Time and Ordering II

ECE 599 / CS 519 – SPRING 2015

Next

- Can we avoid time-synchronization?

- Logical Clocks: Lamport Time Stamps

Ordering Events in a Distributed System

- **To order events across processes, trying to sync clocks is one approach**
- **What if we instead assigned timestamps to events that were not *absolute* time?**

- **As long as these timestamps obey *causality*, that would work**
  - If an event A causally happens before another event B, then timestamp(A) < timestamp(B)
  - Humans use causality all the time
    - E.g., I enter a house only after I unlock it
    - E.g., You receive a letter only after I send it

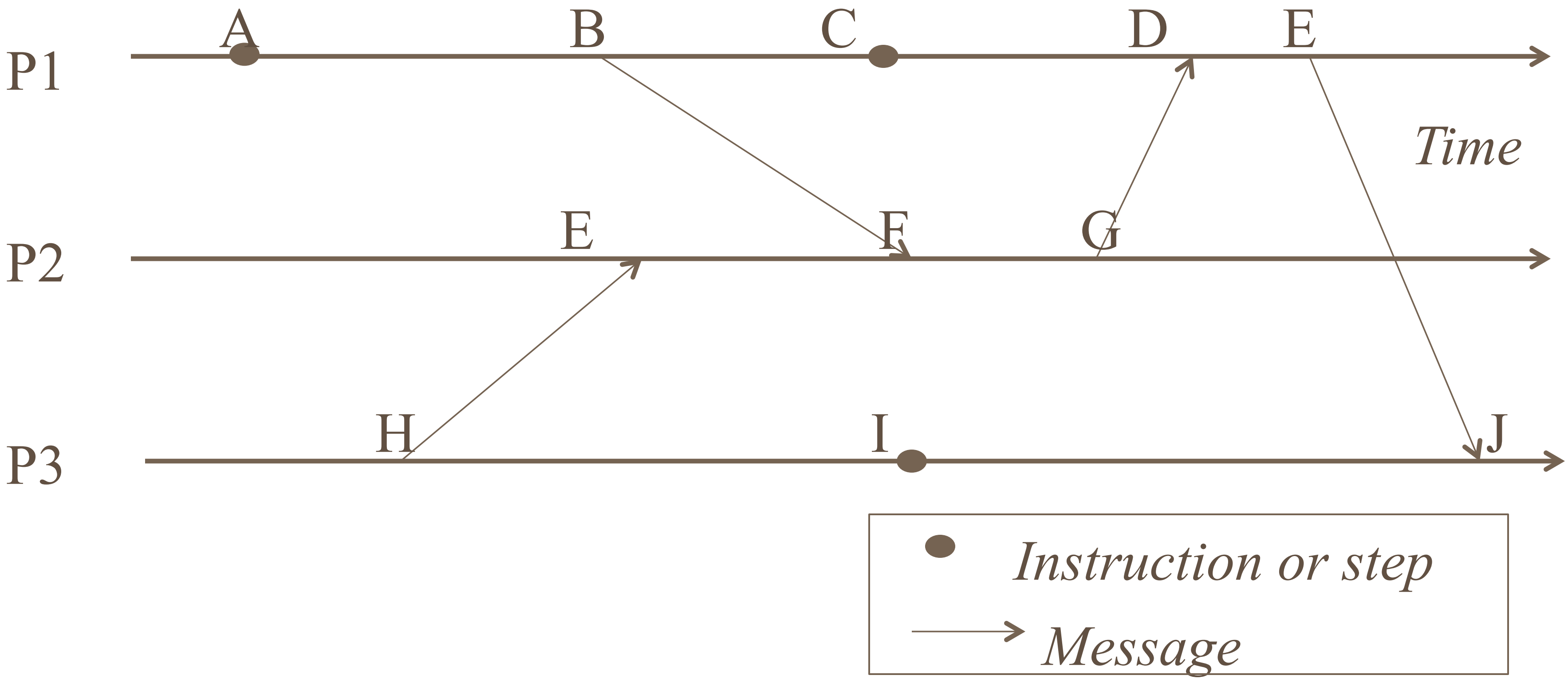
Logical (or Lamport) Ordering

- Proposed by Leslie Lamport in the 1970s
- Used in almost all distributed systems since then
- Almost all cloud computing systems use some form of logical ordering of events

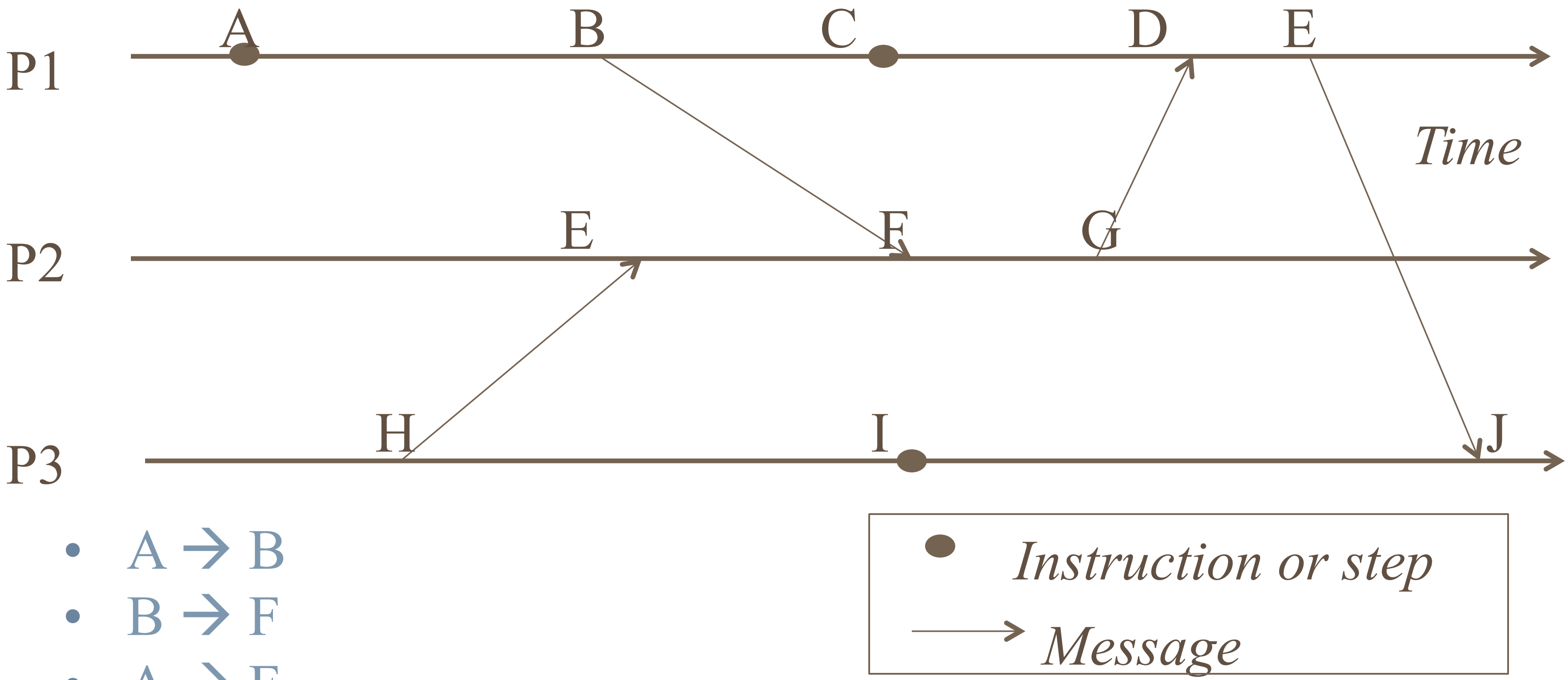
Logical (or Lamport) Ordering(2)

- Define a logical relation *Happens-Before* among pairs of events
- Happens-Before denoted as $\rightarrow$
- Three rules
  1. On the same process: $a \rightarrow b$, if $time(a) < time(b)$ (using the local clock)
  2. If p1 sends m to p2: $send(m) \rightarrow receive(m)$
  3. (Transitivity) If $a \rightarrow b$ *and* $b \rightarrow c$ then $a \rightarrow c$
- Creates a *partial order* among events
  - Not all events related to each other via $\rightarrow$

Example

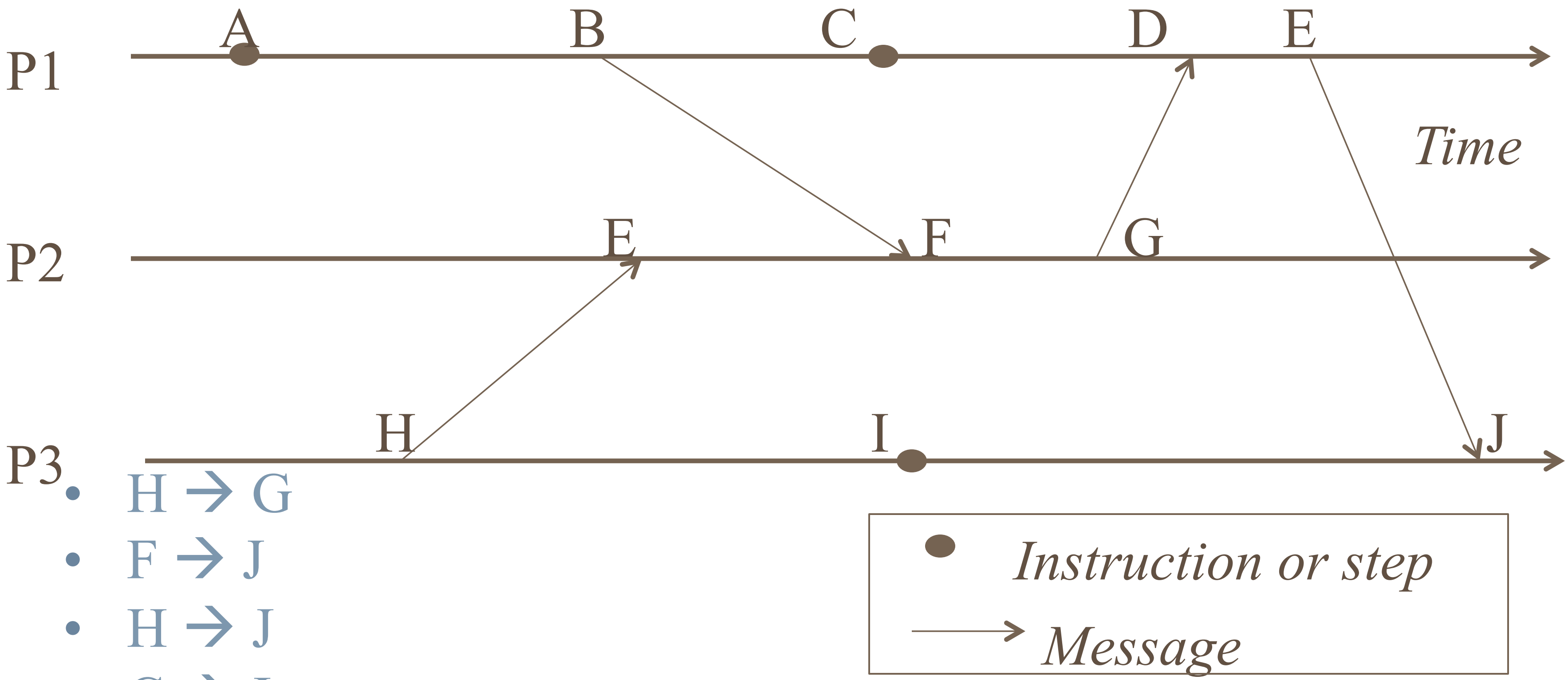


Happens-Before



- A → B
- B → F
- A → F

Happens-Before (2)



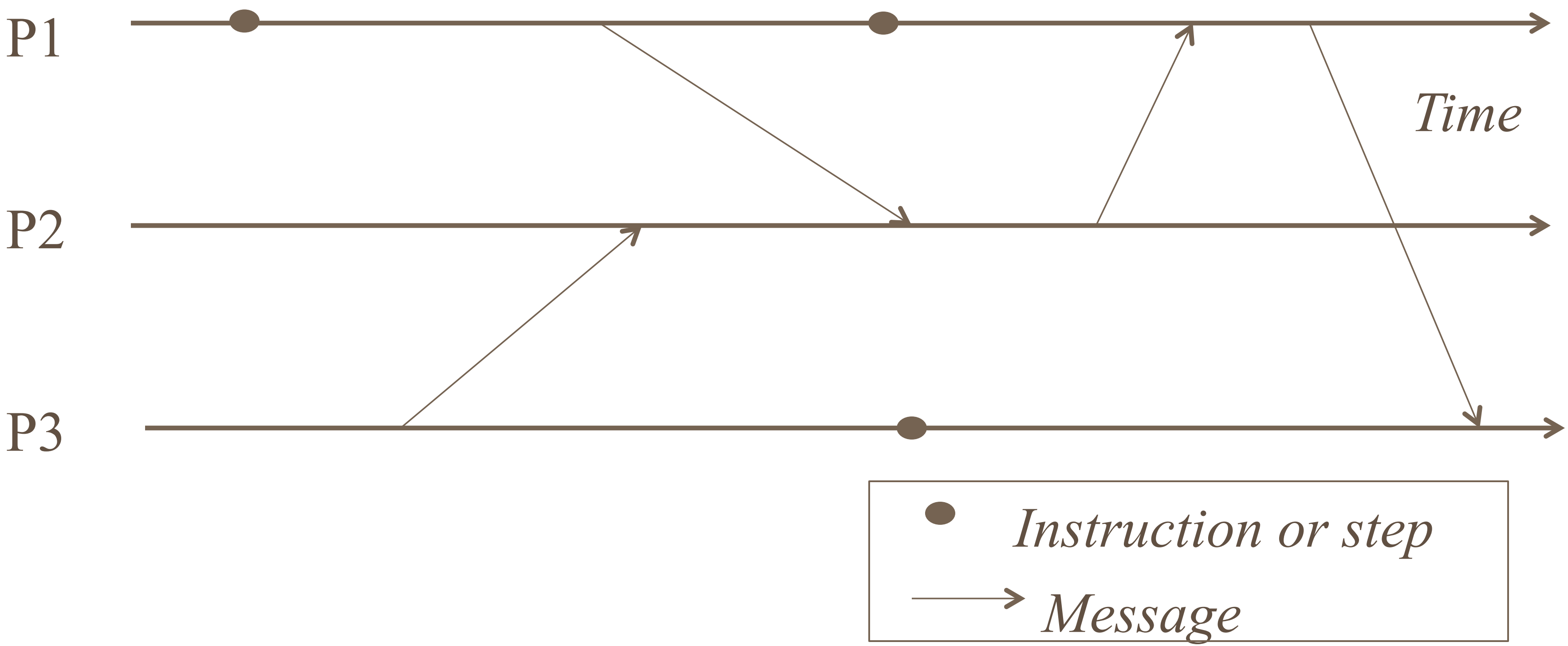
- H → G
- F → J
- H → J
- C → J

In practice: Lamport timestamps

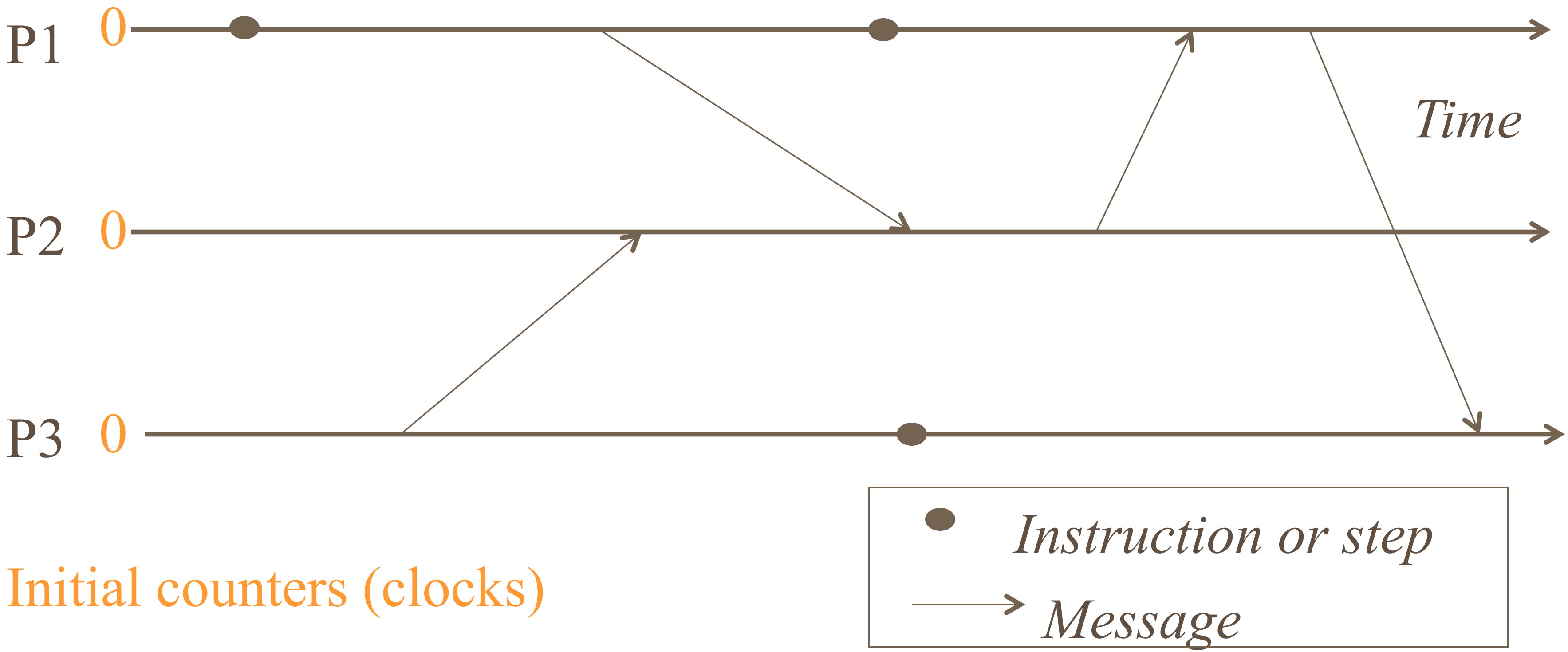
- **Goal: Assign logical (Lamport) timestamp to each event**
- **Timestamps obey causality**
- **Rules**
  - Each process uses a local counter (clock) which is an integer
    - initial value of counter is zero
  - A process increments its counter when a send or an instruction happens at it. The counter is assigned to the event as its timestamp.
  - A send (message) event carries its timestamp
  - For a receive (message) event the counter is updated by

max(local clock, message timestamp) + 1

Example

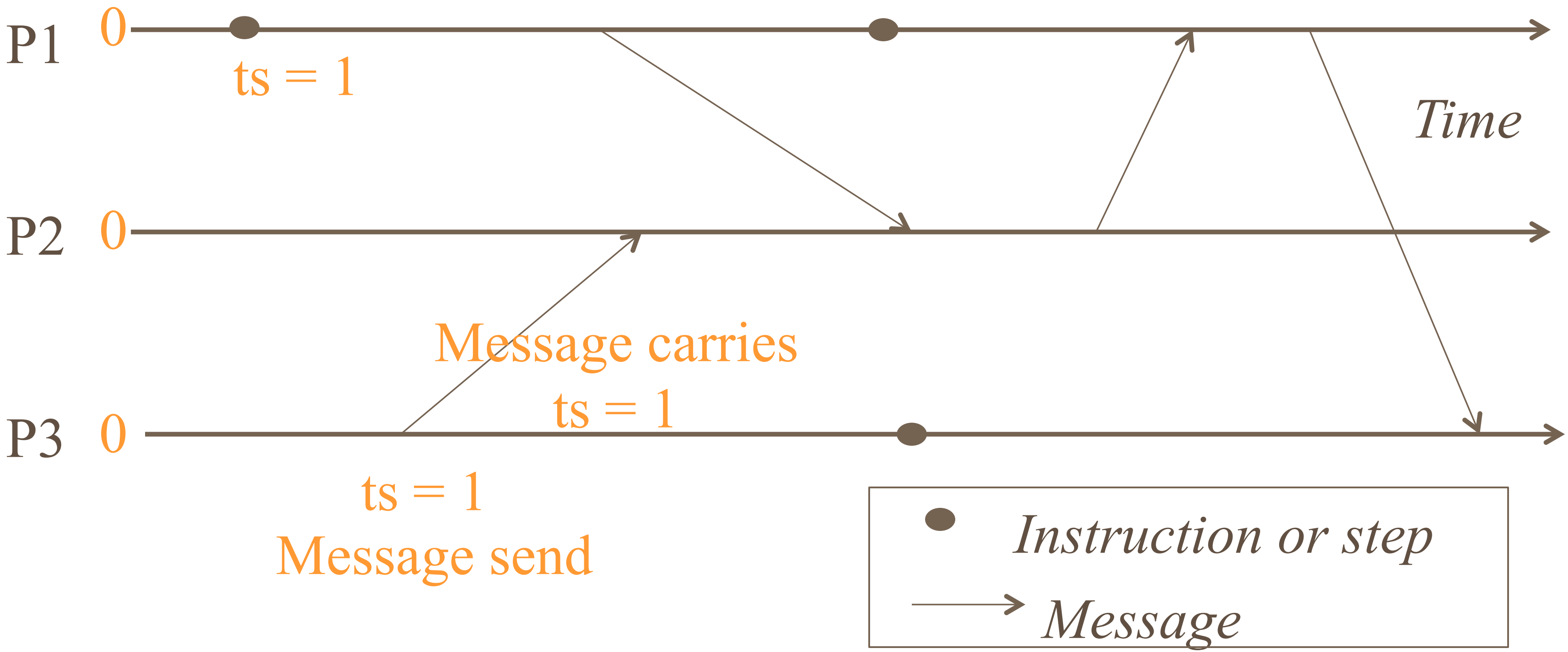


Lamport Timestamps

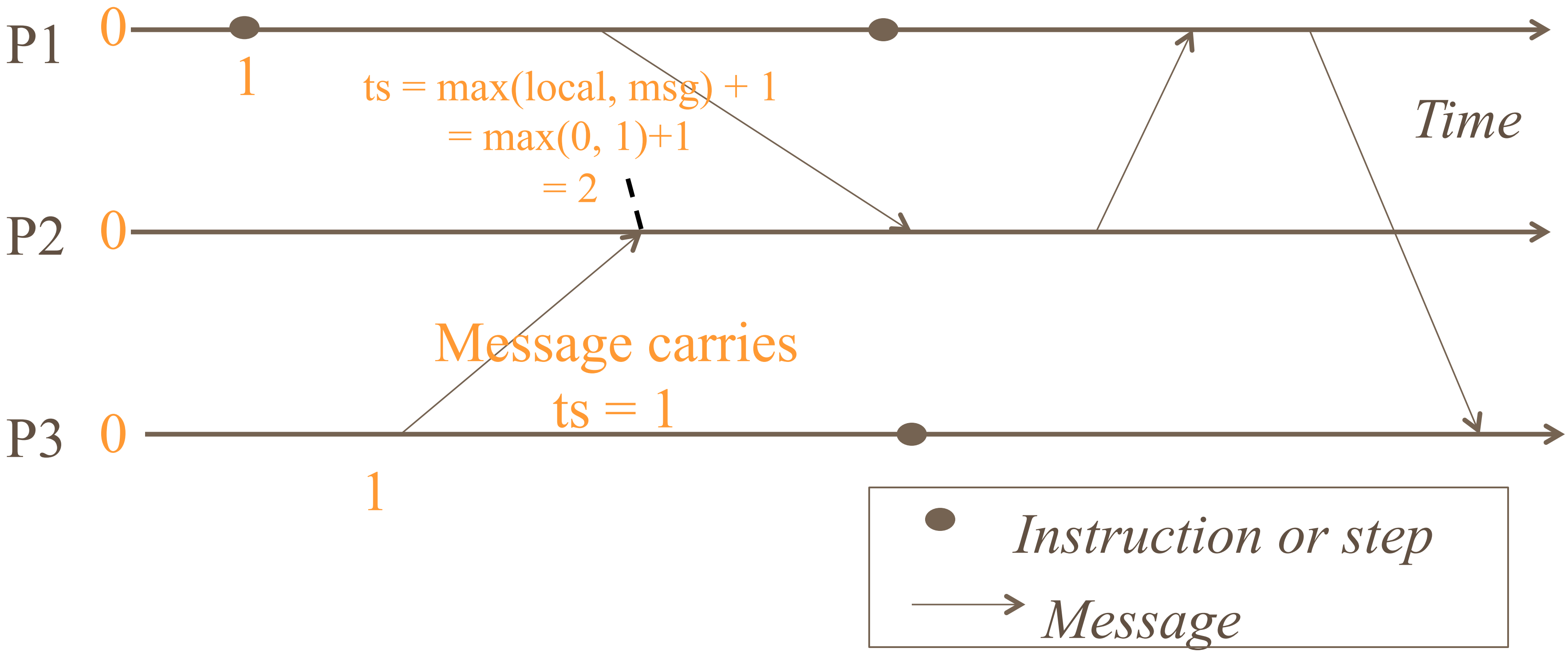


Initial counters (clocks)

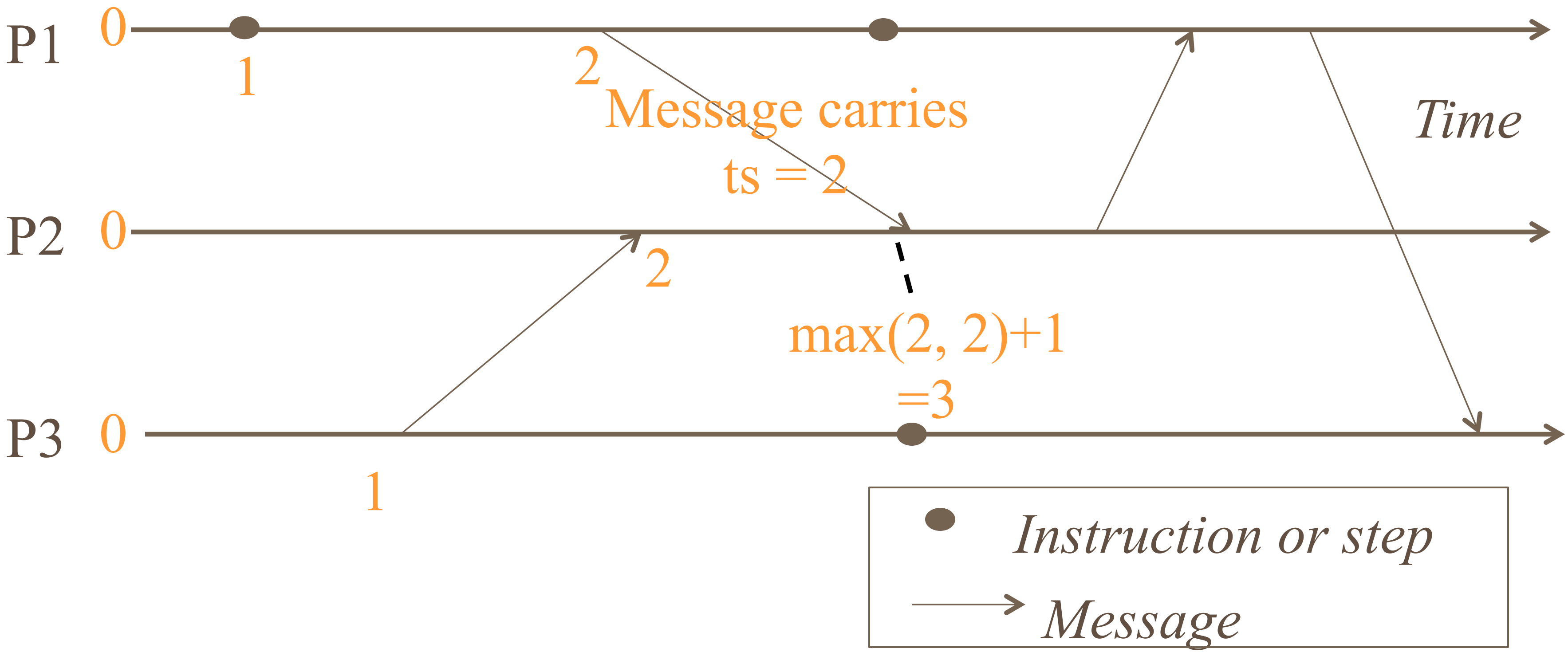
Lamport Timestamps



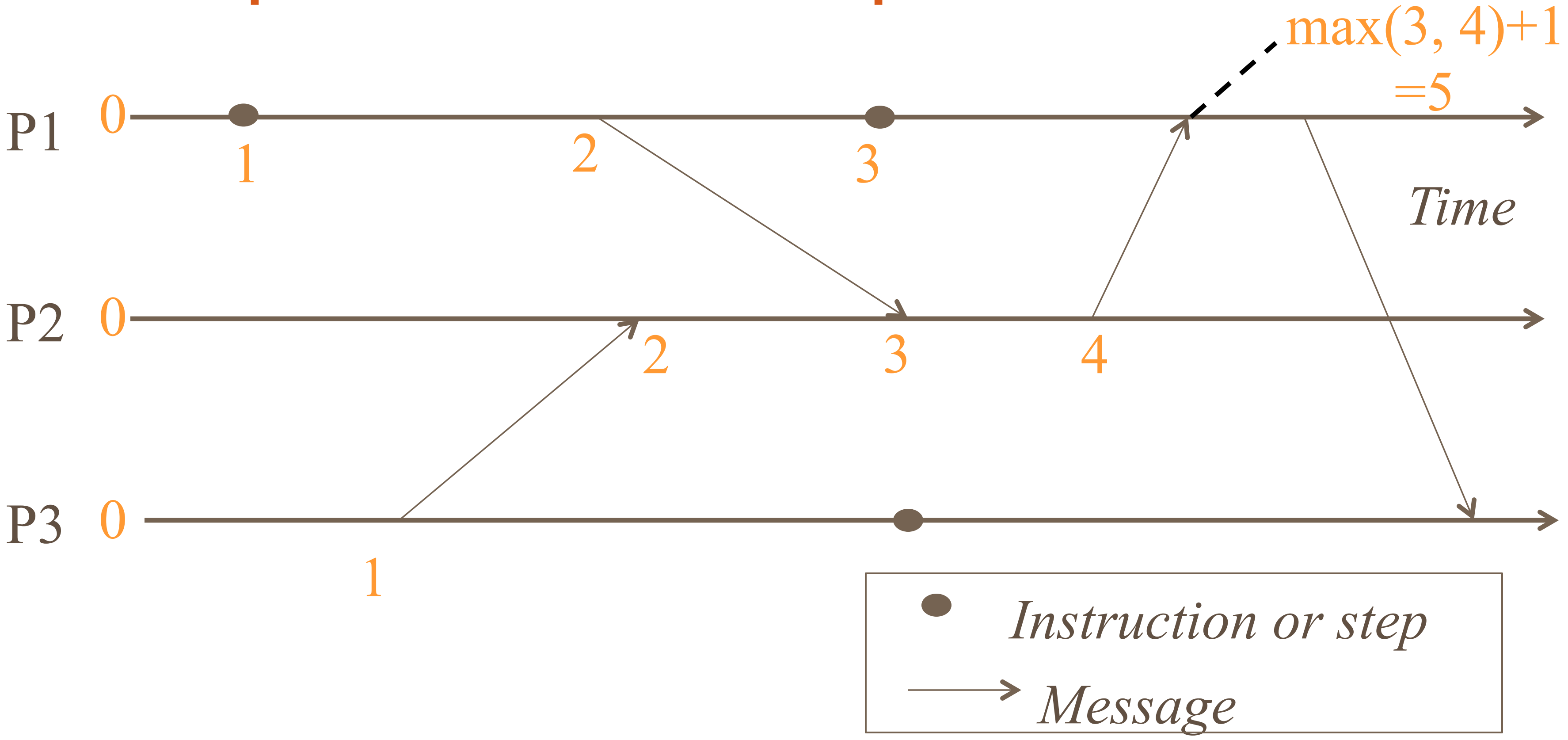
Lamport Timestamps



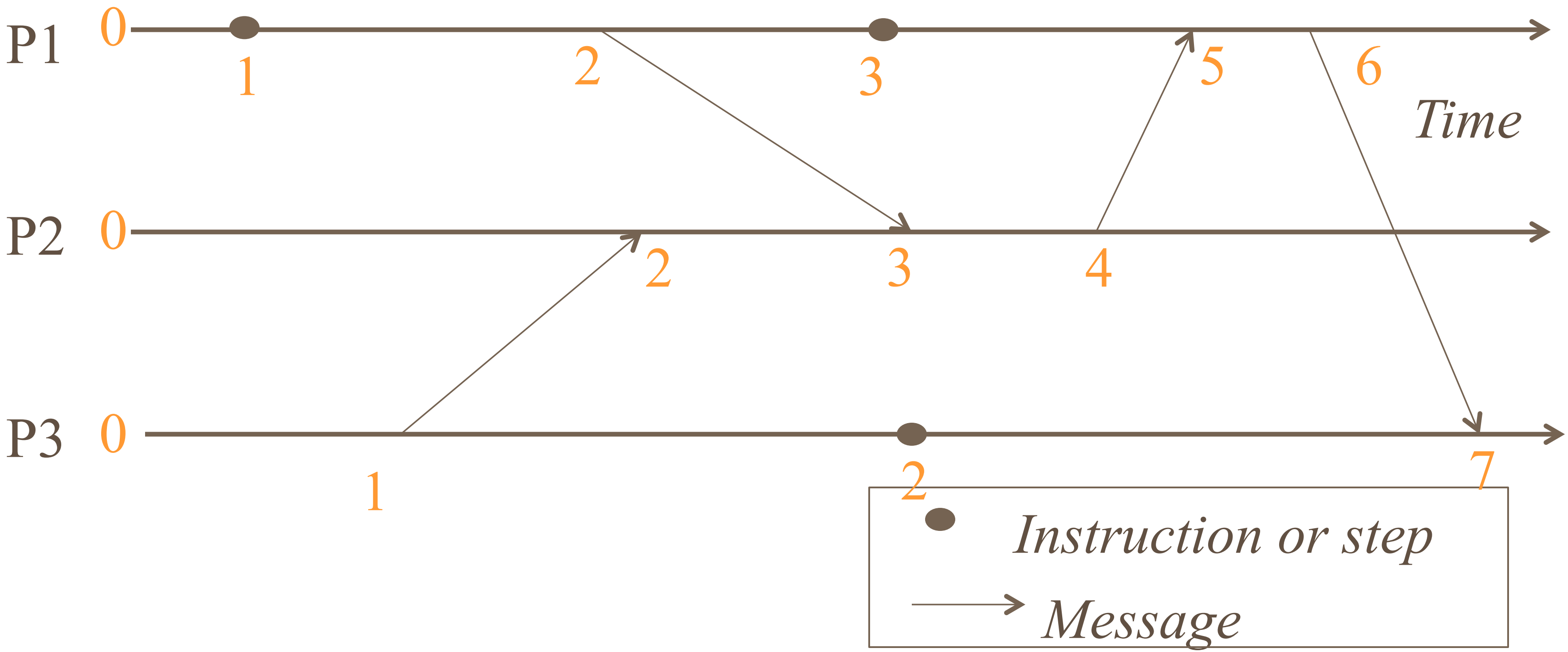
Lamport Timestamps



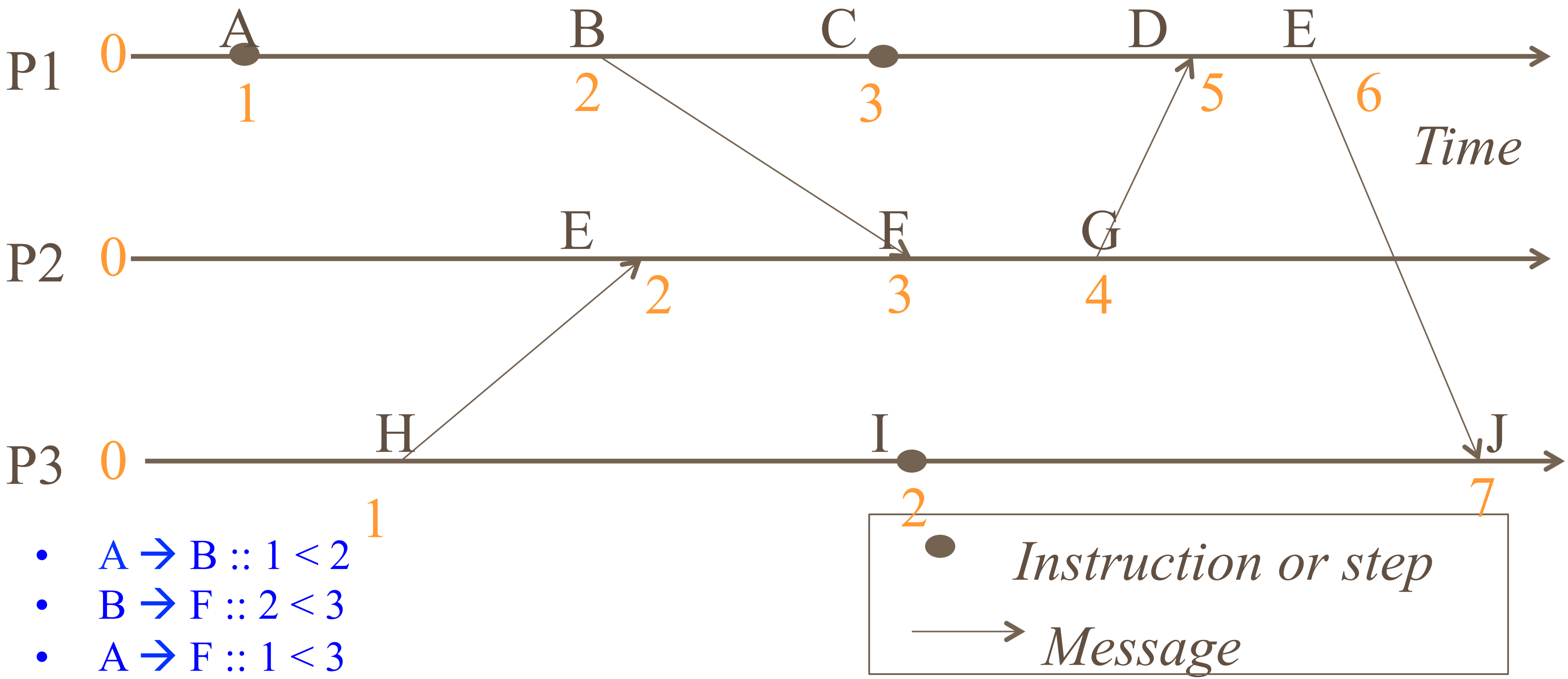
Lamport Timestamps



Lamport Timestamps

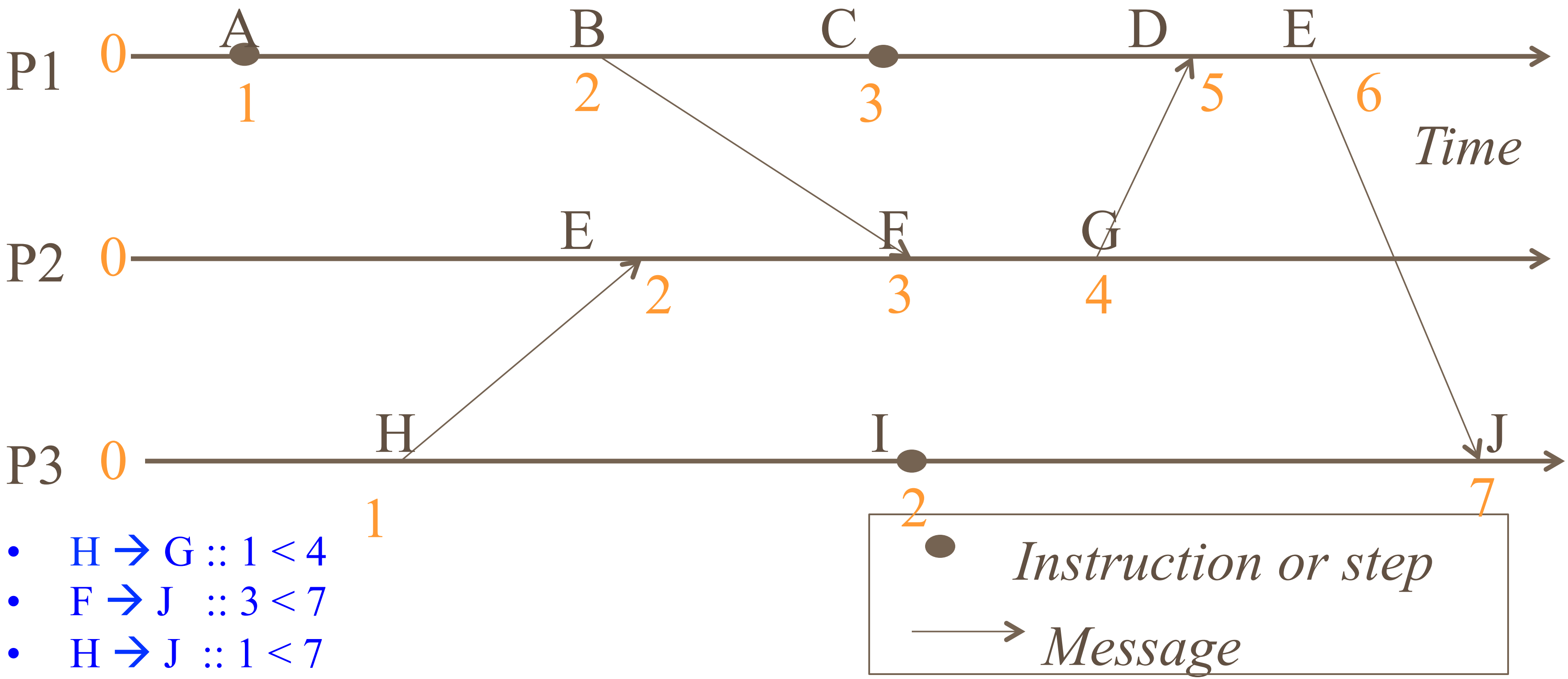


Obeying Causality



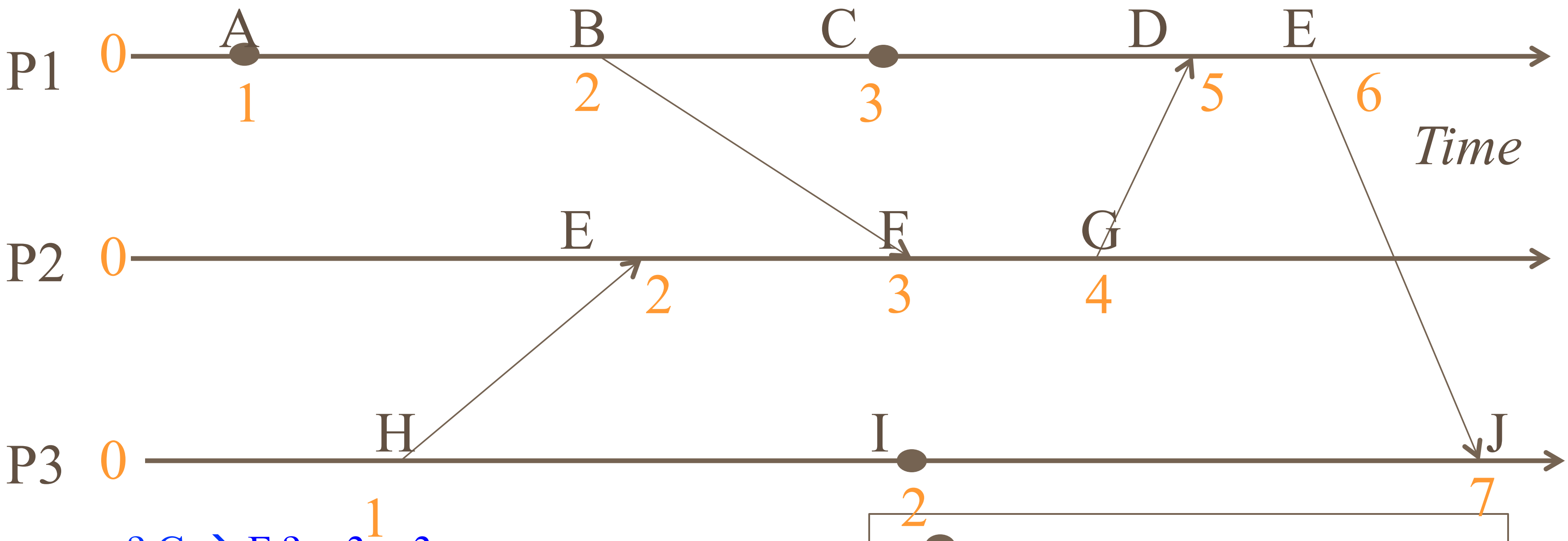
- A → B :: 1 < 2
- B → F :: 2 < 3
- A → F :: 1 < 3

Obeying Causality (2)



- H → G :: 1 < 4
- F → J :: 3 < 7
- H → J :: 1 < 7
- C → J :: 3 < 7

Not always *implying* Causality



- ? C $\rightarrow$ F ? :: 3 = 3
- ? H $\rightarrow$ C ? :: 1 < 3
- (C, F) and (H, C) are pairs of *concurrent* events

Concurrent Events

- A pair of concurrent events doesn't have a causal path from one event to another (either way, in the pair)
- Lamport timestamps not guaranteed to be ordered or unequal for concurrent events
- Ok, since concurrent events are not causality related!
- Remember

  E1 → E2 ⇒ timestamp(E1) < timestamp (E2), BUT

  timestamp(E1) < timestamp (E2) ⇒

  {E1 → E2} OR {E1 and E2 concurrent}

Next

- Can we have causal or logical timestamps from which we can tell if two events are concurrent or causally related?

Next

- Algorithms for Clock Synchronization
- Logical Clocks: Vector Clocks

Vector Timestamps

- Used in key-value stores like Riak
- Each process uses a vector of integer clocks
- Suppose there are N processes in the group 1…N
- Each vector has N elements
- Process *i maintains vector* $\mathbf{V}_i[1\ldots N]$
- *j*th element of vector clock at process *i*, $V_i[j]$, is *i*'s knowledge of latest events at process *j*

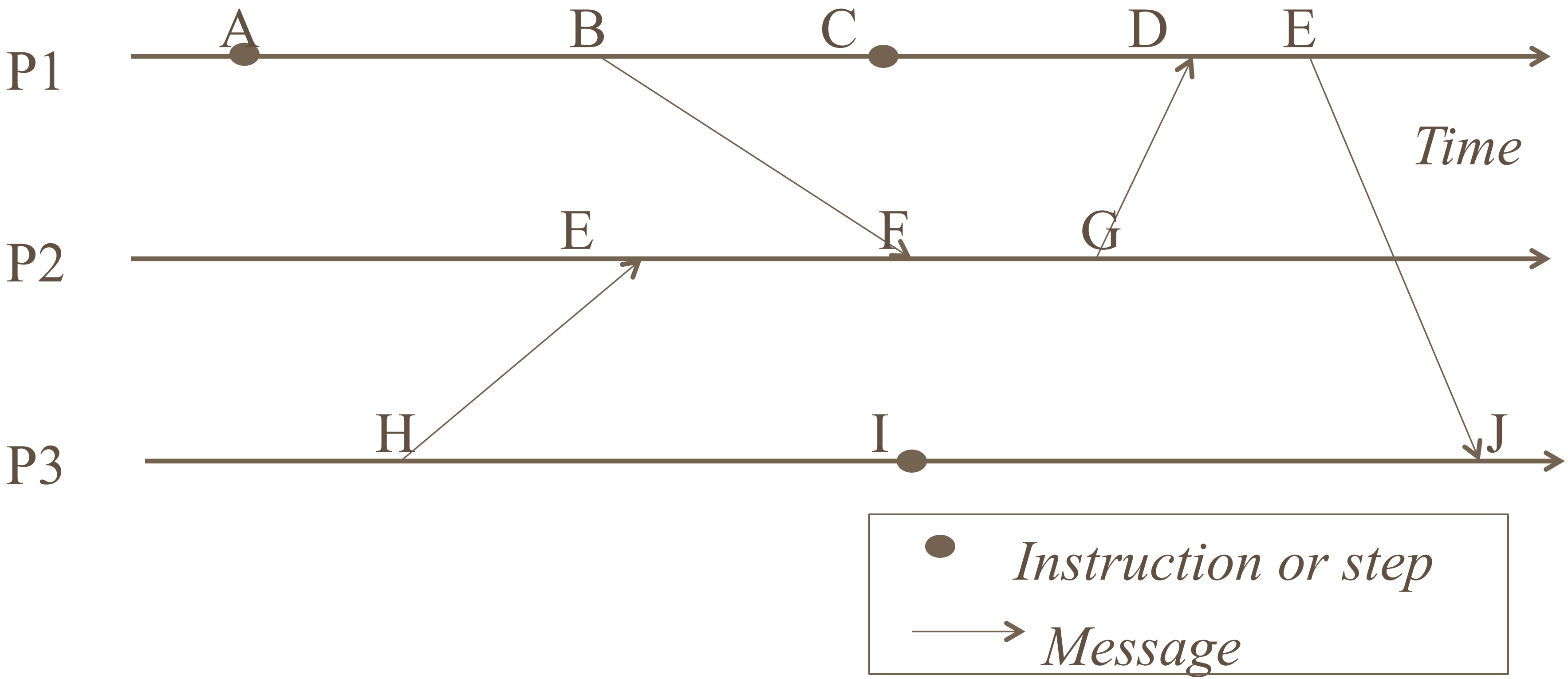
Assigning Vector Timestamps

- Incrementing vector clocks
  1. On an instruction or send event at process *i*, it increments only its *i*th element of its vector clock
  2. Each message carries the send-event's vector timestamp $V_{message}[1 \ldots N]$
  3. On receiving a message at process *i*:

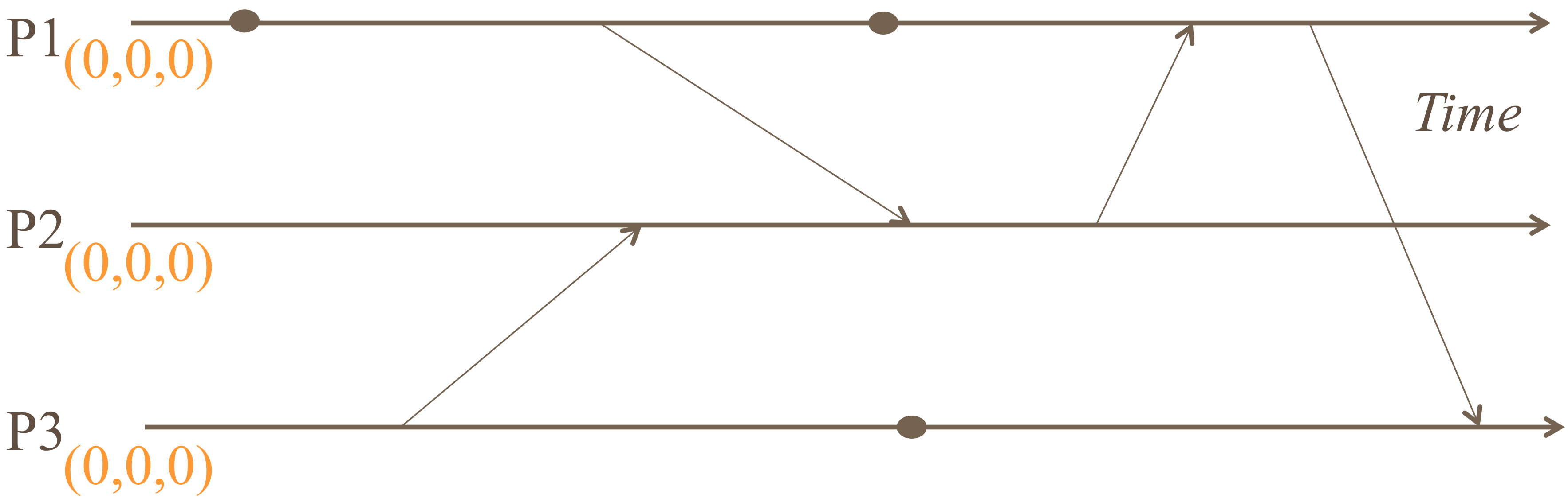
$$V_i[i] = V_i[i] + 1$$

$$V_i[j] = \max(V_{message}[j], V_i[j]) \text{ for } j \neq i$$

Example

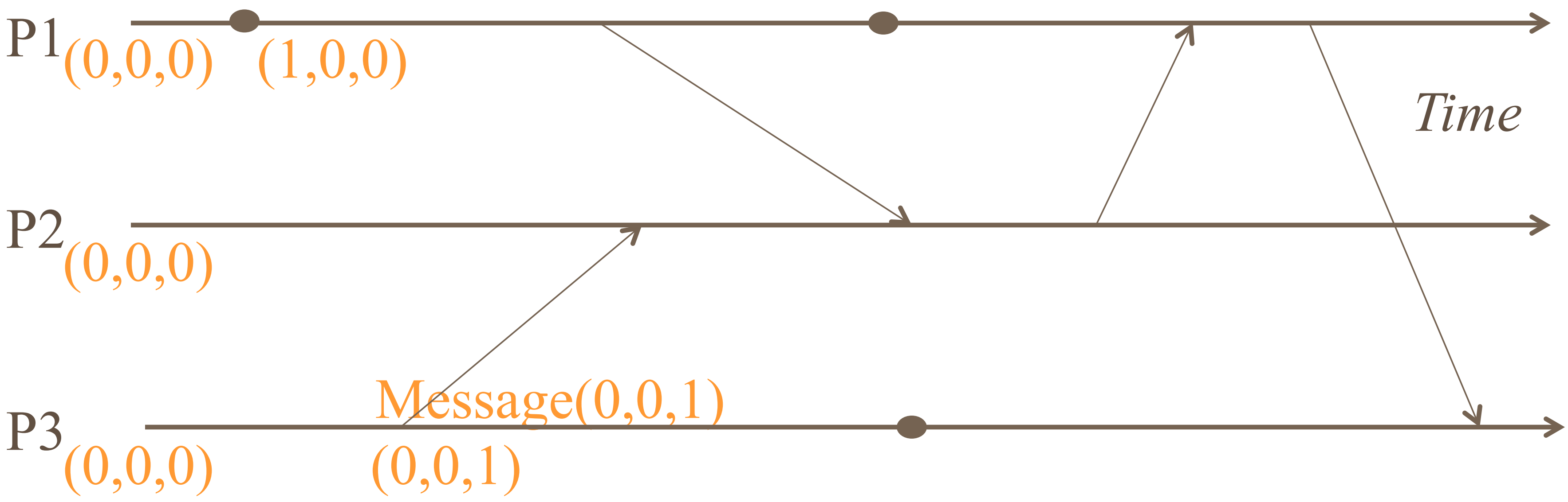


Vector Timestamps

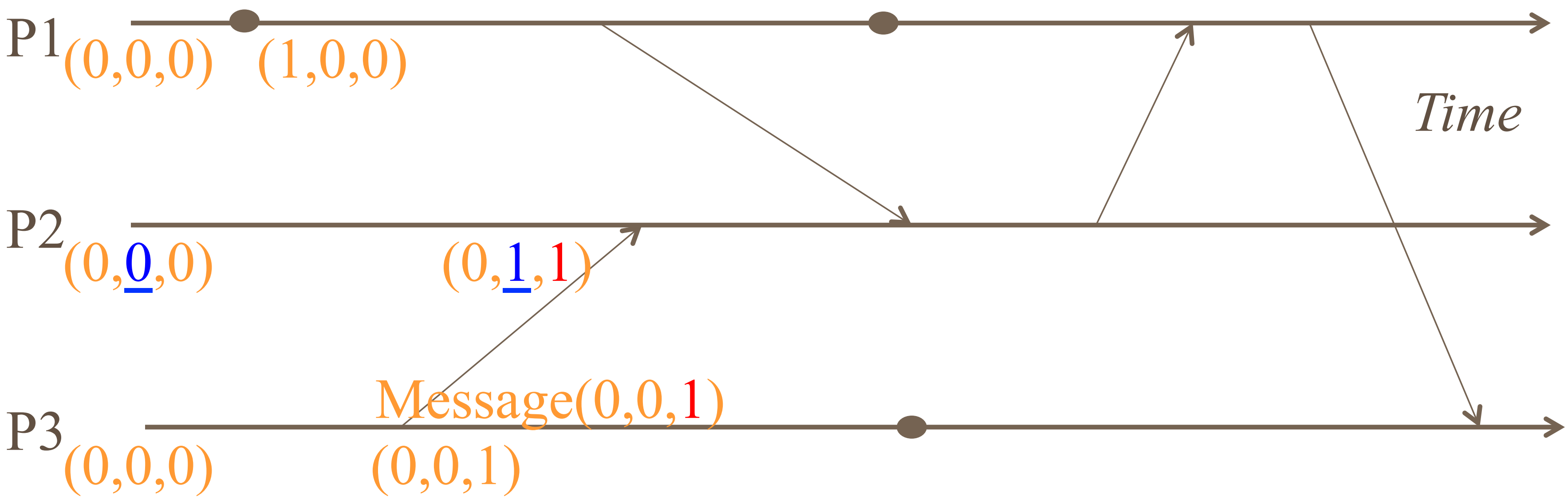


Initial counters (clocks)

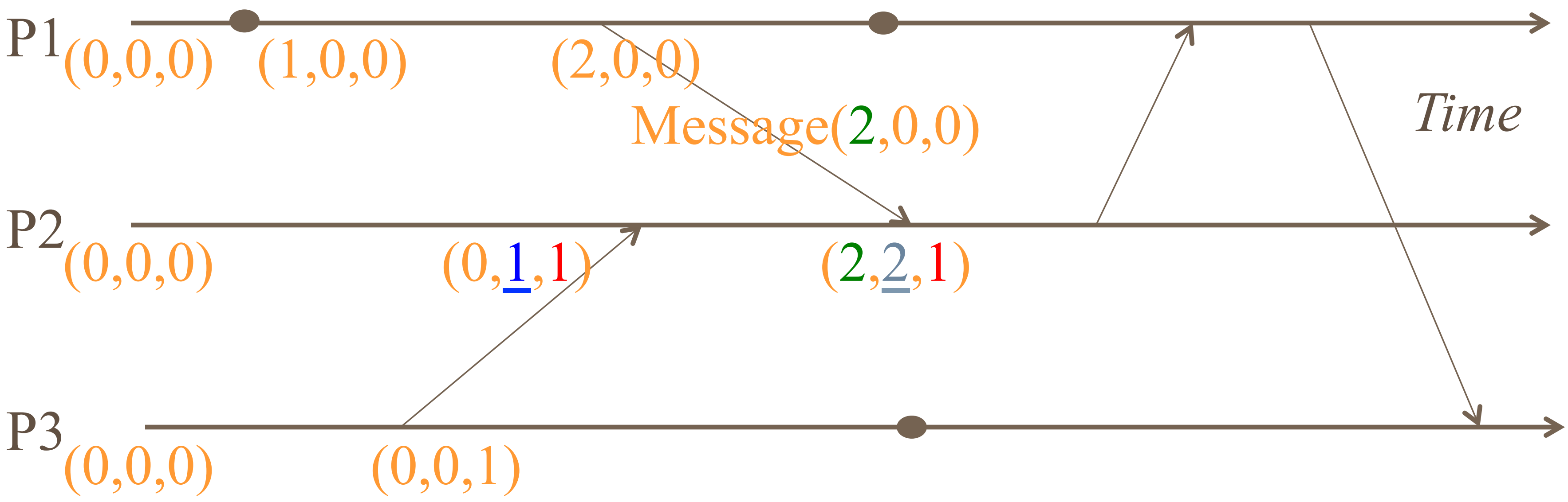
Vector Timestamps



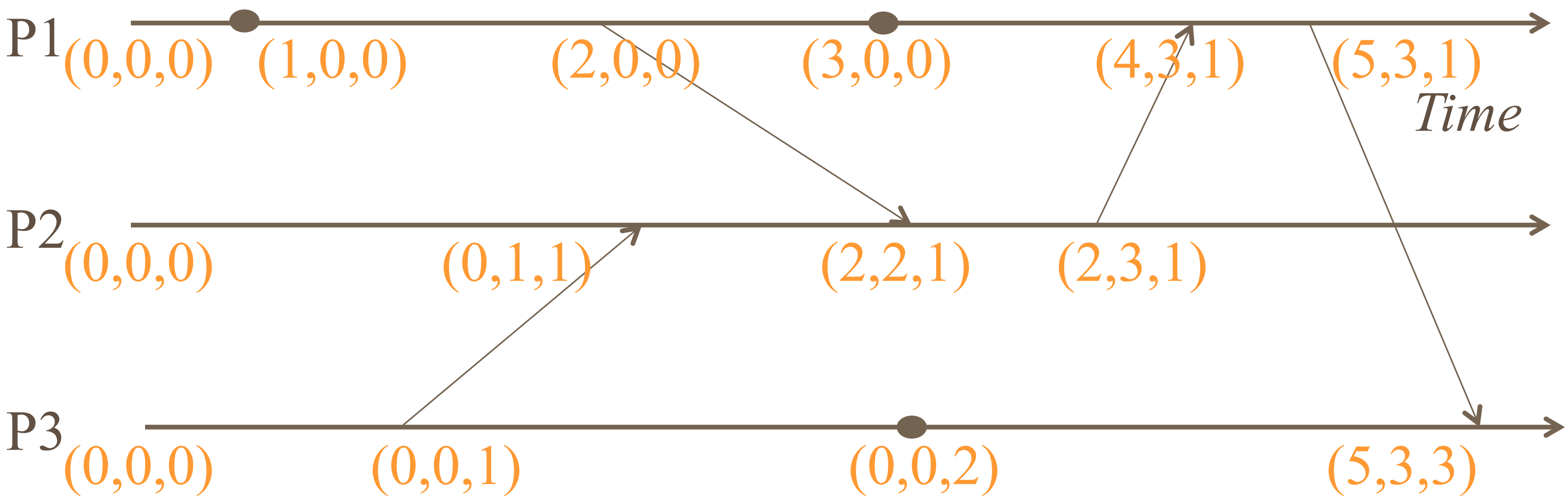
Vector Timestamps



Vector Timestamps



Vector Timestamps



Causally-Related …

$VT_1 = VT_2$,

iff (if and only if)

$VT_1[i] = VT_2[i]$, for all $i = 1, \dots, N$

$VT_1 \leq VT_2$,

iff $VT_1[i] \leq VT_2[i]$, for all $i = 1, \dots, N$

Two events are causally related *iff*

$VT_1 < VT_2$, i.e.,

iff $VT_1 \leq VT_2$ &

there exists j such that

$1 \leq j \leq N$ & $VT_1[j] < VT_2[j]$

… or Not Causally-Related

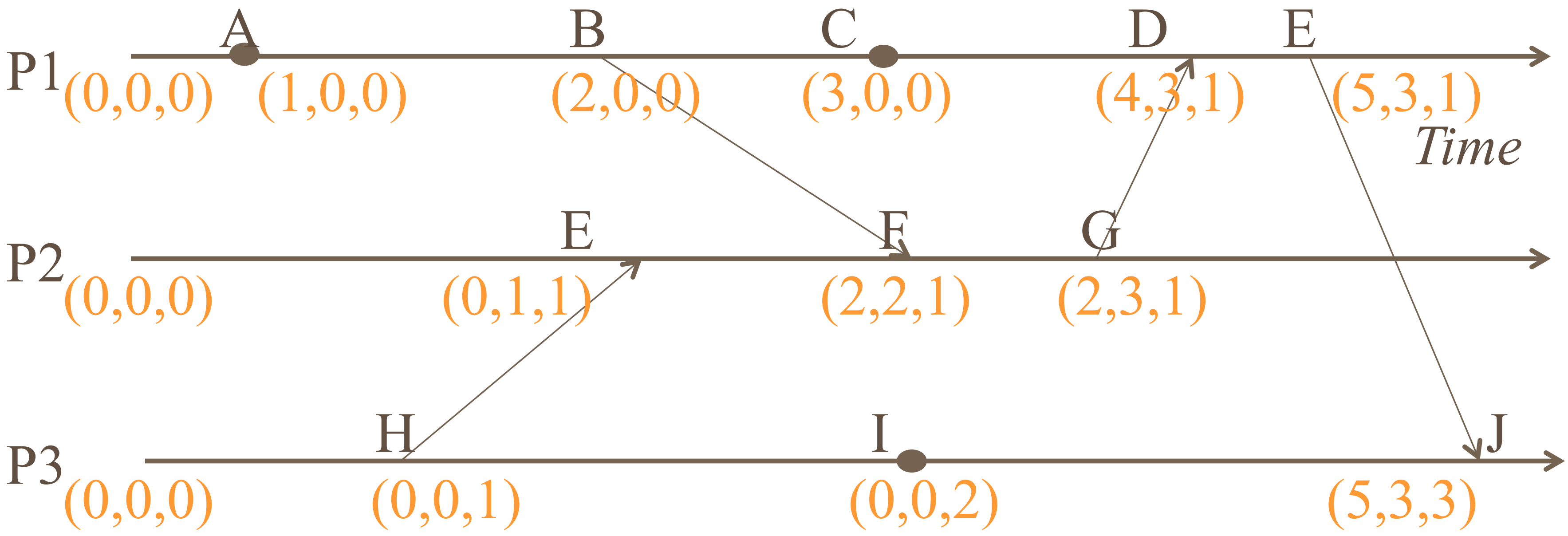
Two events VT_1 and VT_2 are concurrent

iff

$$\text{NOT } (VT_1 \leq VT_2) \text{ AND NOT } (VT_2 \leq VT_1)$$

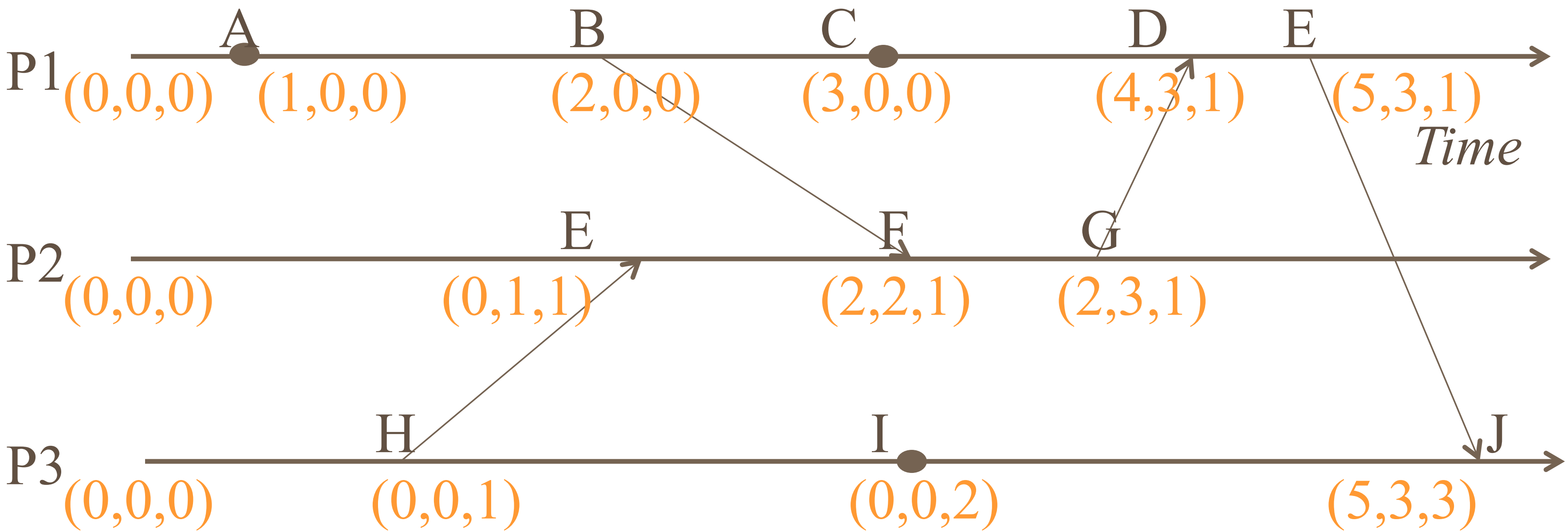
We'll denote this as $VT_2 \,|||\, VT_1$

Obeying Causality



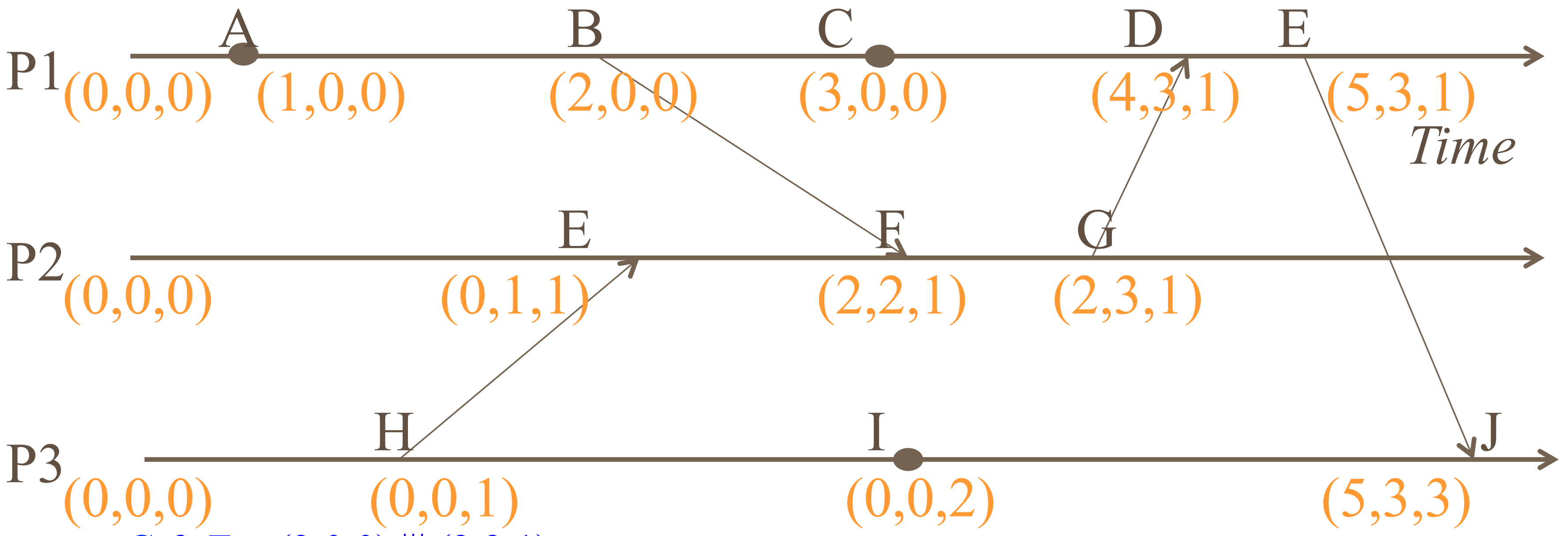
- A → B :: (1,0,0) < (2,0,0)
- B → F :: (2,0,0) < (2,2,1)
- A → F :: (1,0,0) < (2,2,1)

Obeying Causality (2)



- H → G :: (0,0,1) < (2,3,1)
- F → J :: (2,2,1) < (5,3,3)
- H → J :: (0,0,1) < (5,3,3)
- C → J :: (3,0,0) < (5,3,3)

Identifying Concurrent Events



- C & F :: (3,0,0) ||| (2,2,1)
- H & C :: (0,0,1) ||| (3,0,0)
- (C, F) and (H, C) are pairs of *concurrent* events

Logical Timestamps: Summary

- **Lamport timestamps**
  - Integer clocks assigned to events
  - Obeys causality
  - Cannot distinguish concurrent events
- **Vector timestamps**
  - Obey causality
  - By using more space, can also identify concurrent events

Time and Ordering: Summary

- **Clocks are unsynchronized in an asynchronous distributed system**
- **But need to order events, across processes!**
- **Time synchronization**
  - Cristian's algorithm
  - NTP
  - Berkeley algorithm
  - But error a function of round-trip-time
- **Can avoid time sync altogether by instead assigning logical timestamps to events**