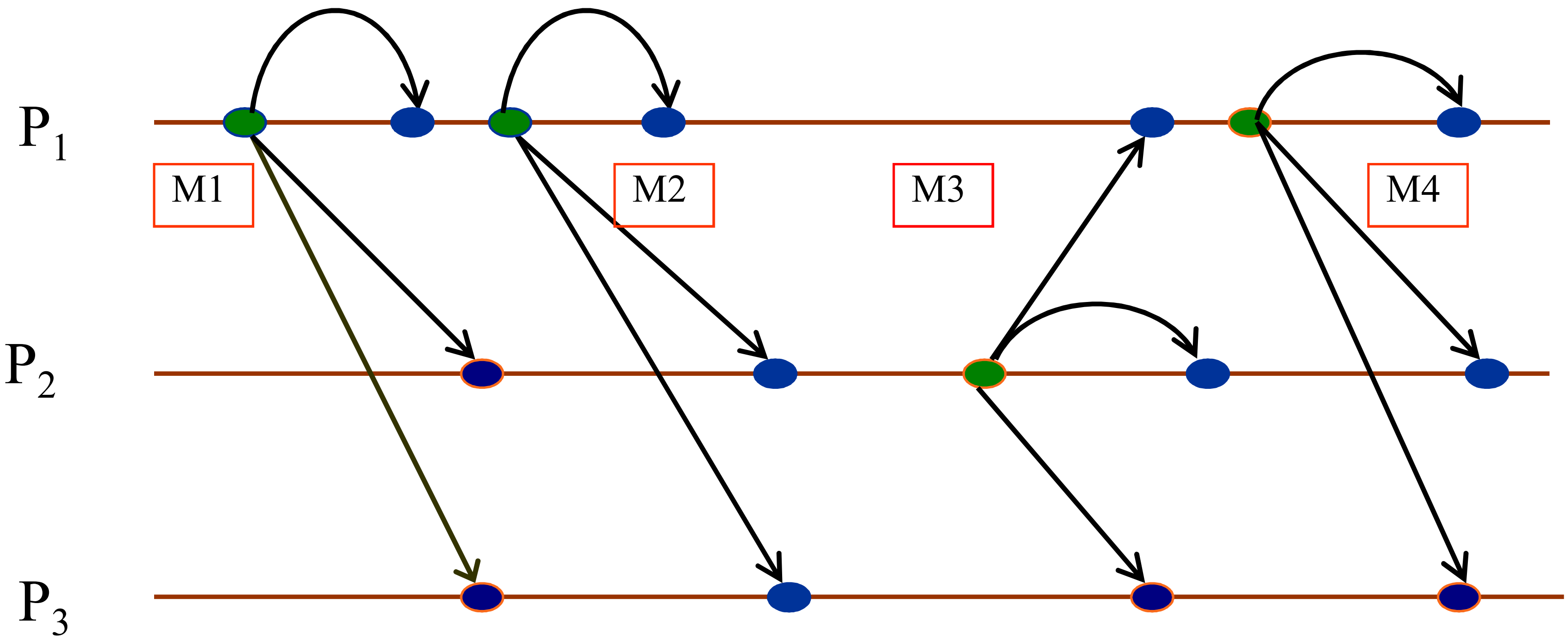

If a process delivers a message $M_1$ before it broadcasts a message $M_2$, then each recipient process delivers $M_1$ before $M_2$.

- Message $M_1$ ( ) shows violation of global causal ordering.

# Causal Order



Delivery order : M1,M2,M3,M4




If the broadcast of a message $M_1$ *causally precedes* the broadcast of a message $M_2$ , then no process delivers $M_2$ unless it has previously delivered $M_1$.

$M1 \prec M2$

$M2 \prec M3$

$M1 \prec M2 \;\wedge\; M2 \prec M3 \;\Longrightarrow\; M1 \prec M3$

# SIMPLE SOLUTION TO ENSURE DELIVERY OF MESSAGES IN CAUSAL ORDER

## BIRMAN-SCHIPER-STEPHENSON PROTOCOL

**REFERENCE:**

1.BIRMAN, JOSHEPH,"RELIABLE COMMUNICATION IN PRESENCE OF FAILURE", ACM TRANSACTIONS ON COMPUTER SYSTEMS,5(1), 1987.

2. BIRMAN, SCHIPHER, STEPHENSON," LIGHWEIGHT CAUSAL AND ATOMIC GROUP MULTICAST". ACM TRANSACTIONS ON COMPUTER SYSTEMS, 9(3),1991

# BIRMAN-SCHIPER-STEPHENSON PROTOCOL

## BASIC IDEA

BASIC IDEA OF PROTOCOL IS TO DELIVER A MESSAGE TO A PROCESS ONLY IF THE MESSAGE IMMEDIATELY PRECEDING IT HAS BEEN DELIVERED

OTHERWISE THEY ARE BUFFERED UNTIL PRECEDING MESSAGE ARE DELIVERED.

A VECTOR ACCOMPANYING EACH MESSAGE CONTAIN INFORMATION FOR A RECIPIENT PROCESS TO DECIDE IF ALL PRECEDING MESSAGE HAS BEEN DELIVERED.

# VECTOR CLOCKS : MODIFIED UPDATE RULES

**Each process $P_i$ has a clock $C_i$, which is a vector of size n**

**The clock $C_i$ assigns a vector $C_i(a)$ to any event *a* at $P_i$**

**Update rules (Clock is updated only on Message Events)**

**[IR1 Broadcast Rule]** : Clock $C_i$ is incremented when a message is **broadcast** by a process $P_i$

$$C_i[i] = C_i[i] + d$$

**[IR2 Delivery Rule]**

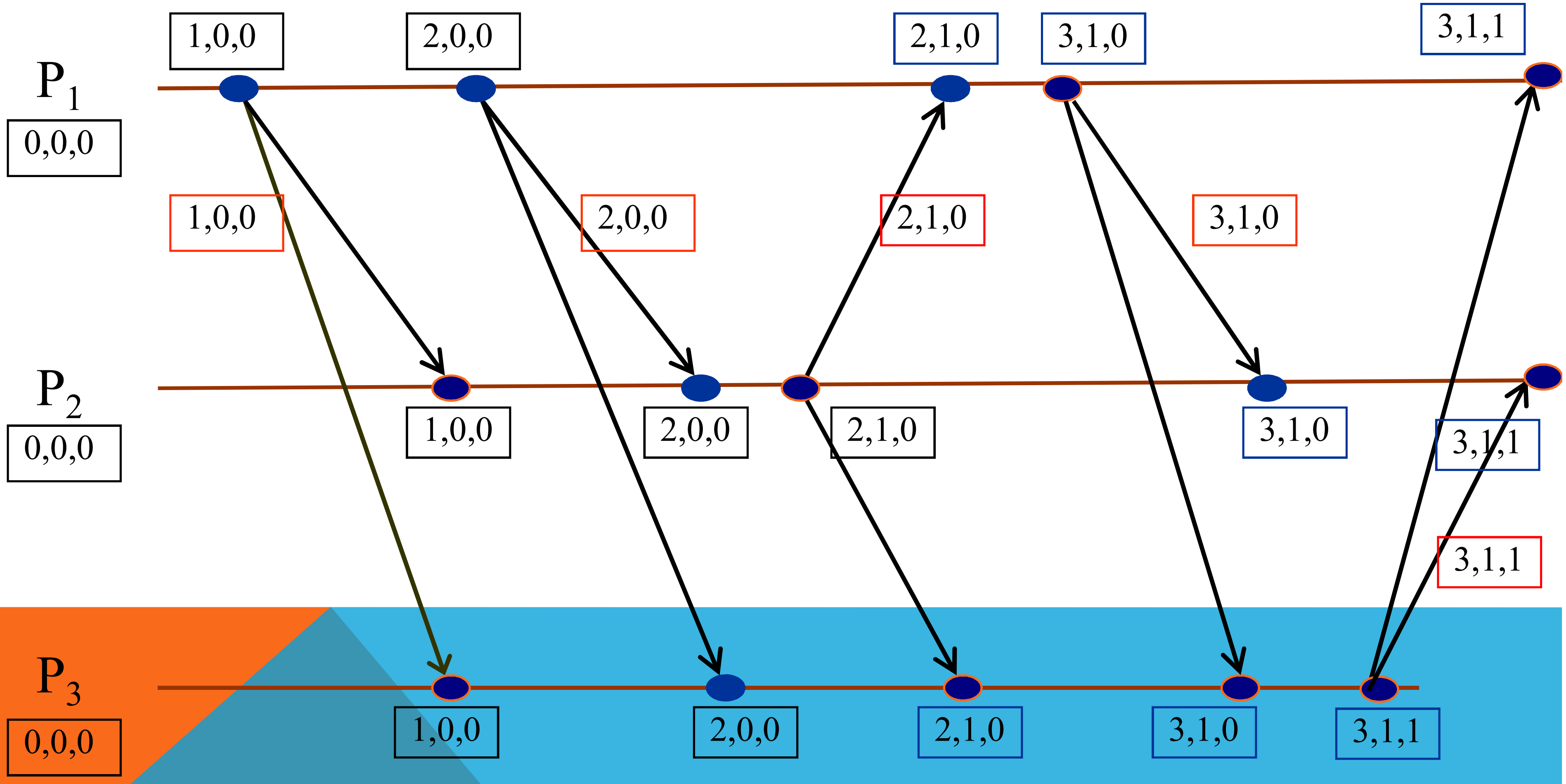
When $P_i$ sends a message m to Process $P_j$ , it piggybacks a logical timestamp t which equals the time of the send event : $t(m) = C_i$

When a **message is received** by a process $P_j$ where a message with timestamp *t* is received, the clock is advanced.

- $C_j[k] = \max(C_j[k], t_m[k])$ **for all k**
- **(Please note the change)**

# APPLYING VECTOR CLOCKS TO BROADCAST SYSTEM

(AS PER UPDATED RULES)



# PROBLEMS OF VECTOR CLOCK

- Message size increases since each message needs to be tagged with the vector
- Size can be reduced in some cases by only sending values that have changed

## Some Good Observations

In a system where vector clock is updated on message send and receive events only

- **$VT_i$ [ i ] indicates number of messages sent by process Pi .**
- **$VT_j$ [ i ] indicates number of messages received by process Pj sent by process Pi .**

# BIRMAN-SCHIPER-STEPHENSON PROTOCOL

- To broadcast M from process **Pi**, increment $VT_{Pi}[i]$ as $VT_{Pi}[i] = VT_{Pi}[i] + 1$ and timestamp M with $VT_m = VT_{Pi}$.

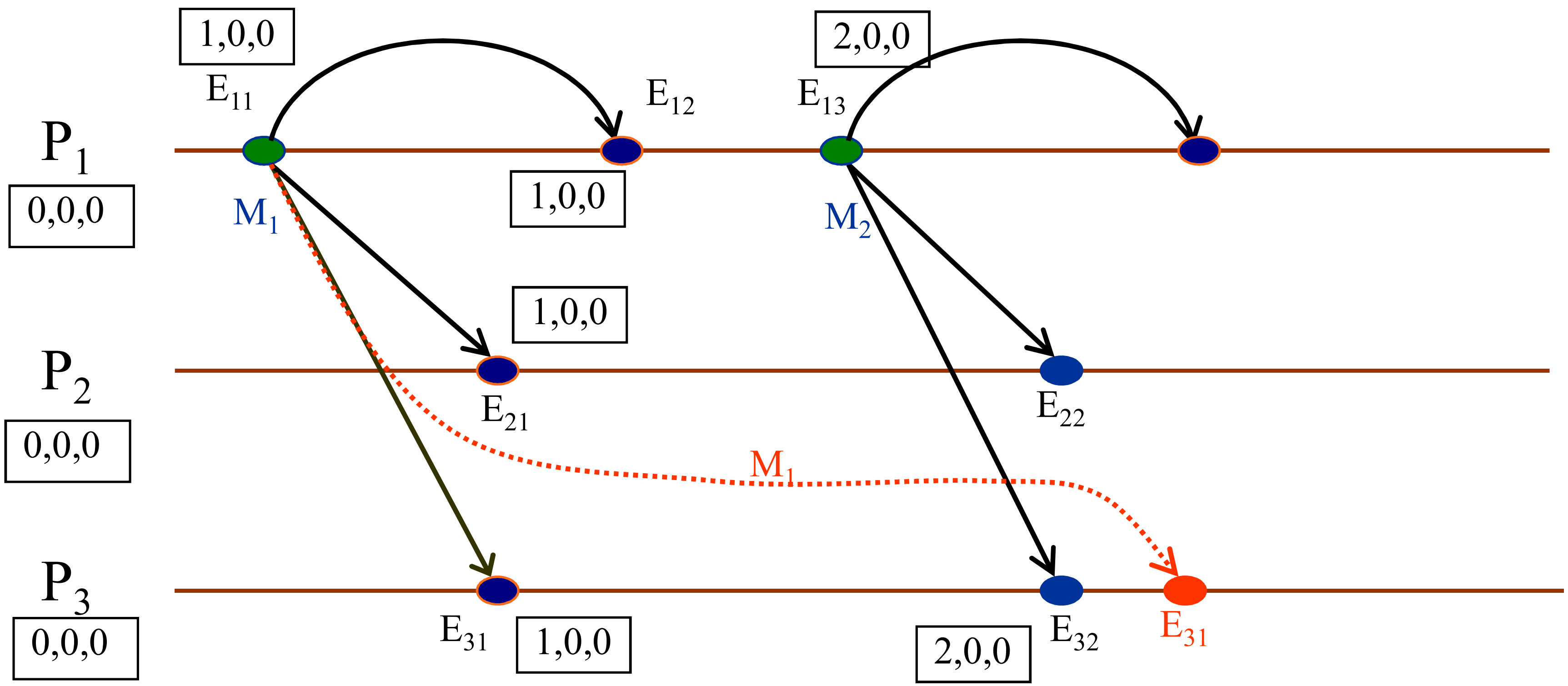
  ( Note : $VT_{Pi}[i] - 1$ indicate how many messages from Pi precedes M)

- When $j \neq i$ receives m, j delays delivery of m until
  - $VT_{Pj}[i] = VT_m[i] - 1$ and
  - $VT_{Pj}[k] \geq VT_m[k]$ for all $k \neq i$ ( i.e., $\forall K \cdot K \in \{1,2,3....,n\} - \{i\}$ )
  - Delayed messages are queued in j sorted by vector time. Concurrent messages are sorted by receive time.

- When M is delivered at **Pj** , $VT_{Pj}$ is updated according to vector clock rule.

$$C_j[k] = \max(C_j[k], t_m[k]) \text{ for all } k$$

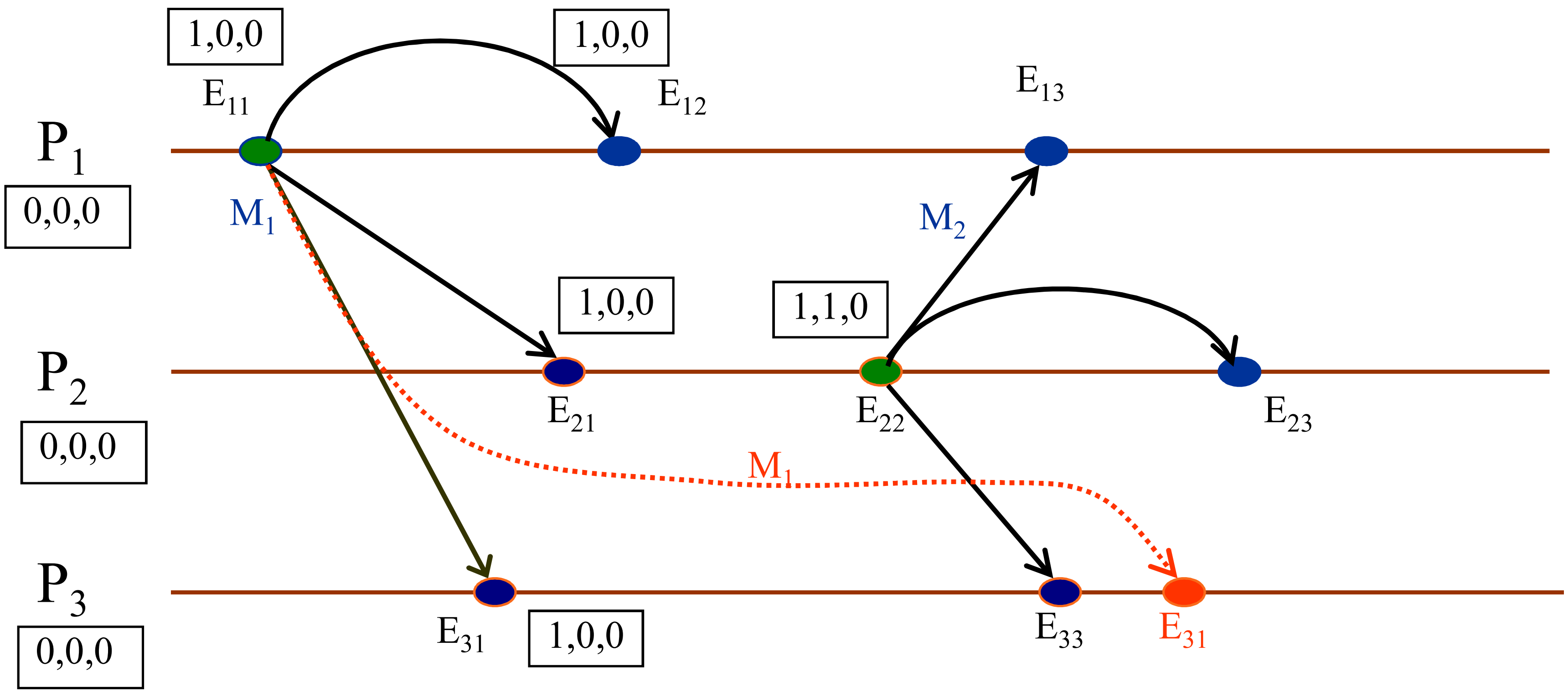
# FIFO ORDER




If a message $M_1$ is delayed and M2 arrives earlier, M2 will not be delivered as VTM2 [1]-1 is not equal to VTP3[1]

- Message $M_1$ ( ) shows violation of FIFO order.

# Local Order




If a process P3 receives $M_2$ before $M_1$ , then $M_2$ will be buffered until receipt of M1. As
VTM2 =[1,1,0] & VTP3 [0,0,0] ( VTP3[1] < VTM2[1] )

- Message $M_1$ ( ) shows violation of global causal ordering.

# END OF LECTURE 4