Distributed Systems
分布式系统

Distributed Time and Clock Synchronization (2)

分布式时间和时间同步

Logical Time

逻辑时间

Motivation of logical clocks

- Cannot synchronize physical clocks perfectly in distributed systems. [Lamport 1978]

- Main function of computer clocks – order events
  - If two processes don’t **interact**, there is no need to sync clocks.
  - This observation leads to “causality”

Causality（因果性）

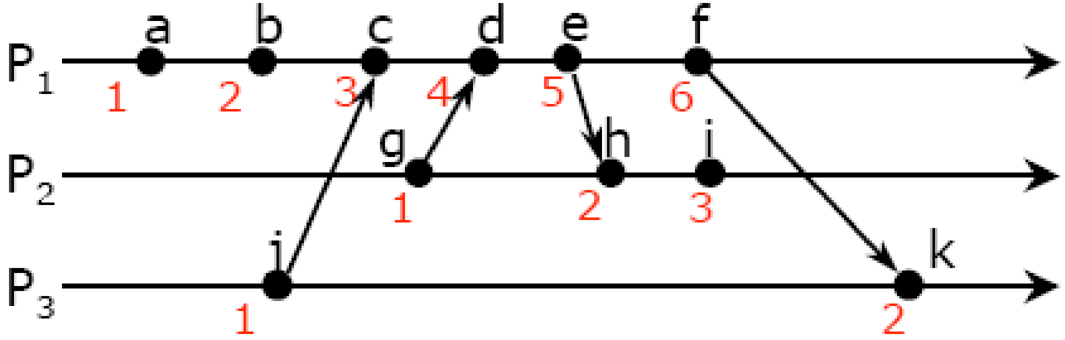
- Order events with happened-before ($\rightarrow$) relation
  - $a \rightarrow b$
    - *a* could have affected the outcome of *b*
  - $a \parallel b$
    - *a* and *b* take place in different processes that don't exchange data
    - Their relative ordering does not matter (they are concurrent)

Definition of *happened-before*

Definition of "$\rightarrow$" relationship:

1. If *a* and *b* take place in the **same process**
   - *a* comes before *b*, then $a \rightarrow b$
2. If *a* and *b* take place in the **different processes**
   - *a* is a "send" and *b* is the corresponding "receive", then $a \rightarrow b$
3. Transitive: if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$

Partial ordering – unordered events are concurrent



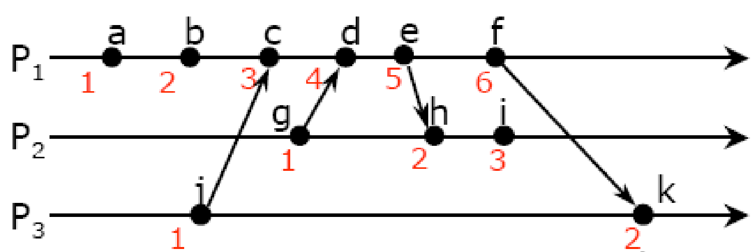
Logical Clocks

- A logical clock is a monotonically increasing software counter. It need not relate to a physical clock.
  - Corrections to a clock must be made by adding, not subtracting

- Rule for assigning "time" values to events
  - if $\boldsymbol{a} \rightarrow \boldsymbol{b}$ then **clock(*a*) < clock(*b*)**

Event counting example

- Three processes: P_0, P_1, P_2, events *a*, *b*, *c*, …
- A local event counter in each process.
- Processes occasionally communicate with each other, where inconsistency occurs, …

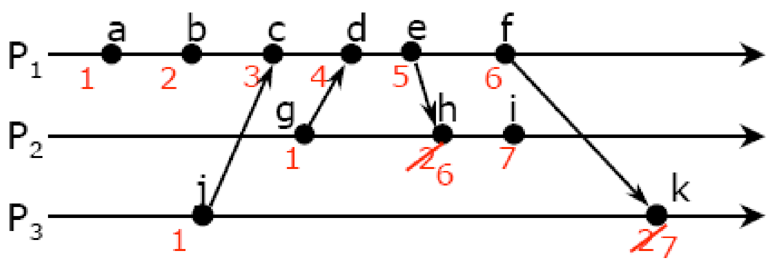


Bad ordering: $e \rightarrow h$, $f \rightarrow k$

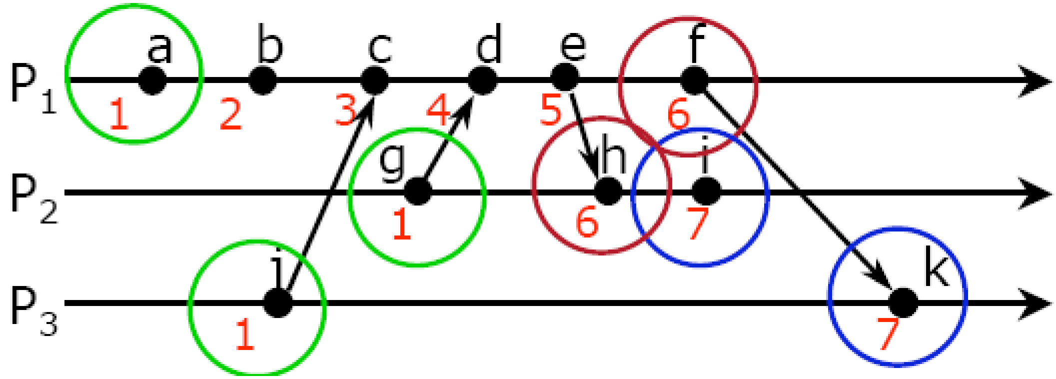
Lamport's algorithm, 1978

Each process P_i has a logical clock L_i. Clock synchronization algorithm:

1. L_i is initialized to 0;
2. Update L_i:
   - LC1: L_i is incremented by 1 for each new event happened in P_i
   - LC2: when P_i sends message m, it attaches $t = L_i$ to m
   - LC3: when P_j receives (m,t) it sets $L_j := max\{L_j, t\}$, and then applies LC1 to increment L_j for event *receive*(m)



Problem: Identical timestamps

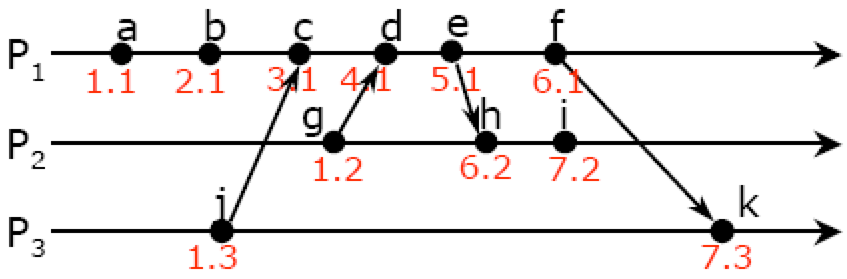


Concurrent events (e.g., a, g) *may* have the same timestamp

Make timestamps unique

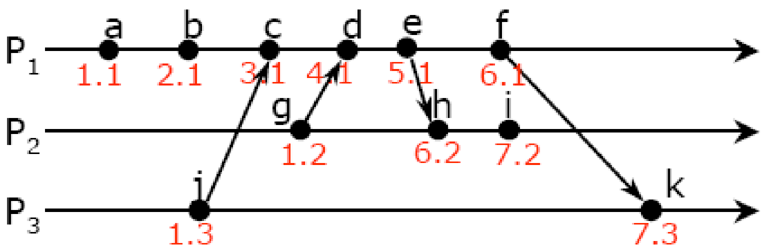
Append the process ID (or system ID) to the clock value after the decimal point:

– e.g. if P_1, P_2 both have $L_1 = L_2 = 40$, make $L_1 = 40.1$, $L_2 = 40.2$



Problem: Detecting causal relations

- If $a \rightarrow b$, then L(a) < L(b), however:
- If L(a) < L(b), we cannot conclude that $a \rightarrow b$
- It is not very useful in distributed systems.



L(g) < L(c), but g || c

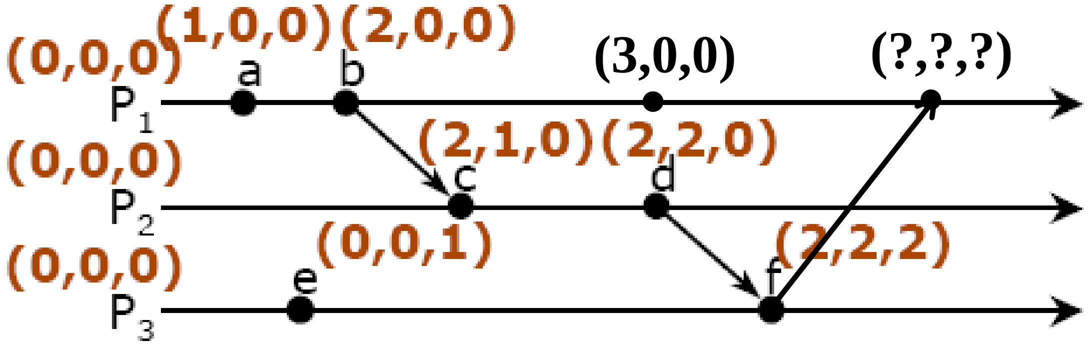
- Solution: use **vector clocks**

Vector of Timestamps

Suppose there are a group of people and each needs to keep track of events happened to others.

Requirement: Given two events, you need to tell whether they are sequential or concurrent.

Solution: you need to have a vector of timestamps, one element for each member.



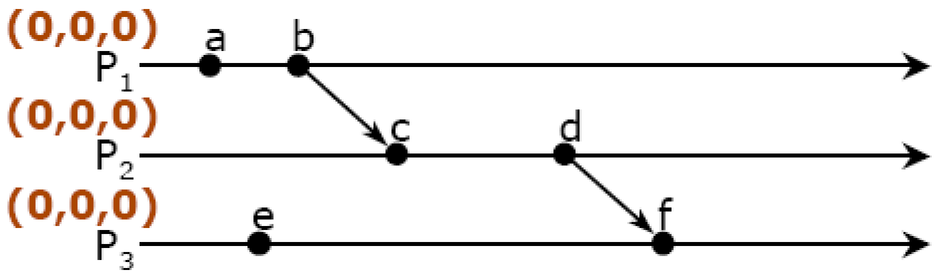
Vector clocks （向量时钟）

Each process P_i keeps a clock V_i which is a vector of N integers

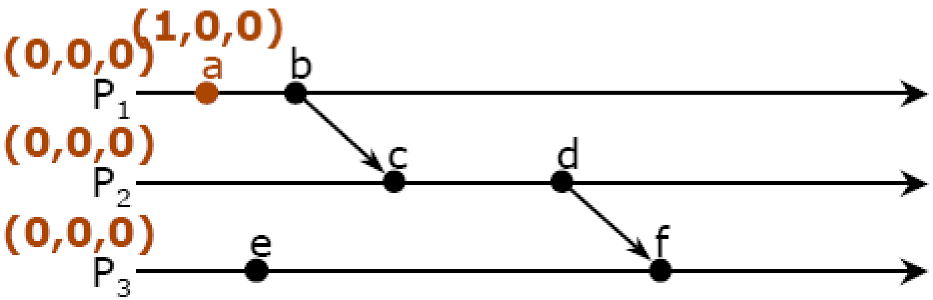
- Initialization: for $1 \leq i \leq N$ and $1 \leq k \leq N$, $V_i[k] := 0$
- Update V_i :
  - VC1: when there is a new event in P_i, it sets $V_i[i] := V_i[i] + 1$
  - VC2: when P_i sends a message m out, it attaches $t = V_i$ to m
  - VC3: when P_j receives (m,t), for $1 \leq k \leq N$, it sets $V_j[k] := \max\{V_j[k], t[k]\}$, then applies VC1 to increment $V_j[j]$ for event *receive*(m,t)

Note: $V_i[j]$ is a timestamp indicating that P_i knows all events that happened in P_j upto this time.

Vector timestamps: example

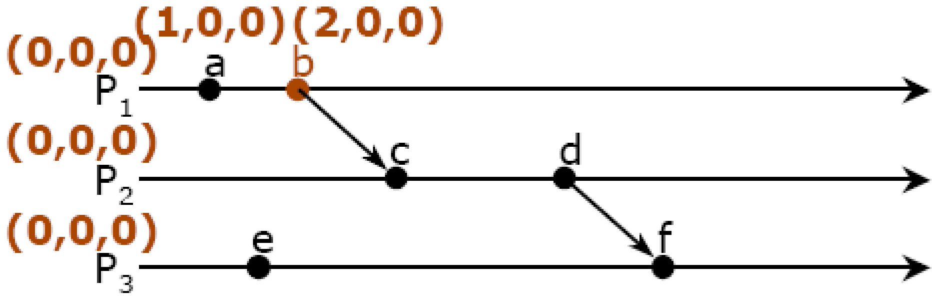


Vector timestamps: example



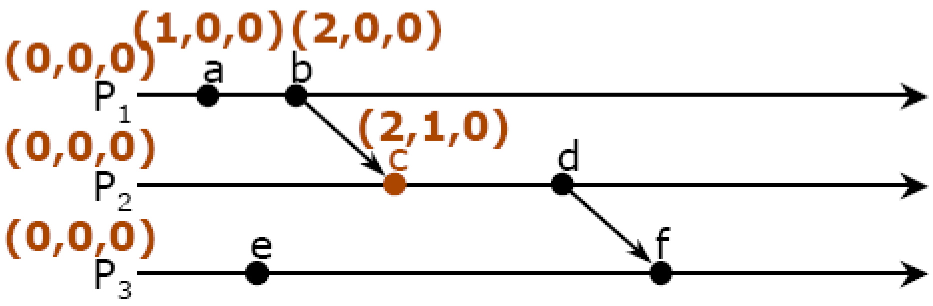
Event	timestamp
a	(1,0,0)

Vector timestamps: example



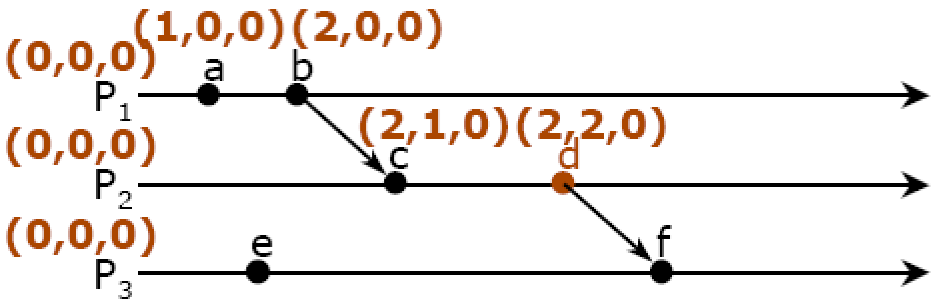
Event	timestamp
a	(1,0,0)
b	(2,0,0)

Vector timestamps: example



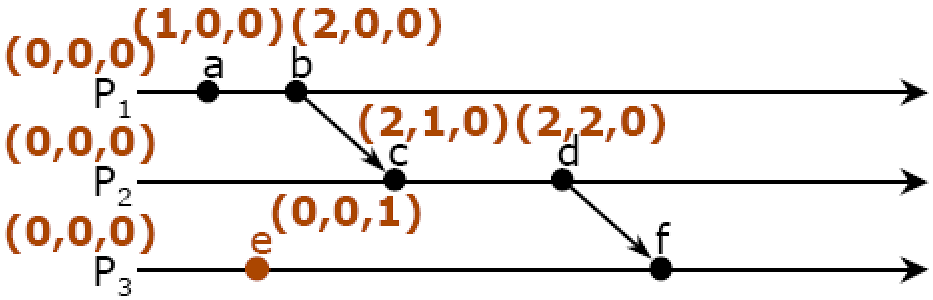
Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)

Vector timestamps: example



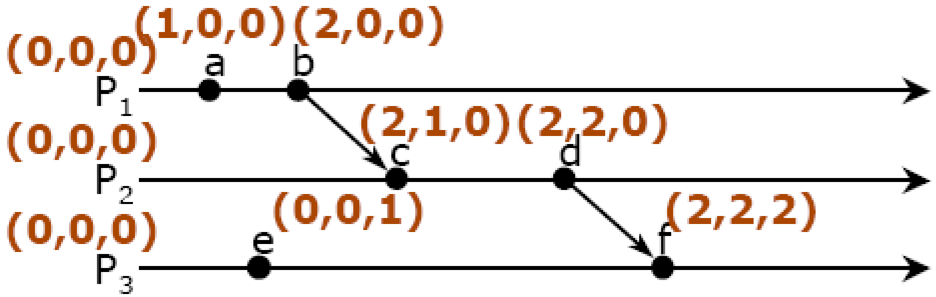
Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)

Vector timestamps: example



Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)

Vector timestamps: example



Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)

Detecting "$\rightarrow$" or "||" events by time vectors

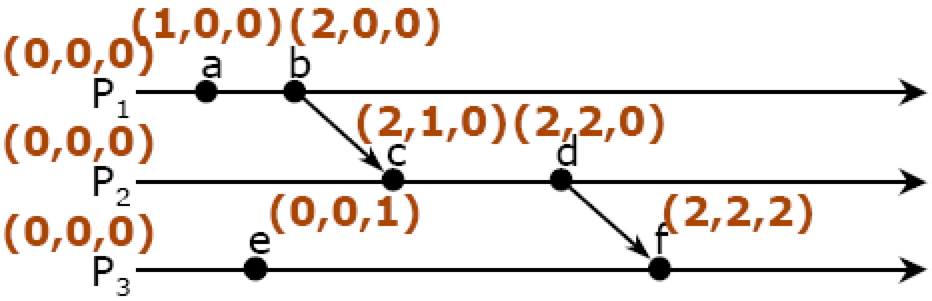
- Define

  $V = V'$ iff $V[i] = V'[i]$ for $i = 1, \ldots, N$

  $V \leq V'$ iff $V[i] \leq V'[i]$ for $i = 1, \ldots, N$

  $V < V'$ iff $V \leq V'$ and $V \neq V'$
- $V(e) \equiv$ timestamp vector of an event e
- For any two events a and b,
  - $a \rightarrow b$ iff $V(a) < V(b)$, $a \neq b$
  - $a \parallel b$ iff **neither** $V(a) \leq V(b)$ **nor** $V(b) \leq V(a)$

Detecting "→" or "||" events: an example



Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)

concurrent events

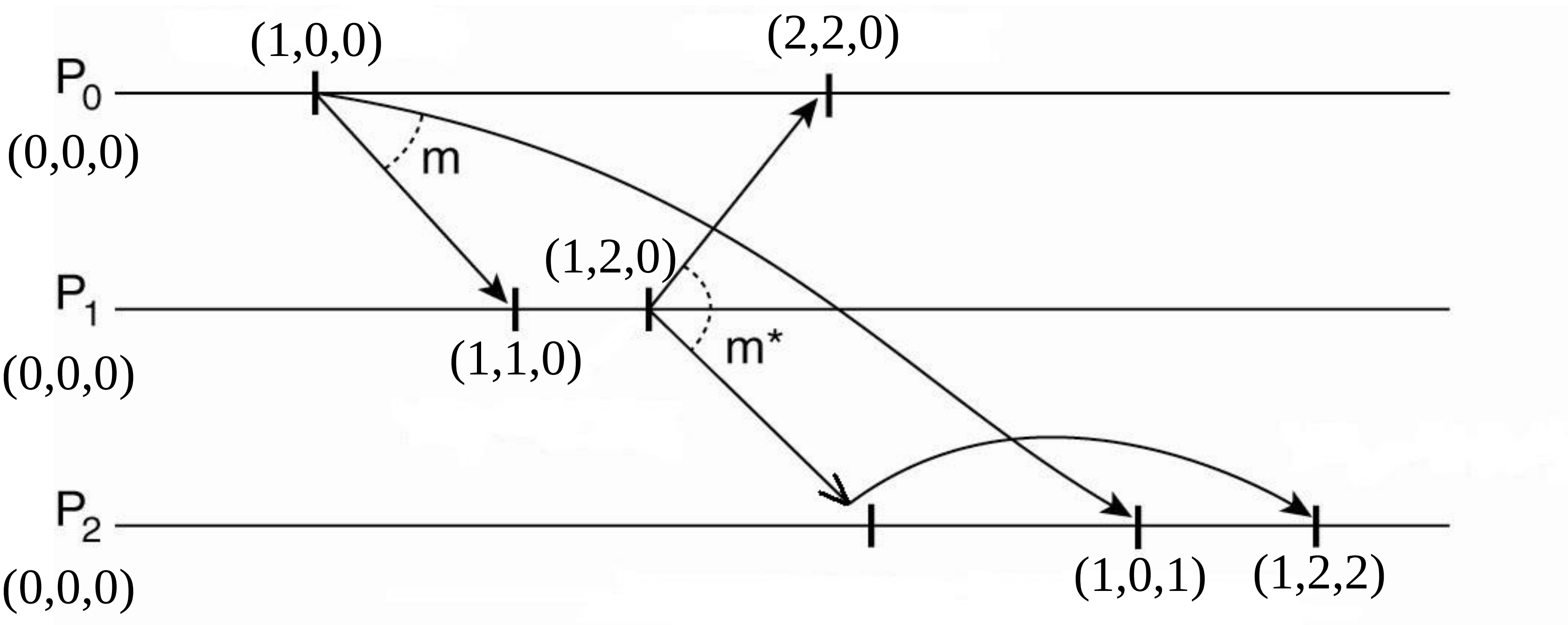
Summary on vector timestamps

- No need to synchronize physical clocks
- Able to order causal events
- Able to identify concurrent events (but cannot order them)

An Application of Timestamp Vectors: causally-ordered multicast

Multicast: a sender sends a message to a group of receivers. Every message must be received by all group members.

Causally ordered multicast: if $m_1 \rightarrow m_2$, m_1 must be received before m_2 by all receivers.



Algorithm of Causally-Ordered Multicast

Each group member keeps a timestamp vector of *n* components (*n* group members), all initialized to 0.

1. When P_i multicasts a message *m*, it increments i-th component of its time vector V_i and attaches V_i to *m*.
2. When P_j with V_j receives (*m*, V_i) from P_i:

   if (**$V_j [k] \geq V_i[k]$ for all k**, $k \neq i$), then

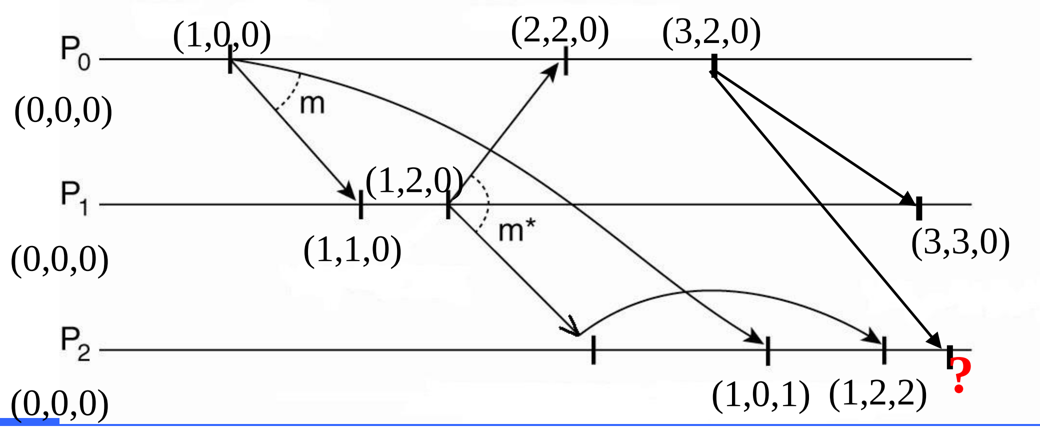
   $V_j [i] := V_i [i]$; // $V_i [i]$ is always greater than $V_j [i]$

   $V_j [j] := V_j [j] + 1$;

   Deliver *m*;

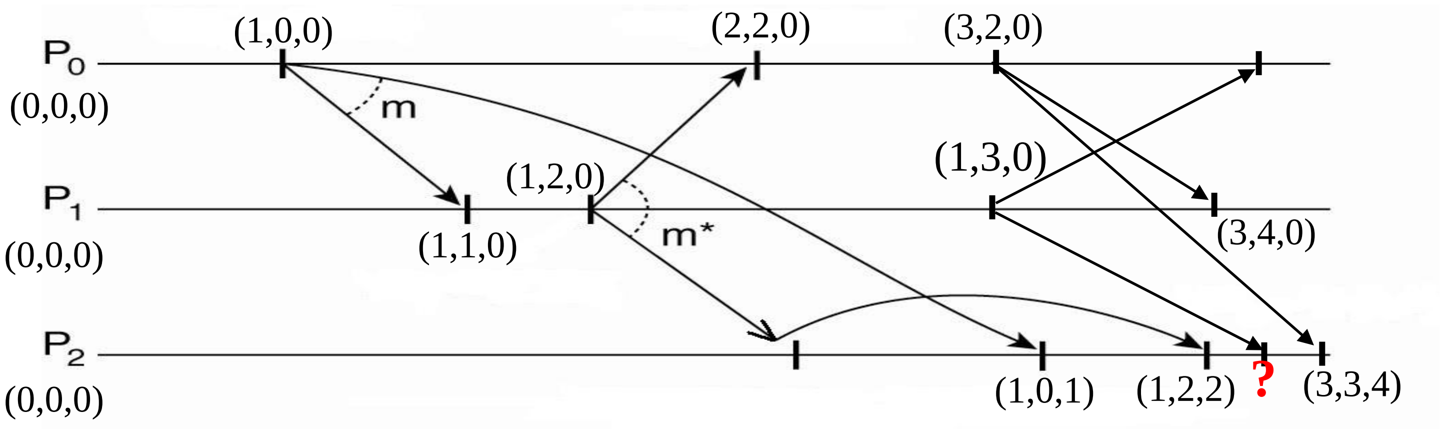
   otherwise

   Delay *m* until "if" is met.



Causal-Order Preserved

- If $m_1 \rightarrow m_2$, m_1 is received by (delivered to) all recipients before m_2.
- If $m_1 \parallel m_2$, m_1 and m_2 can be received in arbitrary order by recipients.
- **Total ordered multicast（全序组播）**: for case of $m_1 \parallel m_2$, m_1 and m_2 must be received in the same order by all recipients (i.e., either all m_1 before m_2, or all m_2 before m_1).



Question

. Suppose a group of process P0 -P5 and timestamp vector is used to represent logical times. P0 sends out a message m_1 with timestamp vector $T_0 = \{5,7,2,3,4,8\}$:

a) what is the event number of P0?

b) what is the event number of the last event that P0 knows about P4?

c) suppose P4 sends a message m_2 with vector $T_4 = \{5,7,3,3,6,8\}$. What is relationship between m_1 and m_2?