Timing and Synchronization

Chapter 14

- During a baseball game, I receive two messages:
- “The hitter hits a homerun!”
- “Timeout! Pitcher A is replaced by Pitcher B”
- Q: Who surrenders the homerun? A or B?

- During a baseball game, I receive two messages:

- “The hitter hits a homerun!”
  - 19:00:03

- “Timeout! Pitcher A is replaced by Pitcher B”
  - 19:00:02

- Q: Who surrenders the homerun? A or B?
- Sol 1: Check the time stamps of the messages
- Only works when clocks are synchronized

Coordinated Universal Time (UTC)

- Astronomical time: Historically, time is defined by the relative positions between the earth and the sun
- UTC: An authoritative atomic clock with very high precision
  - "Leap second" is added occasionally to make it consistent with astronomical time
  - Used to synchronize all satellites
- GPS satellites broadcast time information to land-based devices
  - Precision is within 0.1 - 10ms from UTC
- GPS time is not always available

Some Definitions

- An Asynchronous Distributed System consists of a number of processes.
- Each process has a state (values of variables).
- Each process takes actions to change its state, which may be an instruction or a communication action (send, receive).
- An event is the occurrence of an action.
- Each process has a local clock – events *within* a process can be assigned timestamps, and thus ordered linearly.
- But – in a distributed system, we also need to know the time order of events *across* different processes.

Clocks

- Hardware clock: each computer has a device counting the oscillations of a crystal
- Denote $H_i(t)$ as the hardware clock of process i
- The operating system translates the hardware clock into a software clock:
- $C_i(t) = \alpha H_i(t) + \beta$
- Ideally, we want $C_i(t) = t$
- To do clock synchronization, OS changes the values of α and β

Clock Skew and Clock Drift

- **Each process (running at some end host) has its own clock.**
- **When comparing two clocks at two processes**:
  - Clock Skew = Relative Difference in clock *values* of two processes
    - Like distance between two vehicles on a road
    - The error in β
  - Clock Drift = Relative Difference in clock *frequencies (rates)* of two processes
    - Like difference in speeds of two vehicles on the road
    - The error in α
- **A non-zero clock skew implies clocks are not synchronized.**
- **A non-zero clock drift causes skew to increase (eventually).**

- **The skew of a typically computer is about $\mathbf{10^{-6}}$sec/sec**
- **About 3ms error per hour**

Types of Synchronization

- Consider a group of processes
- External Synchronization
  - Each process i's clock is within a bound D of a well-known clock $S(t)$ external to the group
  - $|C_i(t) - S(t)| < D$ at all times
  - External clock may be connected to UTC (Universal Coordinated Time) or an atomic clock
- Internal Synchronization
  - Every pair of processes in group have clocks within bound D
  - $|C_i(t) - C_j(t)| < D$ at all times and for all processes i, j
- External Synchronization with D => Internal Synchronization with 2*D
  - Why?
- Internal Synchronization does not imply External Synchronization

Correctness of Hardware Clocks

- Require that the drift of a hardware clock cannot exceed some threshold, ρ
- In other words, given $t' > \mathrm{t}$
- $(1-\rho)(t'-t) \leq H(t') - H(t) \leq (1+\rho)(t'-t)$

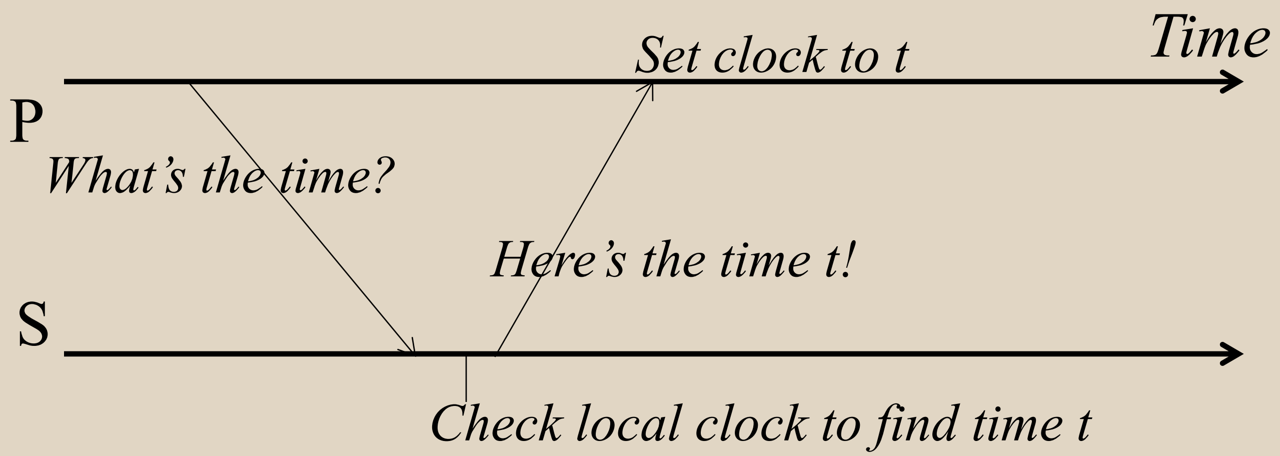
Monotonicity of Software Clocks

- For causality, it is sometimes required that $C_i(t)$ needs to be non-decreasing
  - The timestamp of “result” is always larger than the timestamp of “cause”
- During synchronization, process i obtains UTC time $S(t)$
- If $S(t) > C_i(t)$, set $C_i(t) = S(t)$
- If $S(t) < C_i(t)$, we cannot set $C_i(t) = S(t)$
- Recall $C_i(t) = \alpha H_i(t) + \beta$
- Reduce α for some time

CRISTIAN'S ALGORITHM

Basics

- External time synchronization
- All processes P synchronize with a time server S

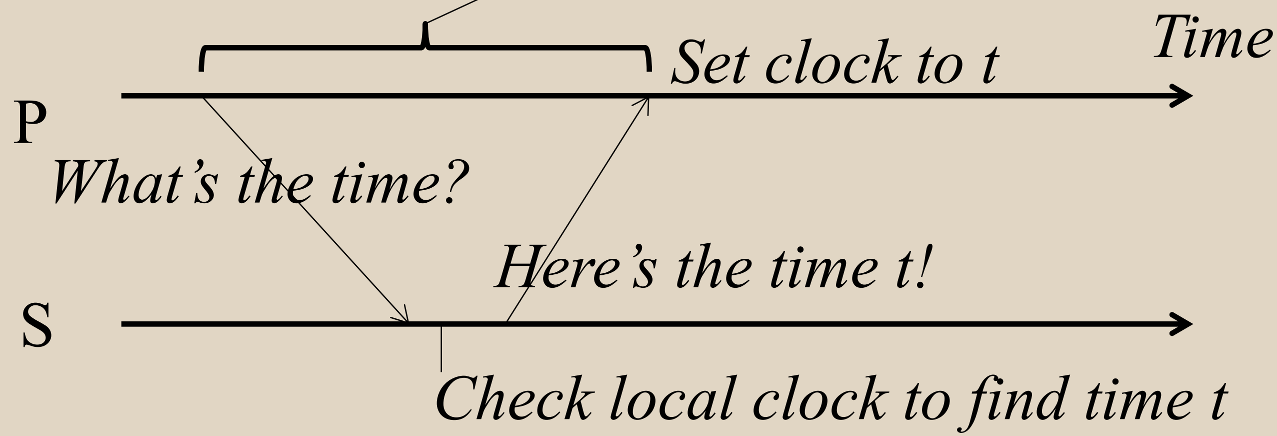


What’s Wrong

- By the time response message is received at P, time has moved on
- P’s time set to *t* is inaccurate!
- Inaccuracy a function of message latencies
- Since latencies unbounded in an asynchronous system, the inaccuracy cannot be bounded

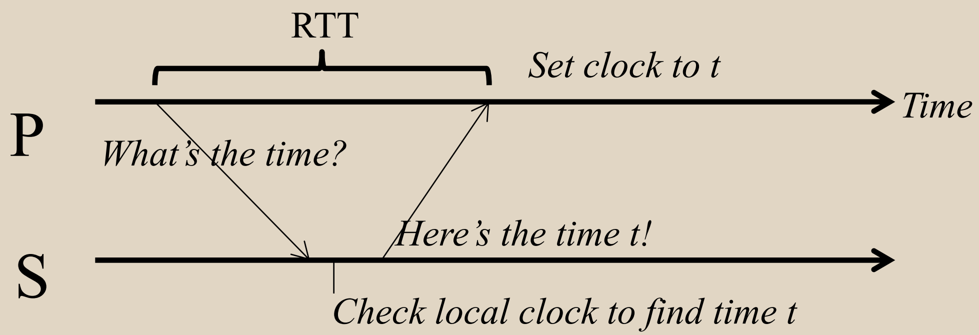
Cristian's Algorithm

- P measures the round-trip-time RTT of message exchange



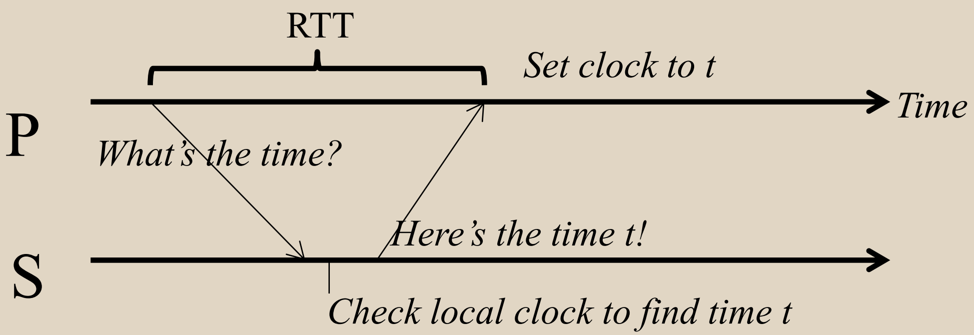
Cristian's Algorithm (2)

- **P measures the round-trip-time RTT of message exchange**
- **Suppose we know the minimum P → S latency min1**
- **And the minimum S → P latency min2**
  - min1 and min2 depend on Operating system overhead to buffer messages, TCP time to queue messages, etc.



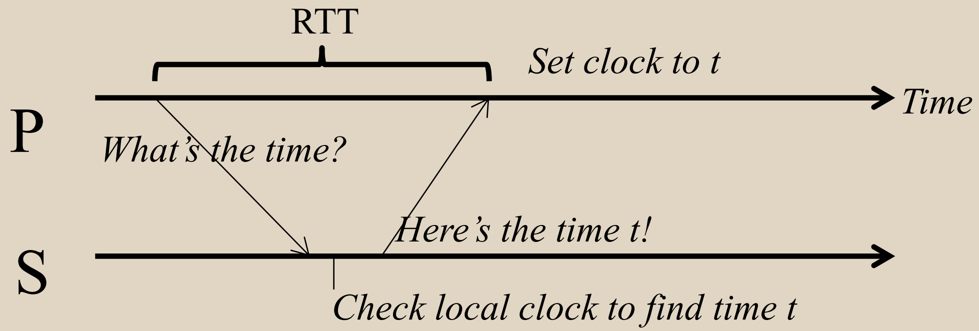
Cristian's Algorithm (3)

- P measures the round-trip-time RTT of message exchange
- Suppose we know the minimum P → S latency min1
- And the minimum S → P latency min2
  - min1 and min2 depend on Operating system overhead to buffer messages, TCP time to queue messages, etc.
- The actual time at P when it receives response is between [t+min2, t+RTT-min1]



Cristian's Algorithm (4)

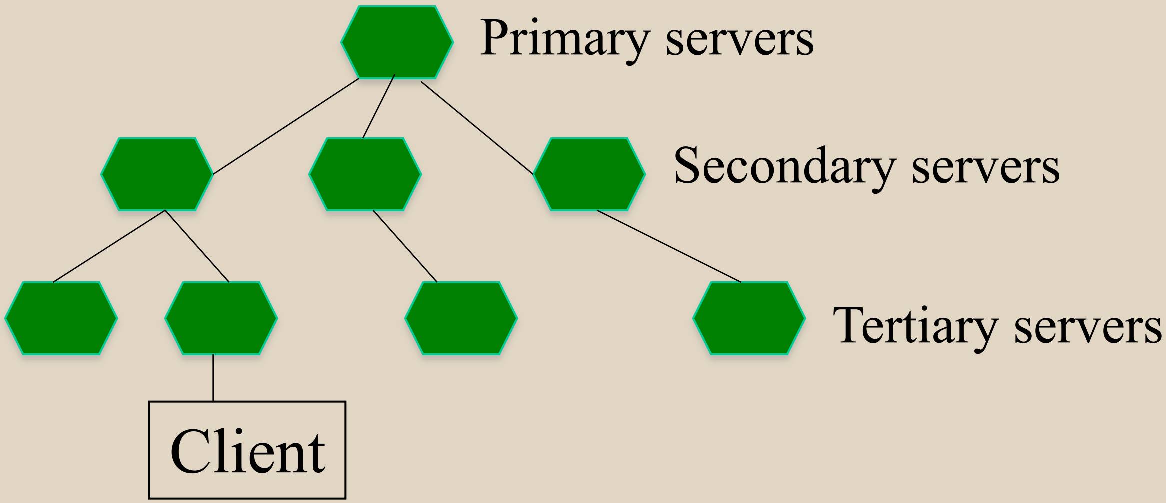
- The actual time at P when it receives response is between [t+min2, t+RTT-min1]
- P sets its time to halfway through this interval
  - To: t + (RTT+min2-min1)/2
- Error is at most (RTT-min2-min1)/2
  - Bounded!



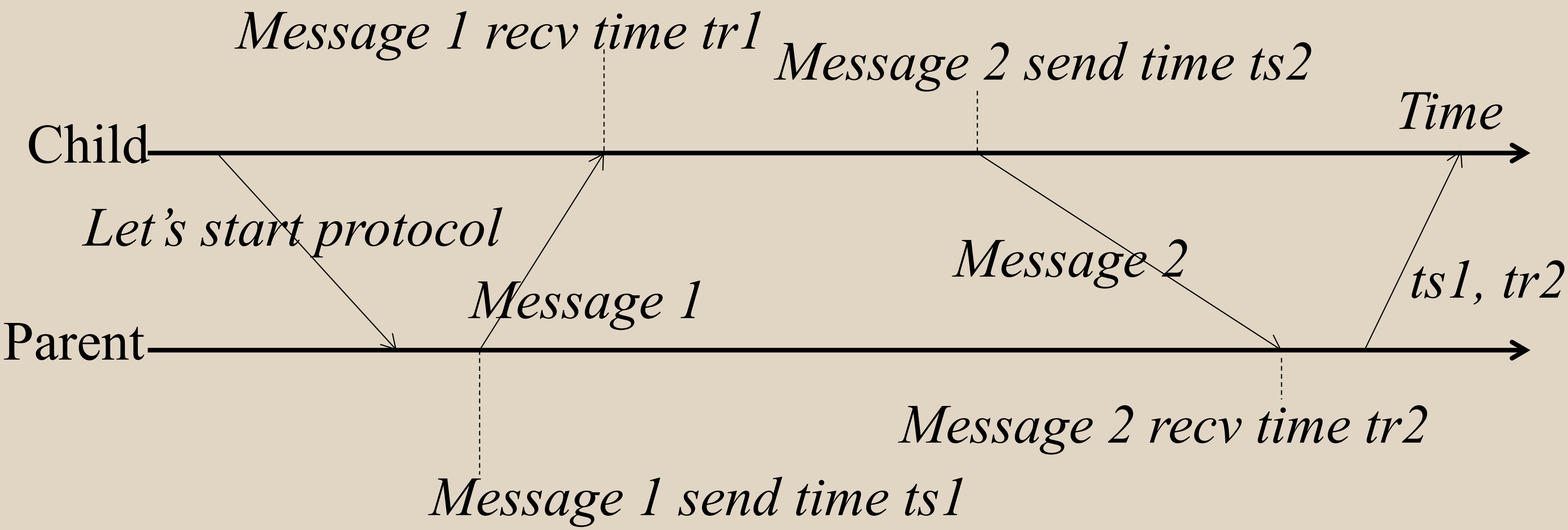
NETWORK TIME PROTOCOL (NTP)

NTP = Network Time Protocol

- NTP Servers organized in a tree
- Each Client = a leaf of tree
- Each node synchronizes with its tree parent



NTP Protocol



Basic Properties

- Let t and t' be the transmission times for Message 1 and Message 2, respectively
- Let d be the total transmission time of the two messages
  - $d = t + t'$
- Let o be the offset of the clocks
- We now have
- $tr1 = ts1 + t + o$
- $tr2 = ts2 + t' - o$
- $d = t + t' = tr1 - ts1 + tr2 - ts2$ (Can be thought of as RTT)
- $o = o_i + (t' - t)/2$, where $o_i = (tr1 - ts1 + ts2 - tr2)/2$

What the Child Does

- Child calculates *offset* between its clock and parent's clock
- Uses *ts1, tr1, ts2, tr2*
- Offset is estimated as

$$o \approx o_i = (tr1 - tr2 + ts2 - ts1)/2$$

- Error is bounded by $d/2$

- NTP uses multiple pairs of (o_i, d) to obtain a more accurate clock

And yet…

- **We still have a non-zero error!**
- **We just can't seem to get rid of error**
  - Can't, as long as message latencies are non-zero
- **Can we avoid synchronizing clocks altogether, and still be able to order events?**

LOGICAL CLOCK

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

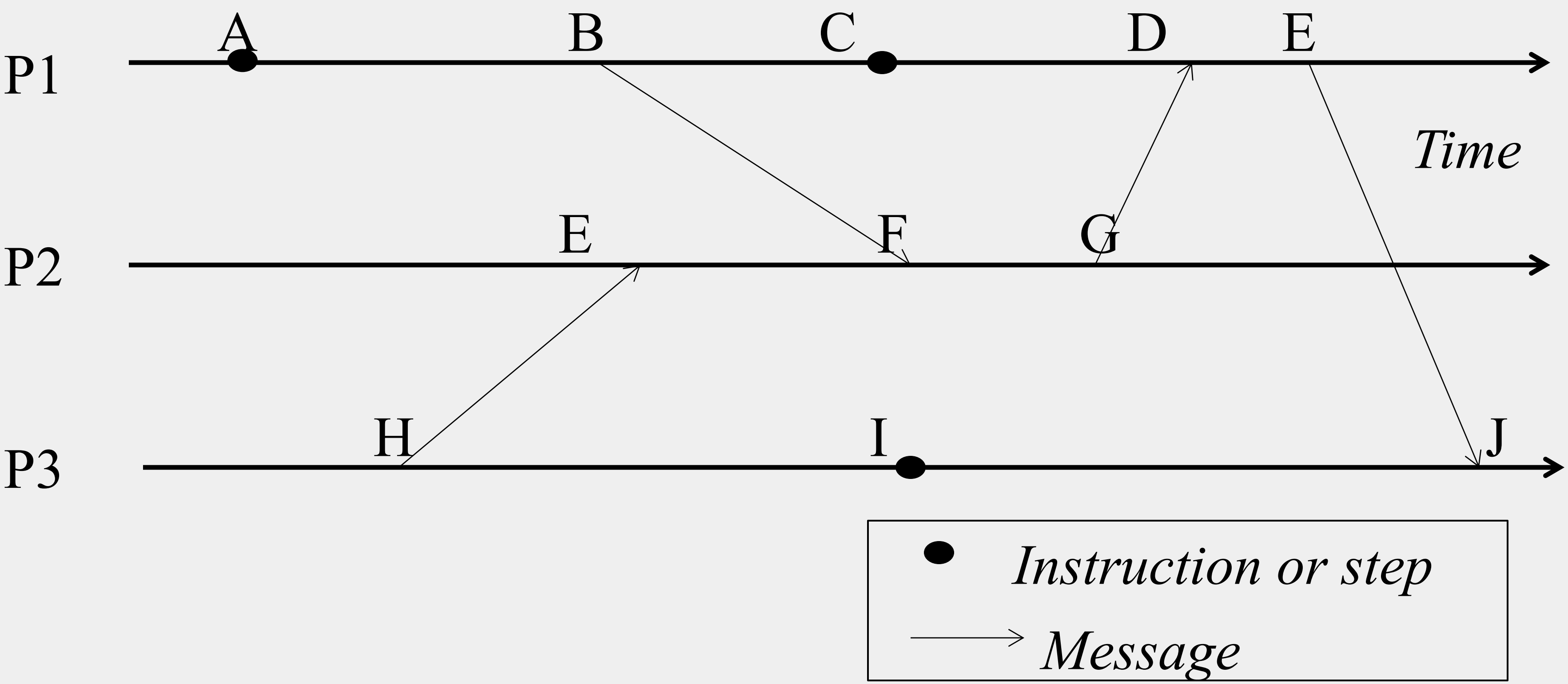
Logical (or Lamport) Ordering(2)

- Define a logical relation *Happens-Before* among pairs of events
- Happens-Before denoted as $\rightarrow$
- Three rules

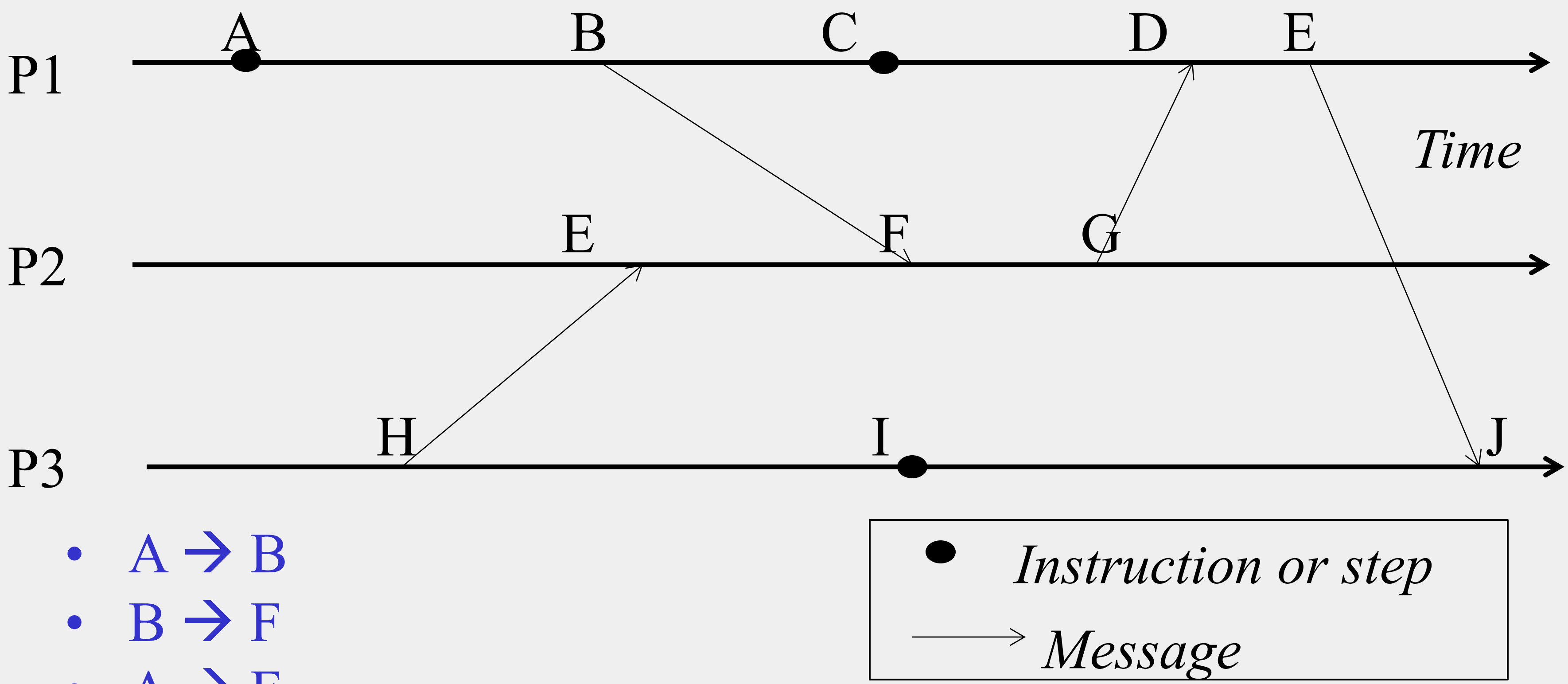
1. On the same process: $a \rightarrow b$, if $time(a) < time(b)$ (using the local clock)
2. If p1 sends *m* to p2: $send(m) \rightarrow receive(m)$
3. (Transitivity) If $a \rightarrow b$ *and* $b \rightarrow c$ then $a \rightarrow c$

- Creates a *partial order* among events
  - Not all events related to each other via $\rightarrow$

Example

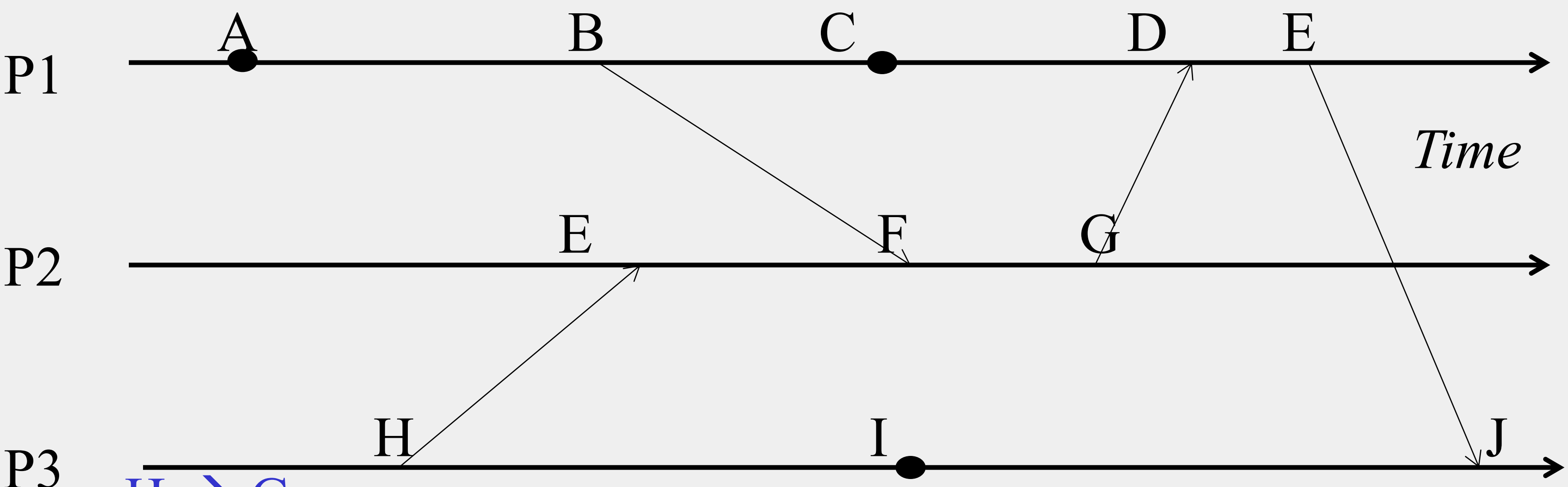


Happens-Before

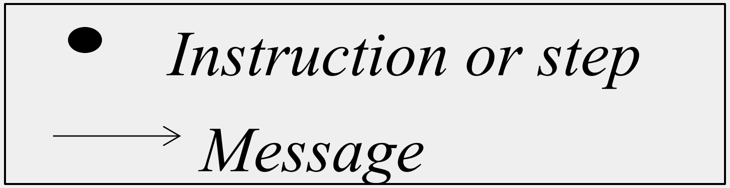


- A → B
- B → F
- A → F

Happens-Before (2)



- H → G
- F → J
- H → J
- C → J

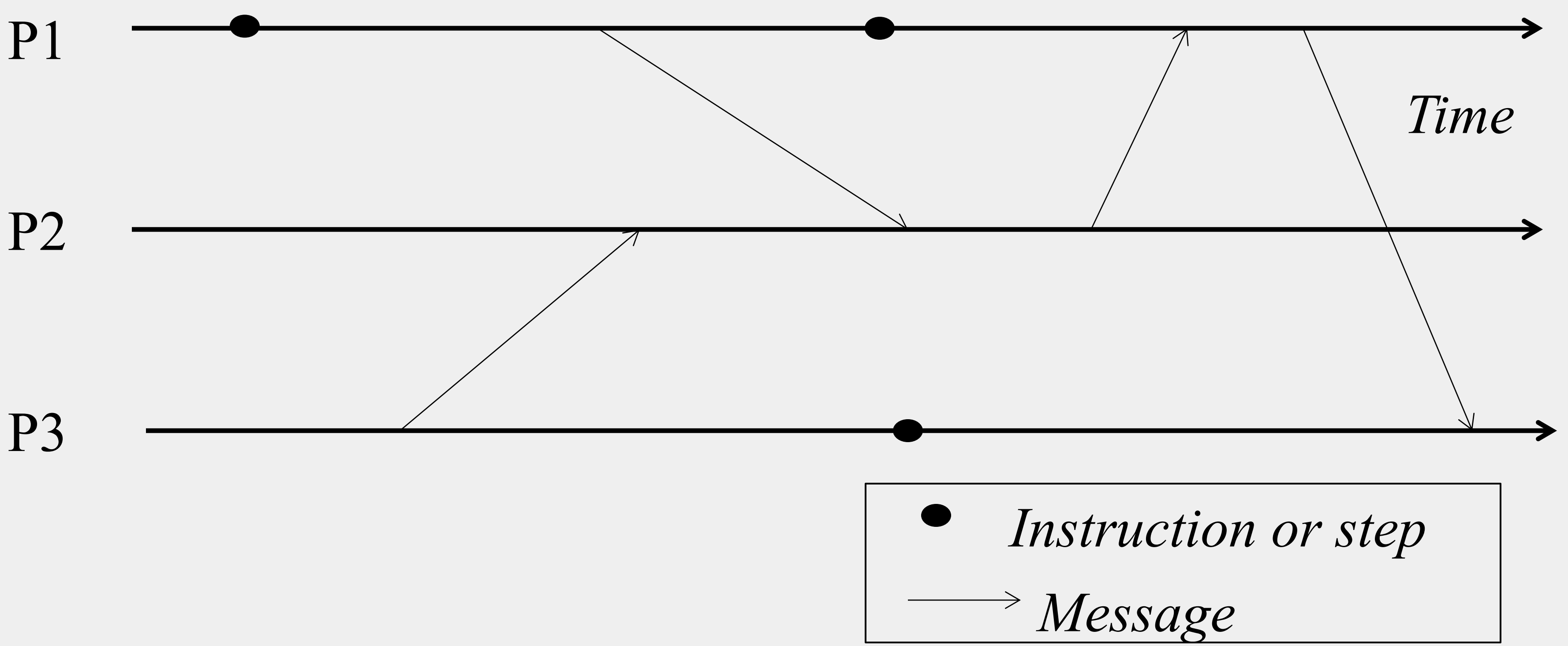


In practice: Lamport timestamps

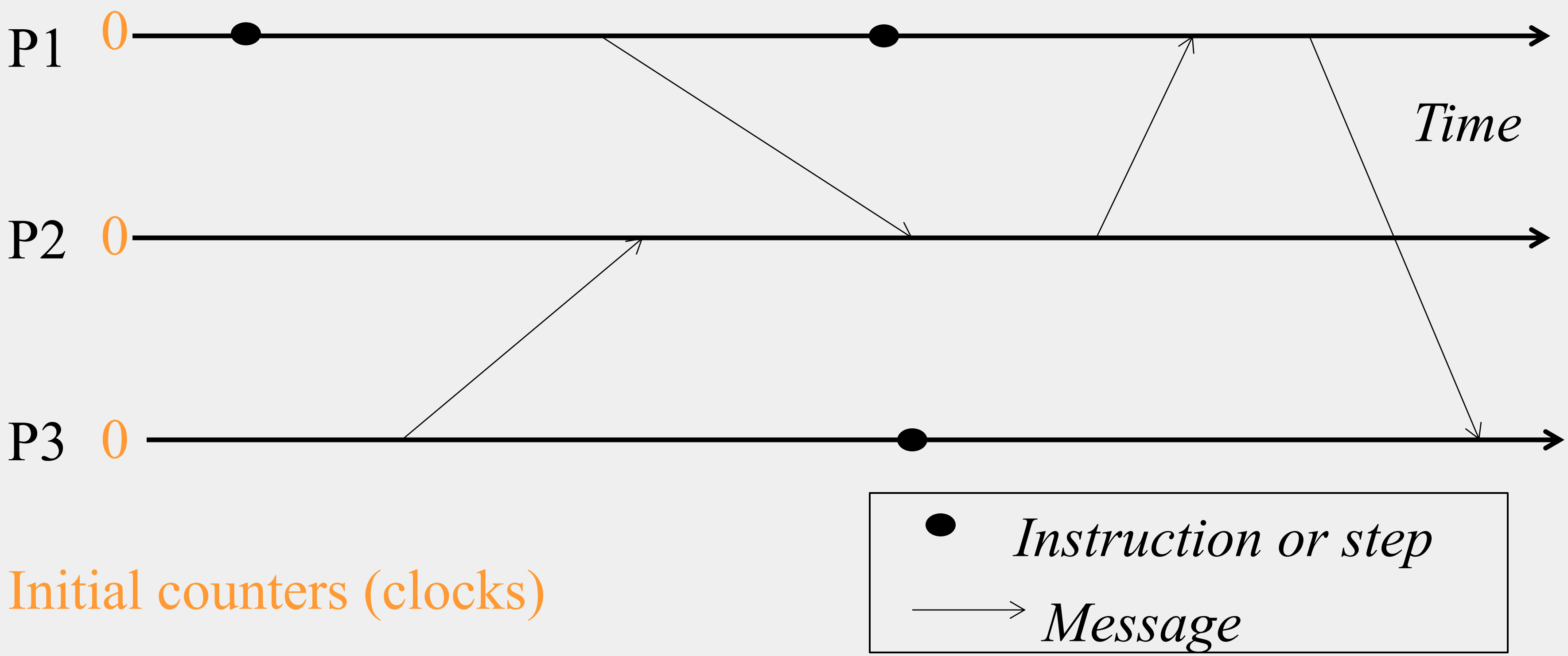
- **Goal: Assign logical (Lamport) timestamp to each event**
- **Timestamps obey causality**
- **Rules**
  - Each process uses a local counter (clock) which is an integer
    - initial value of counter is zero
  - A process increments its counter when a send or an instruction happens at it. The counter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

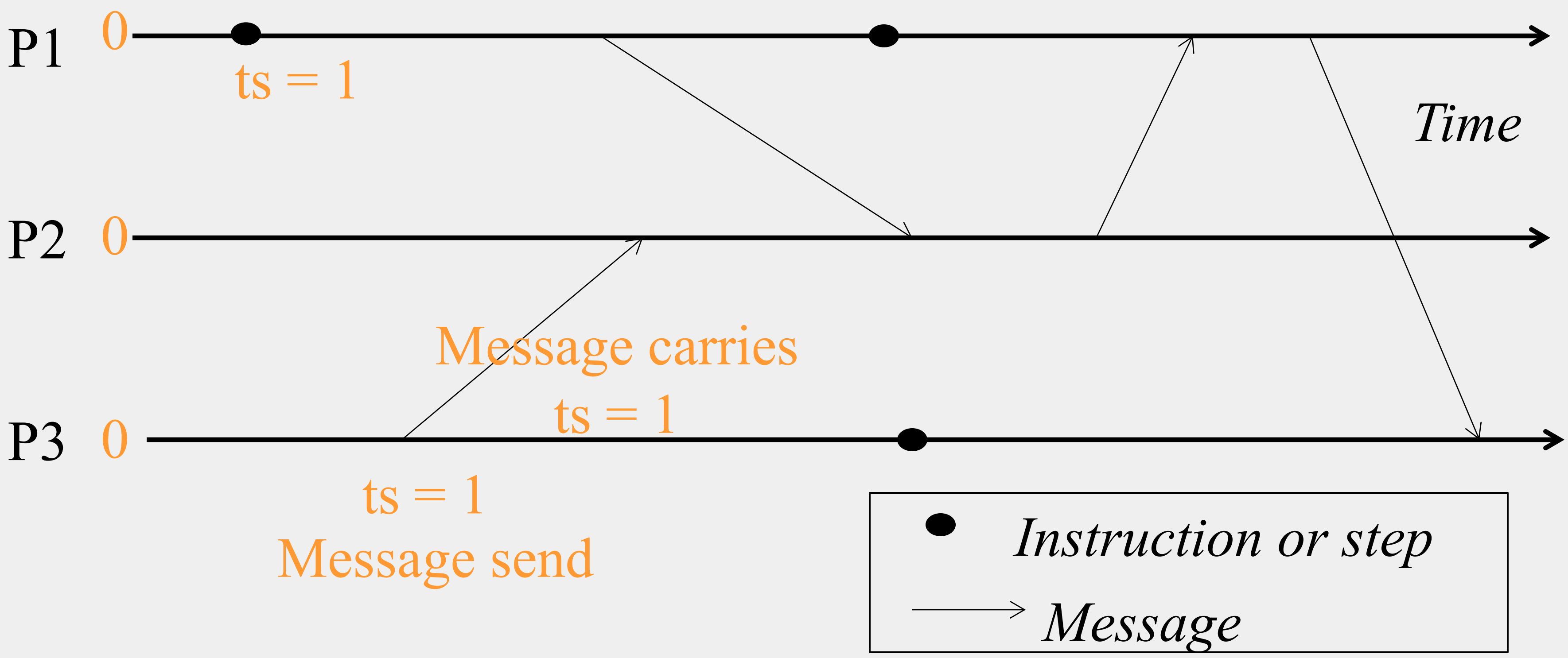


Lamport Timestamps

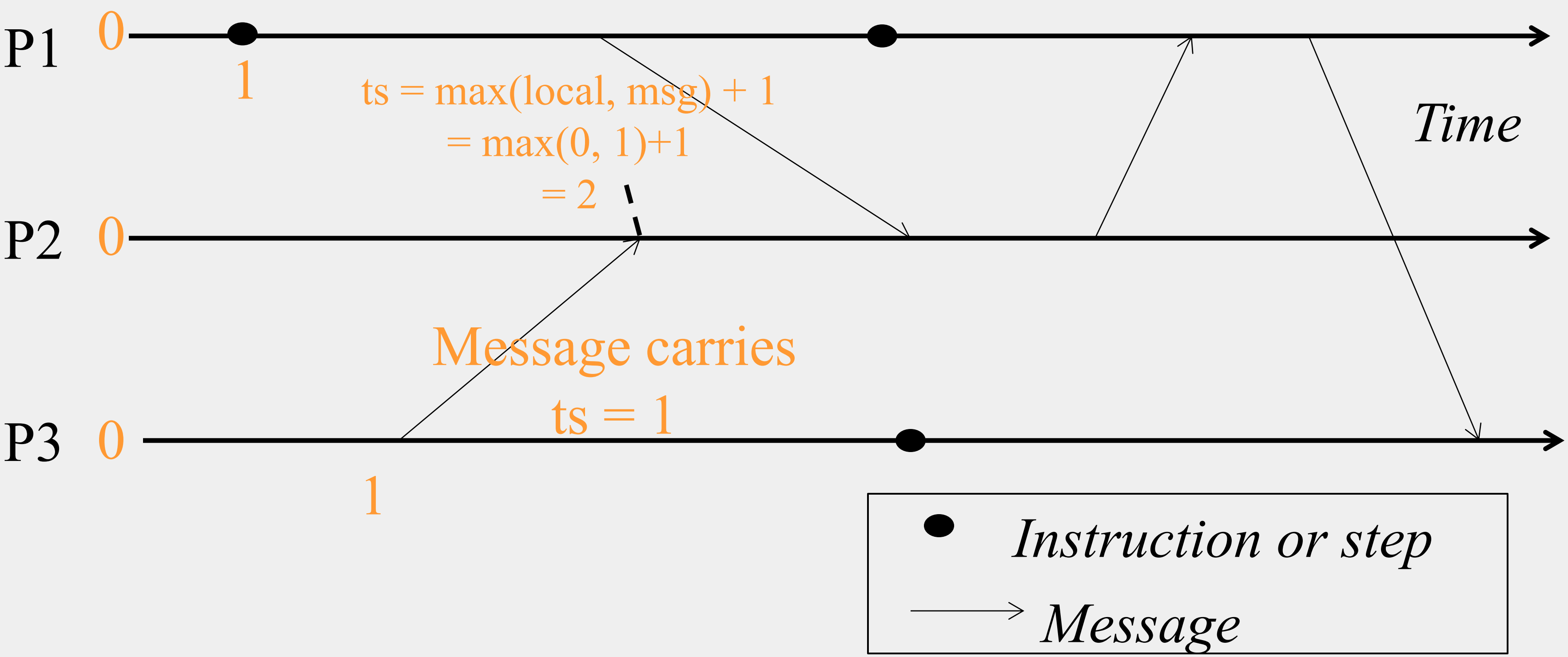


Initial counters (clocks)

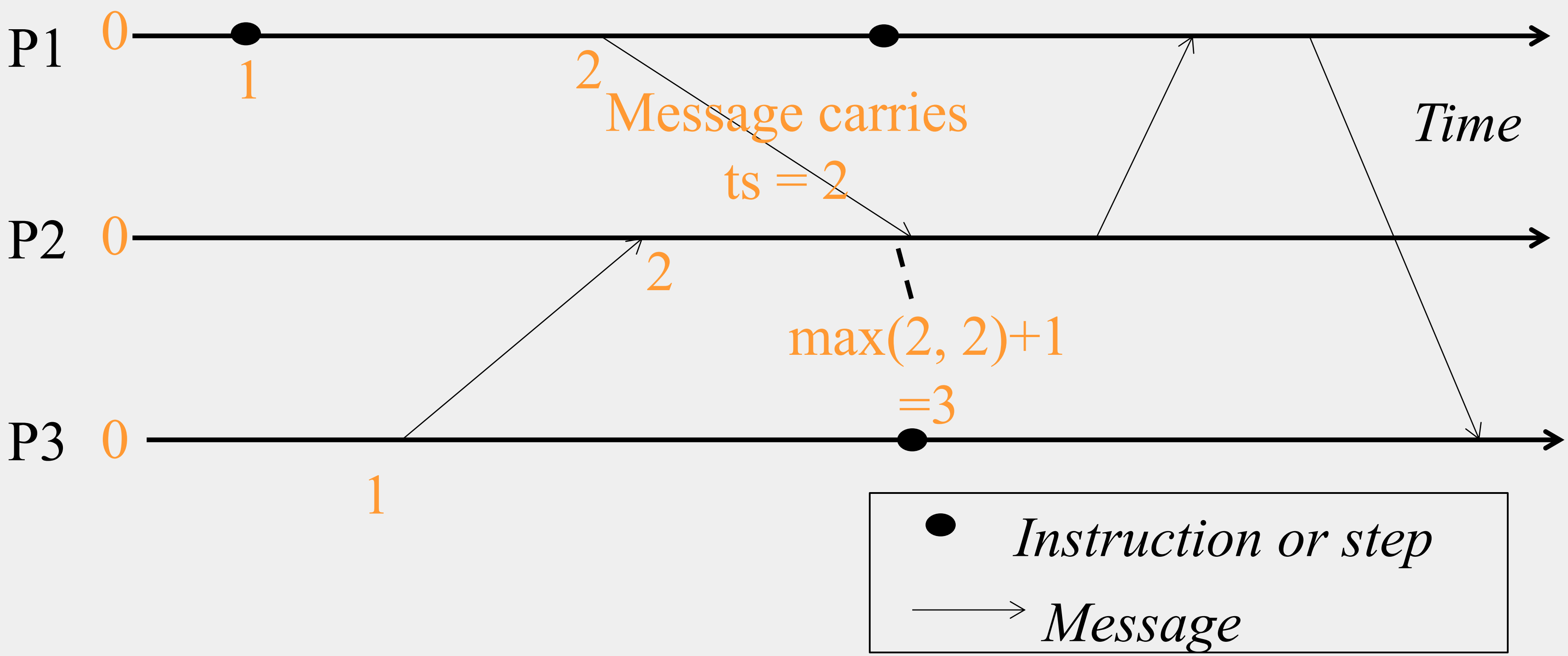
Lamport Timestamps



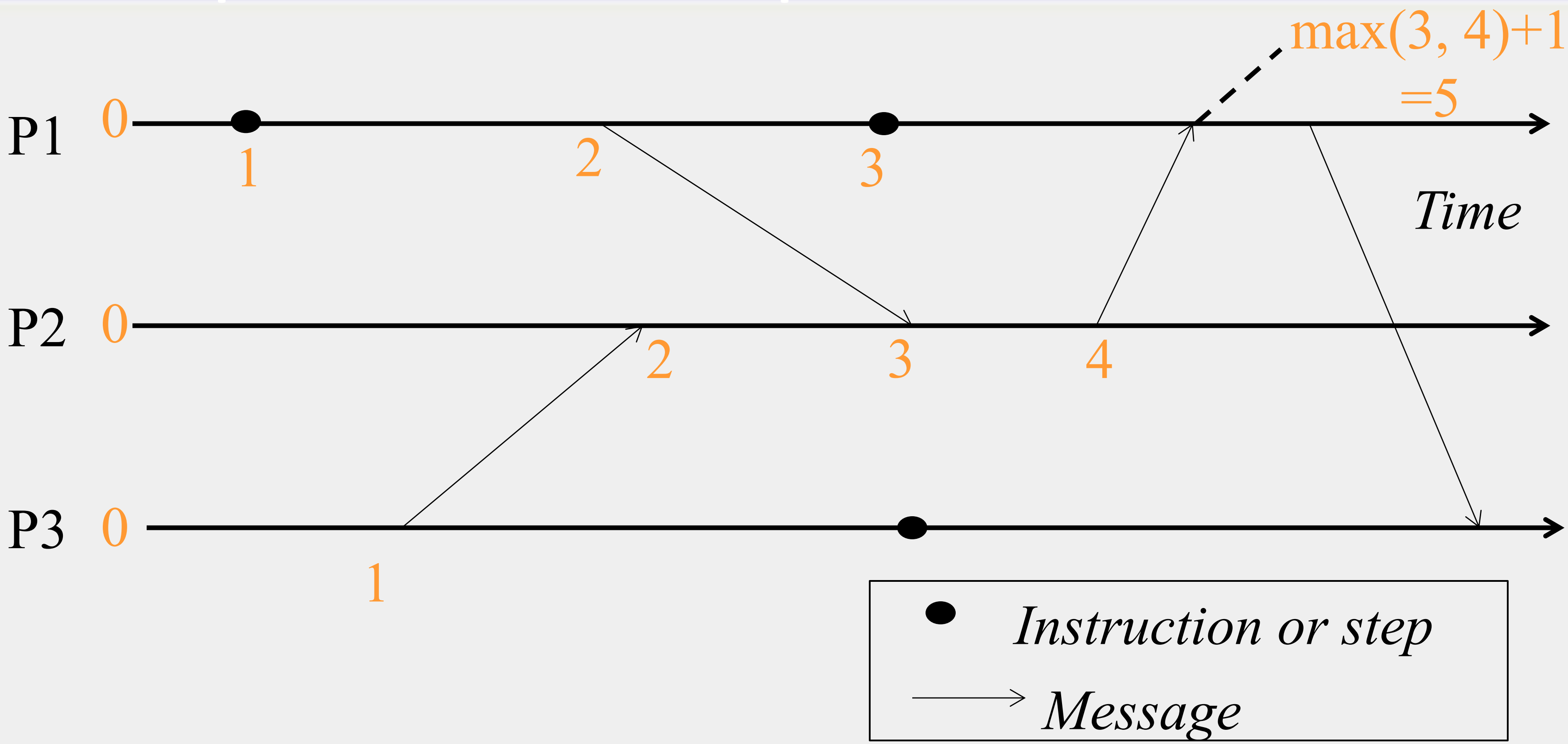
Lamport Timestamps



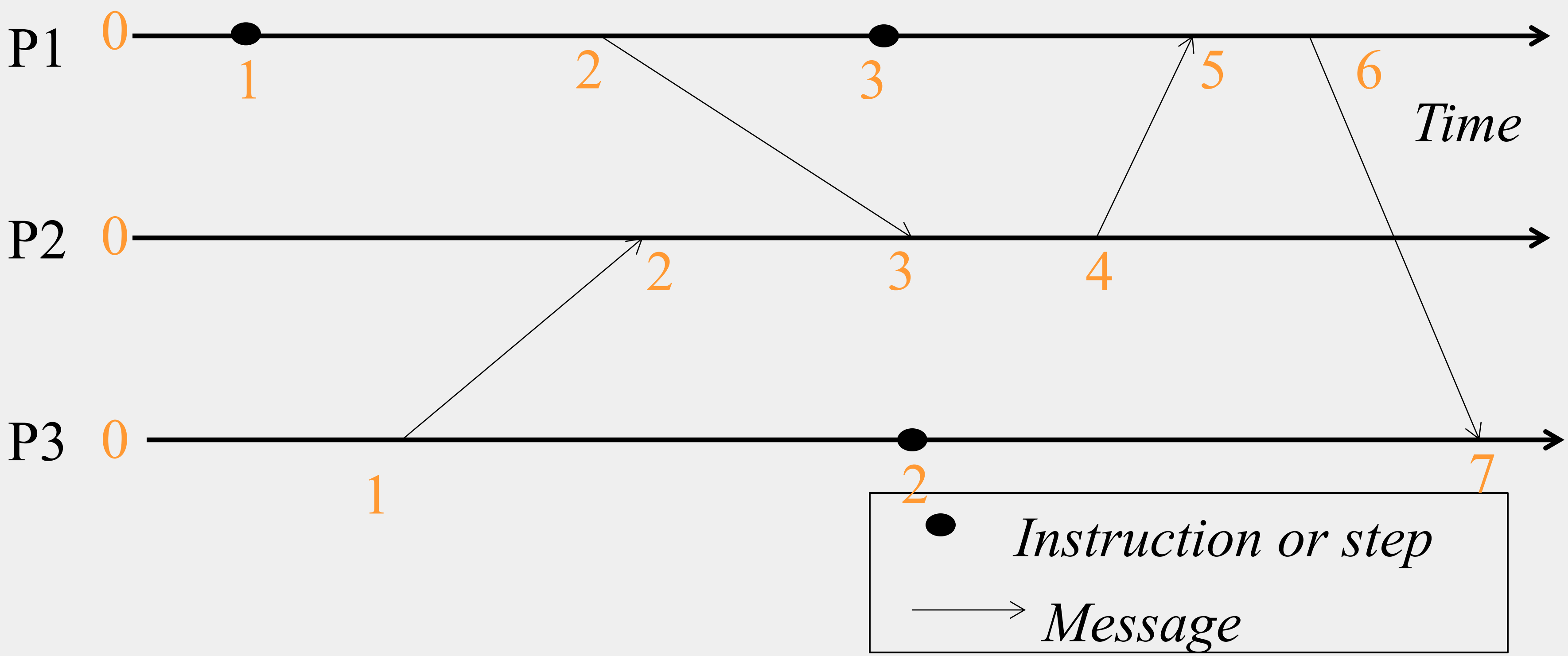
Lamport Timestamps



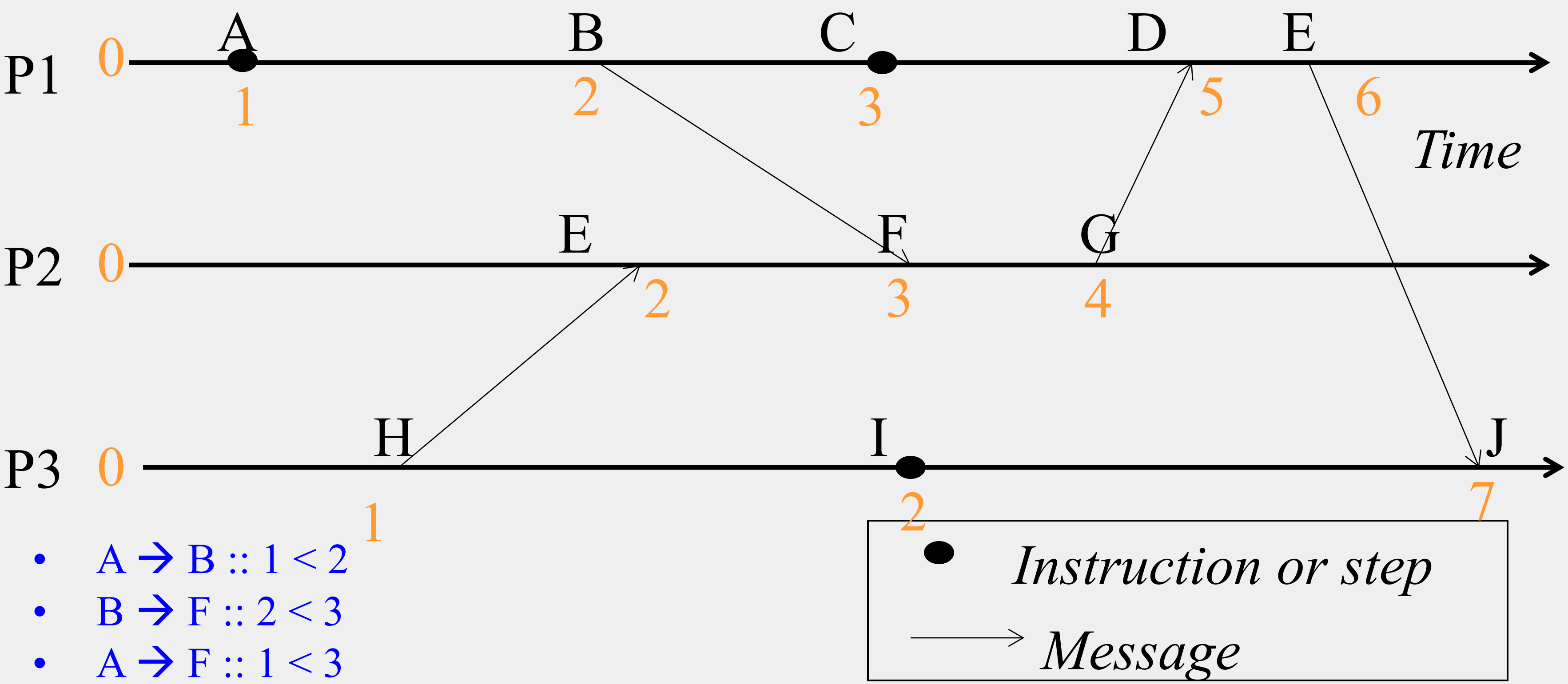
Lamport Timestamps



Lamport Timestamps

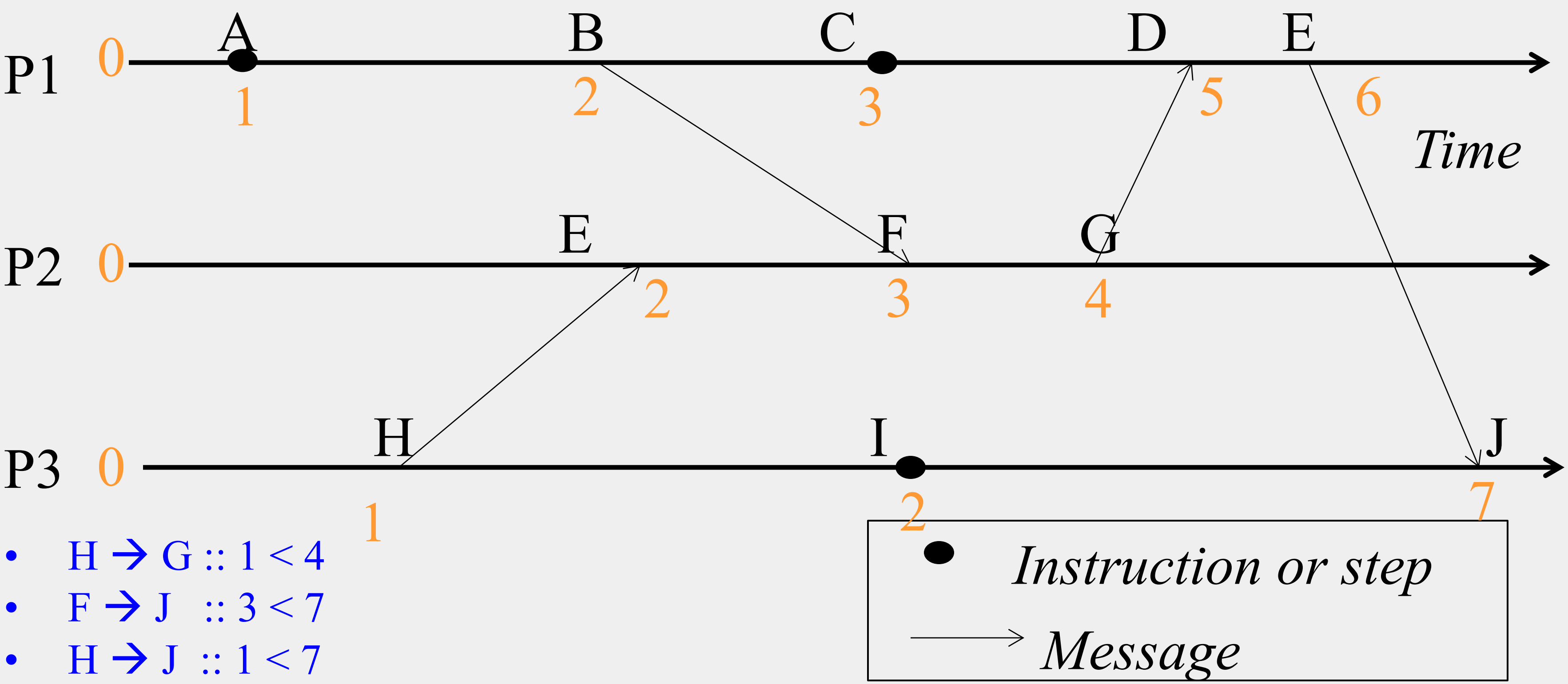


Obeying Causality



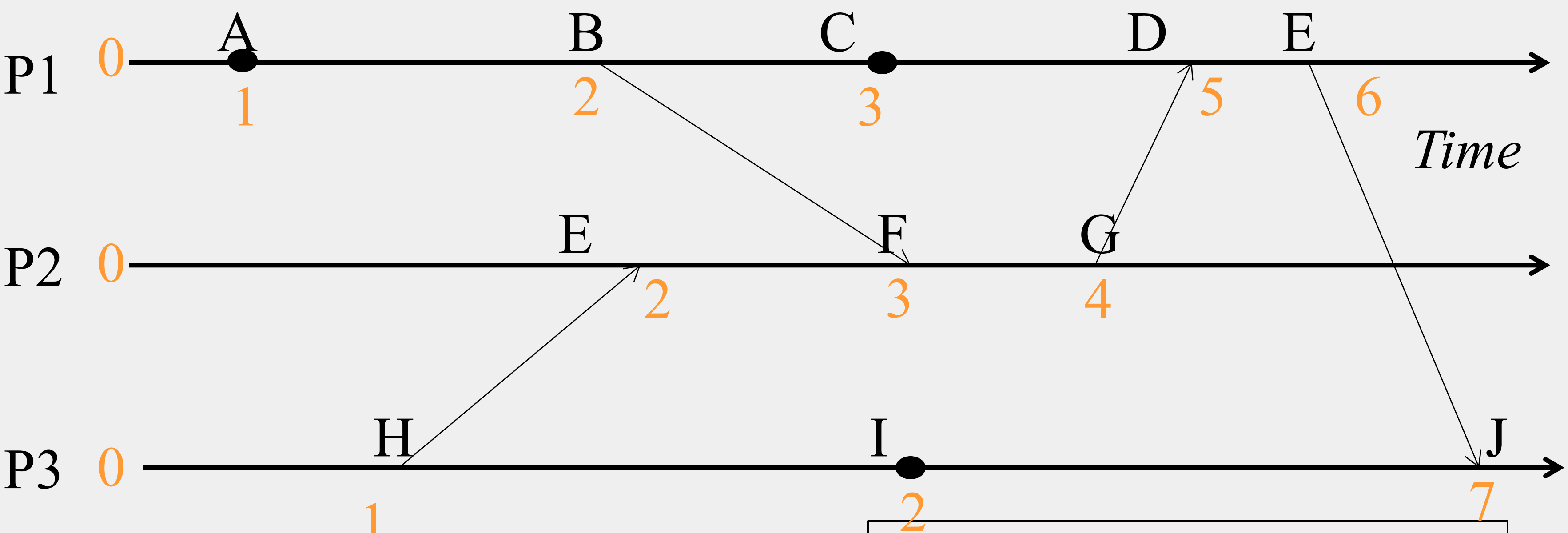
- A → B :: 1 < 2
- B → F :: 2 < 3
- A → F :: 1 < 3

Obeying Causality (2)

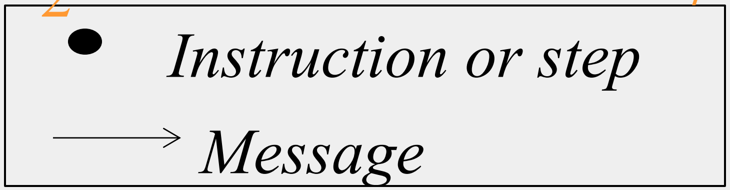


- H → G :: 1 < 4
- F → J :: 3 < 7
- H → J :: 1 < 7
- C → J :: 3 < 7

Not always *implying* Causality



- ? C → F ? :: 3 = 3
- ? H → 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**

  $E1 \rightarrow E2 \Rightarrow$ timestamp(E1) < timestamp (E2), BUT

  timestamp(E1) < timestamp (E2) $\Rightarrow$

  {E1 $\rightarrow$ E2} OR {E1 and E2 concurrent}

VECTOR CLOCK

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

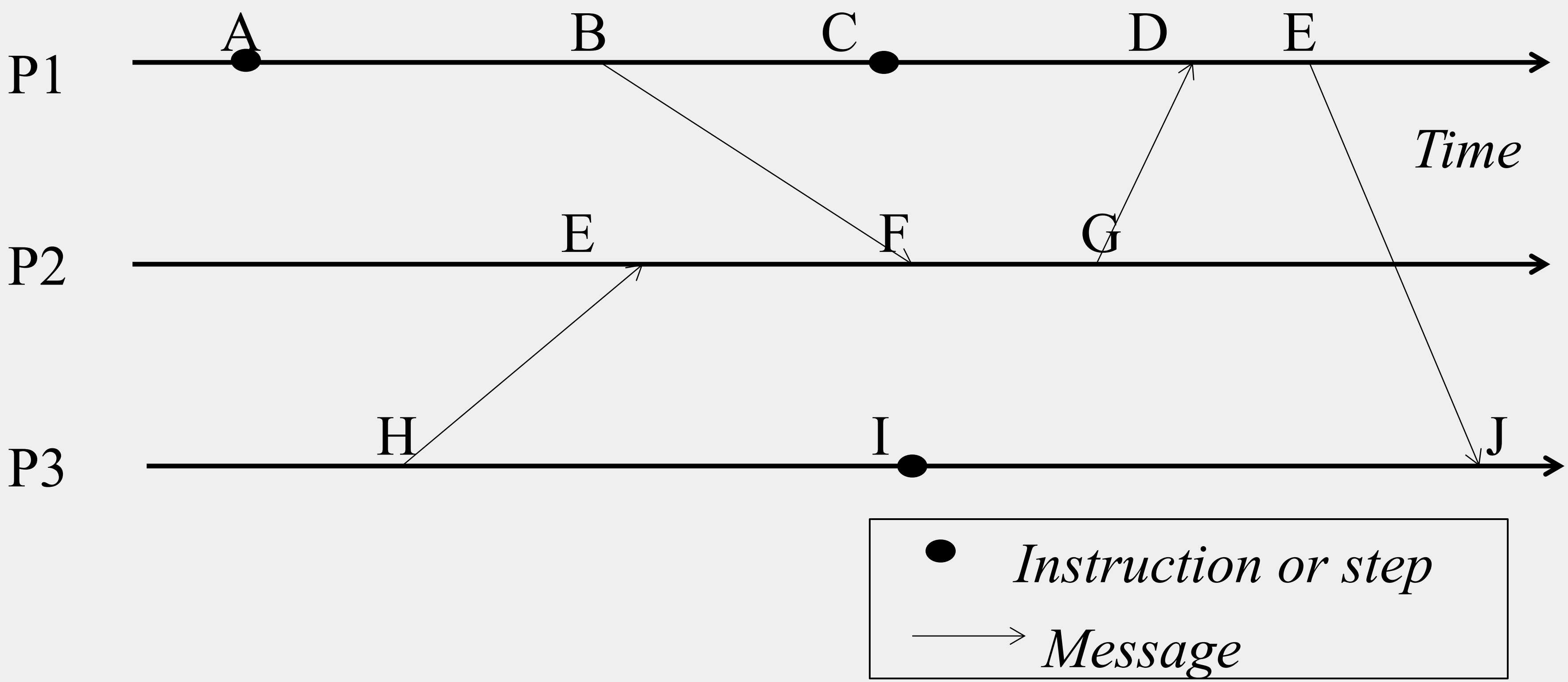
- Incrementing vector clocks

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*:

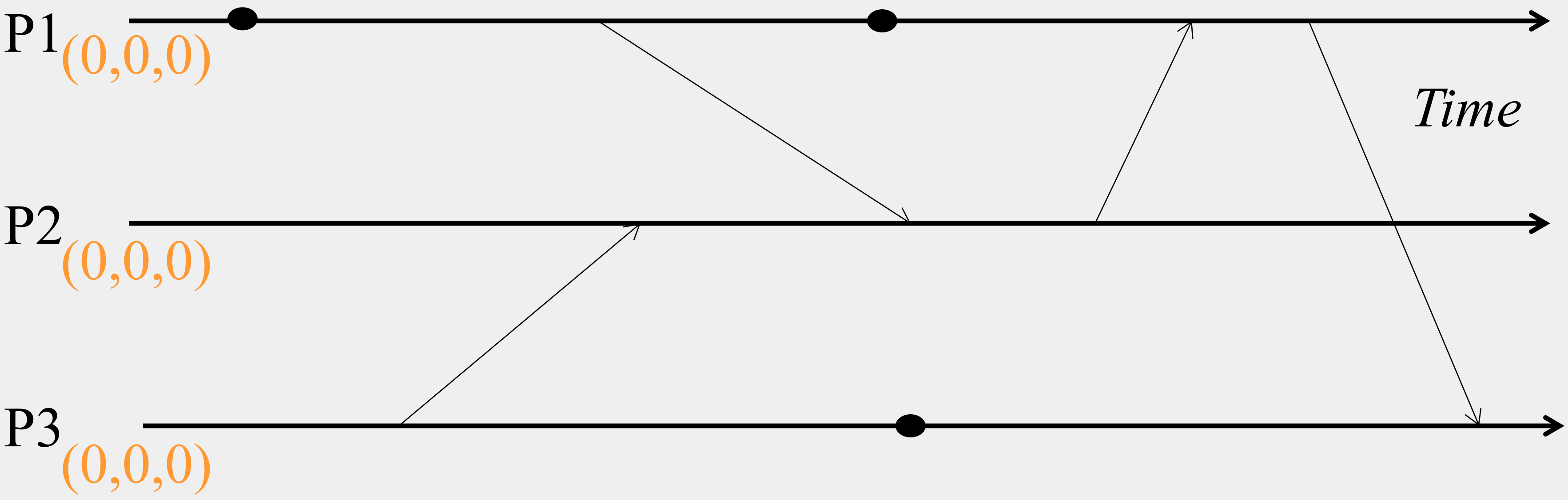
$$\mathbf{V}_i[i] = \mathbf{V}_i[i] + 1$$

$$\mathbf{V}_i[j] = \max(\mathbf{V}_{message}[j], \mathbf{V}_i[j]) \text{ for } j \neq i$$

Example

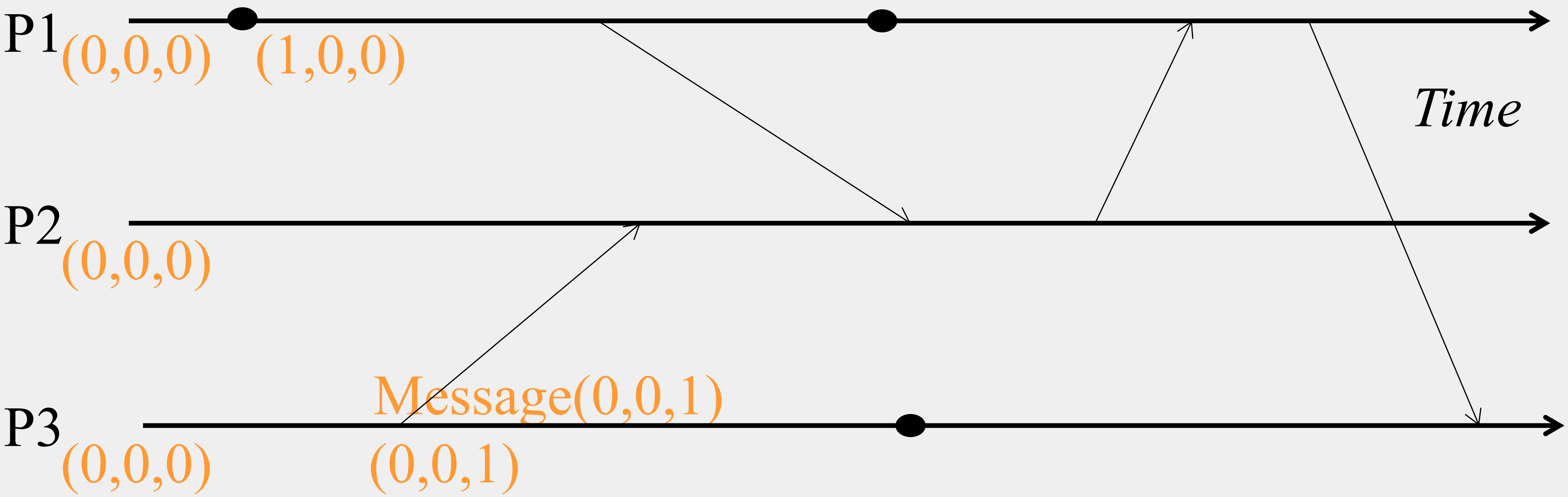


Vector Timestamps

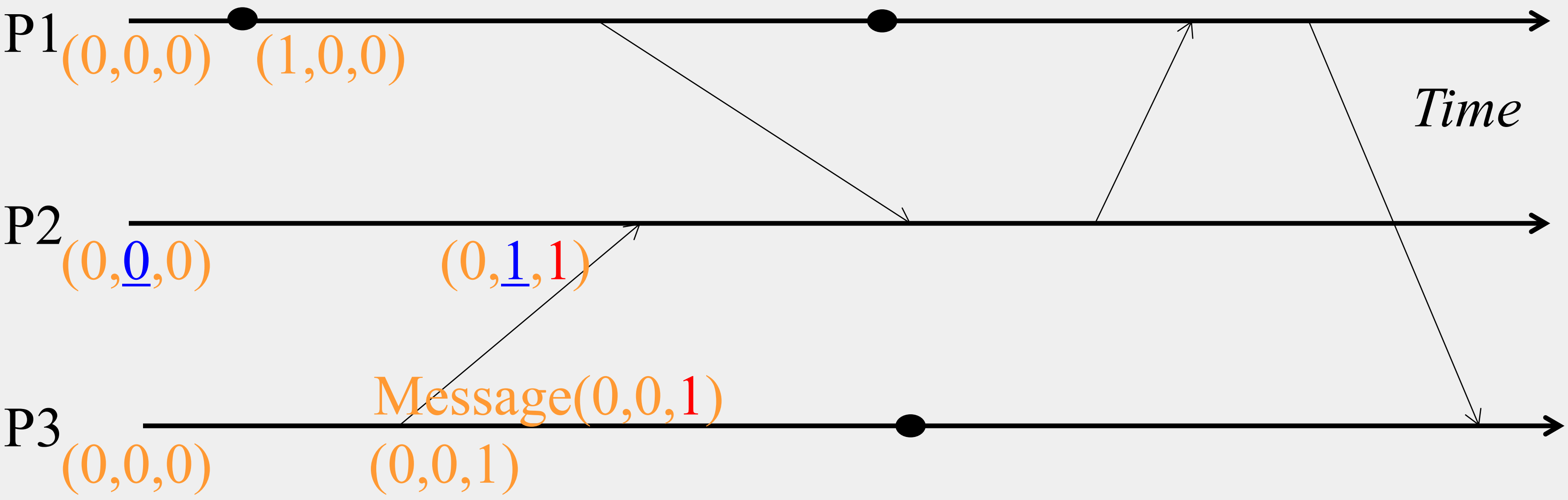


Initial counters (clocks)

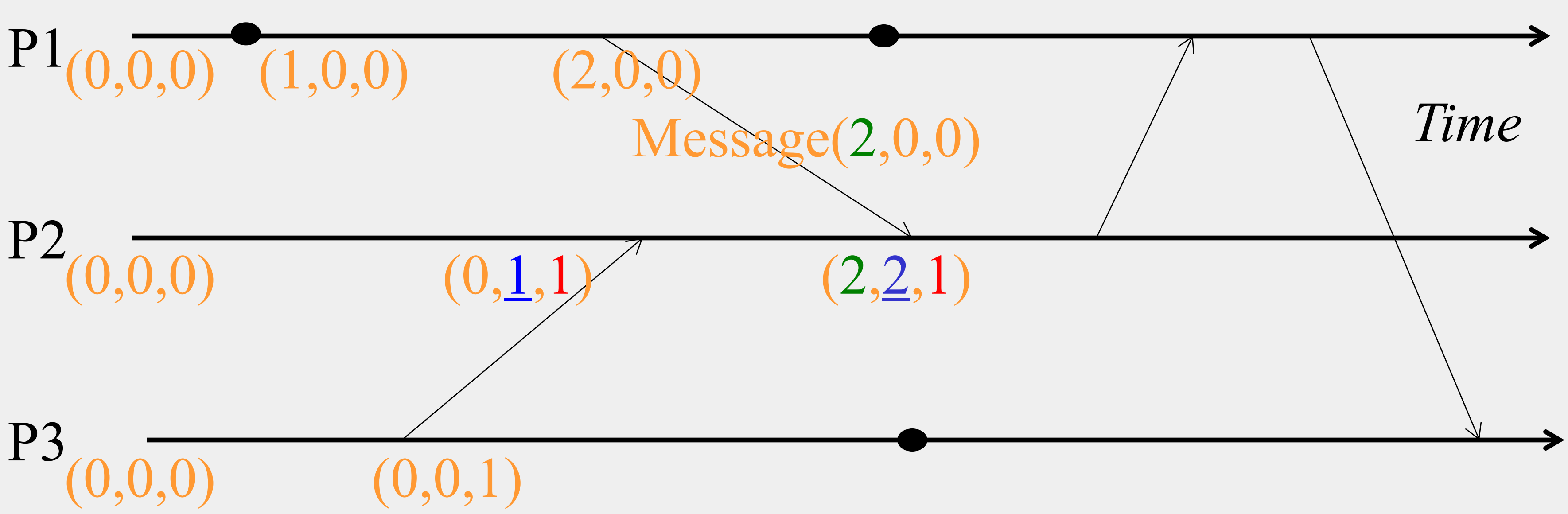
Vector Timestamps



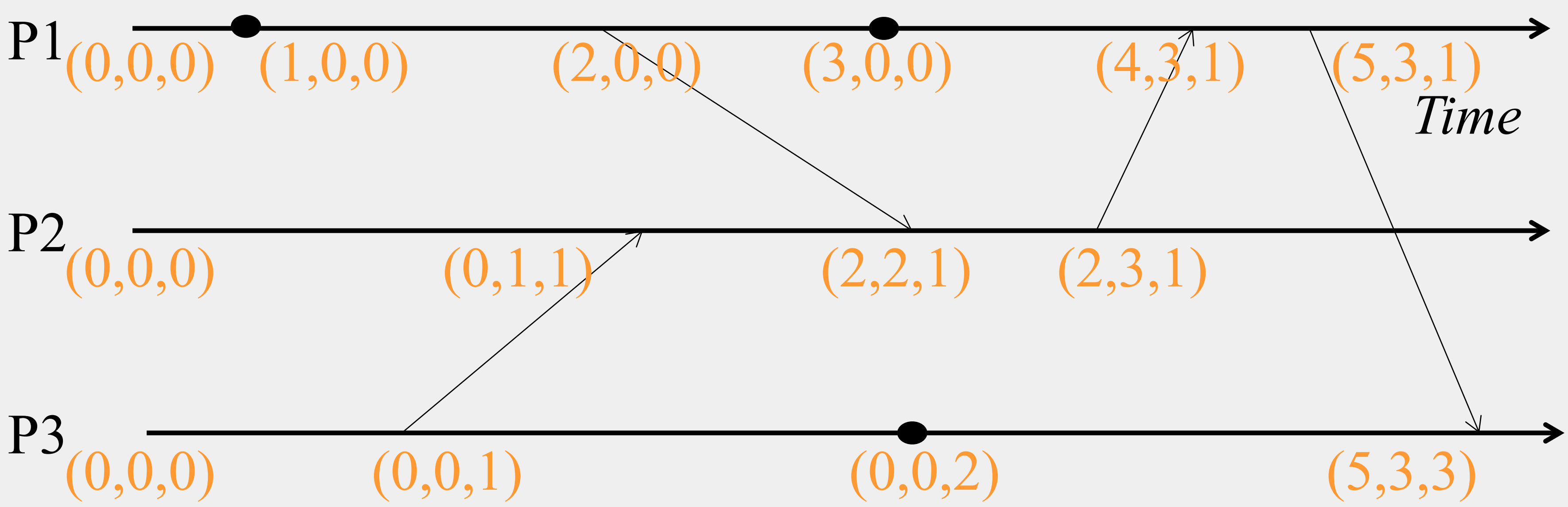
Vector Timestamps



Vector Timestamps



Vector Timestamps



Causally-Related …

- $VT_1 = VT_2$,

  iff (if and only if)

  $VT_1[i] = VT_2[i]$, for all $i = 1, \ldots, N$
- $VT_1 \leq VT_2$,

  iff $VT_1[i] \leq VT_2[i]$, for all $i = 1, \ldots, N$
- Two events are causally related *iff*

  $VT_1 < VT_2$, i.e.,

  iff $VT_1 \leq VT_2$ &

  there exists j such that

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

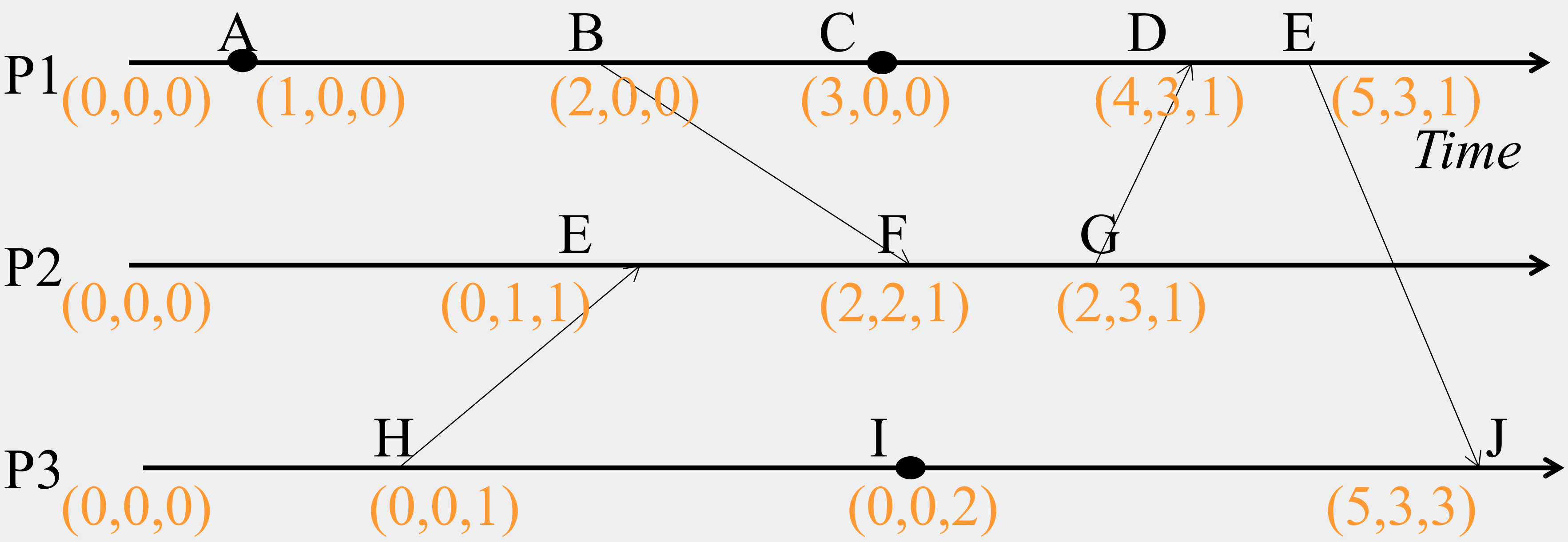
… or Not Causally-Related

- Two events VT_1 and VT_2 are concurrent
  iff

  NOT ($VT_1 \leq VT_2$) AND NOT ($VT_2 \leq VT_1$)

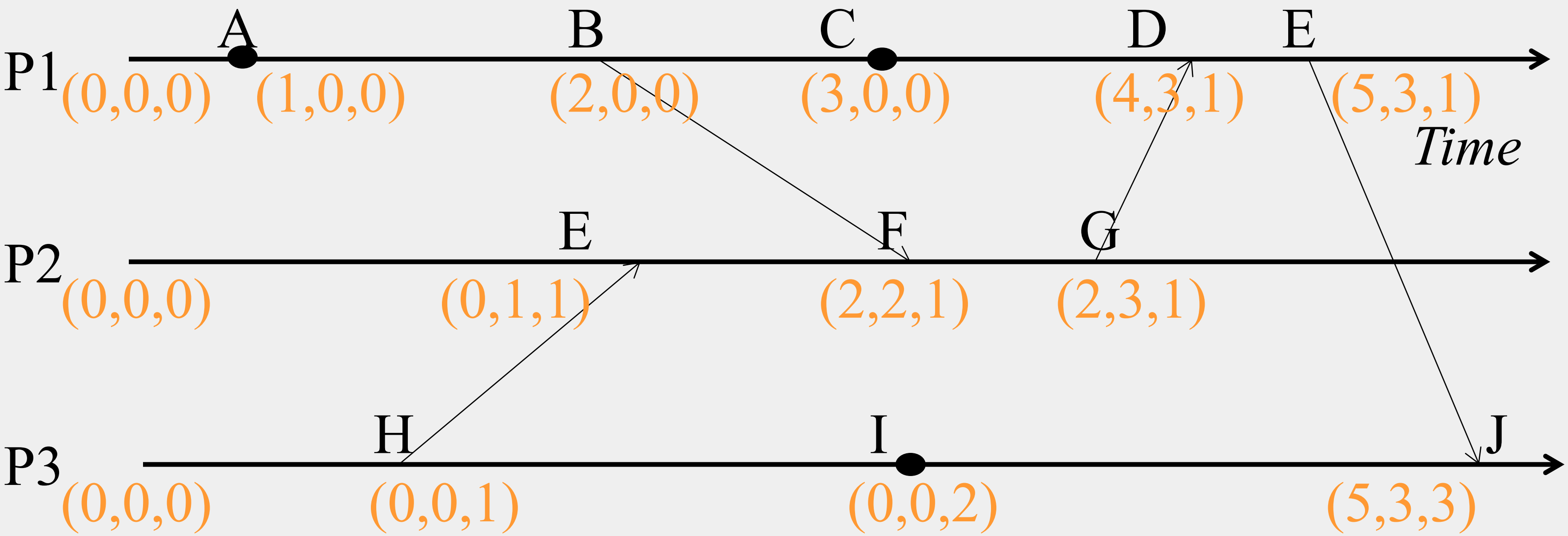
  We'll denote this as $VT_2 ||| VT_1$

Obeying Causality



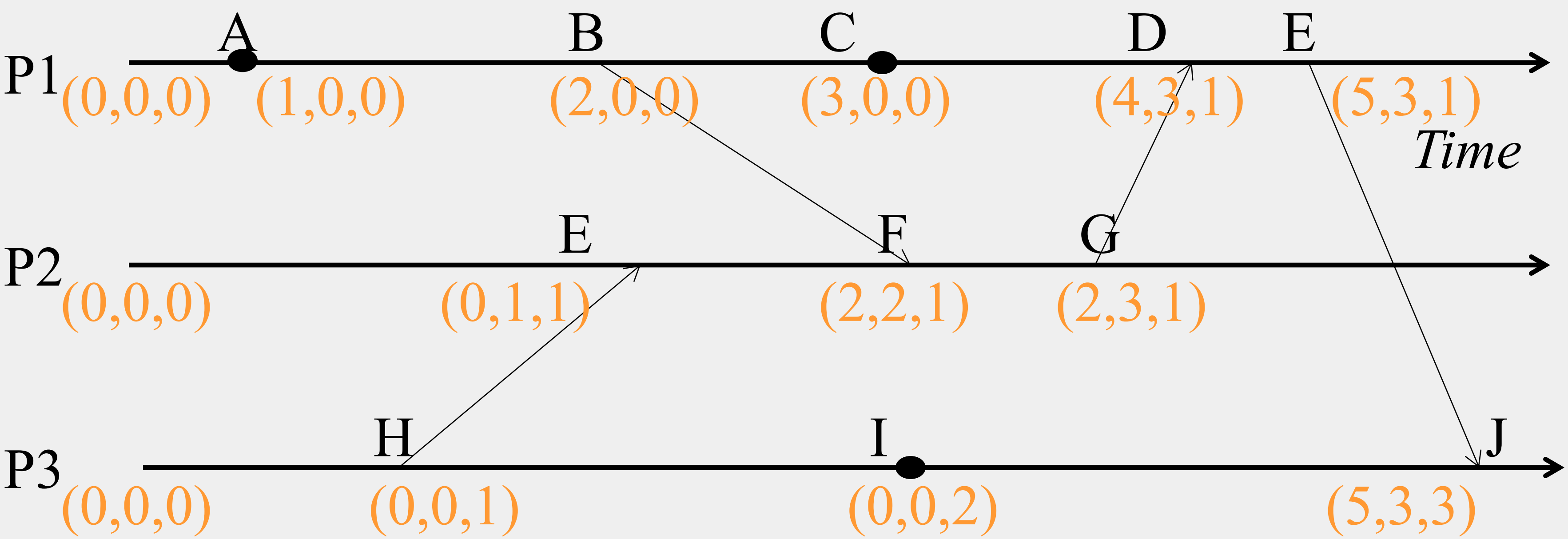
- A → B :: (1,0,0) < (2,0,0)
- B → F :: (2,0,0) < (2,2,1)
- A → F :: (1,0,0) < (2,2,1)

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