# M120: DISTRIBUTED SYSTEMS

## Time

*Slides are variant of slides provided by  Andrew Tanenbaum & MarartenVan Steen

# Time in Distributed Systems

- Related to notions of replication/consistency is notion of time
- Simplest (incomplete) defn of DS:  set of processes that communicate by msg passing and carrying out desired actions over time
- Components in DS need some sense of time for synchronizing and/or coordinating tasks
  - specs of DSs include terms like “when”, “before”, “after”, “simultaneously”

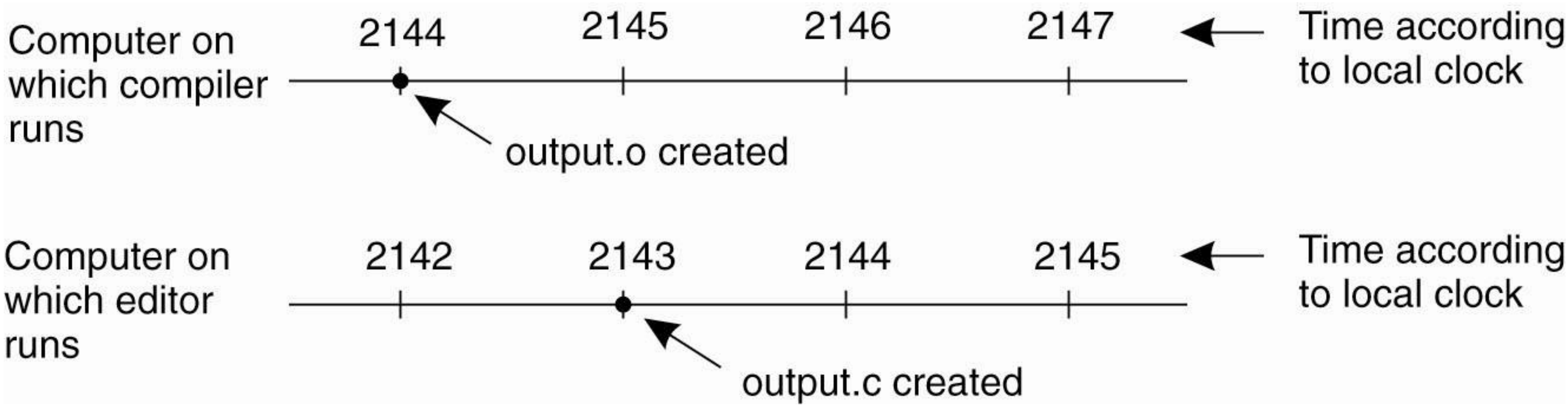
# Synchronization

- Allows processes
  - To share resources (e.g., data or printer) in orderly manner
  - To figure out ordering of events (msg1 from P was sent before msg2 from Q)
- Outline of lecture
  - Synchronization based on actual time
  - Synchronization where only relative ordering matters

# Clock synchronization

- In centralized system, time is unambiguous
  - Time T1: A asks for time, gets back T1 from kernel
  - At time T2>T1, B asks for time, gets back T2
  - T2 returned to B will always be >= T1 returned to A
- In a DS, achieving agreement on time is NOT trivial
  - Example of why clock syncing is important: running make on multiple machines

# Why clock synchronization is important



- When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

# Why clock synchronization is important (2)

- Cloud airline reservation system
- Server A receives a client request to purchase last ticket on flight ABC 123.
- Server A timestamps purchase using local clock 9h:15m:32.45s, and logs it. Replies ok to client.
- That was the last seat. Server A sends message to Server B saying “flight full.”
- B enters “Flight ABC 123 full” + its own local clock value (which reads 9h:10m:10.11s) into its log.
- Server C queries A’s and B’s logs. Is confused that a client purchased a ticket at A after the flight became full at B.
- This may lead to further incorrect actions by C

# Why is clock synchronization a challenge?

- **End hosts in Internet-based distributed systems (like clouds)**
  - Each have their own clocks
  - Unlike processors (CPUs) within one server or workstation which share a system clock
- **Processes in Internet-based systems follow an *asynchronous* system model**
  - No bounds on
    - Message delays
    - Processing delays
  - Unlike multi-processor (or parallel) systems which follow a *synchronous* system model

# 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.

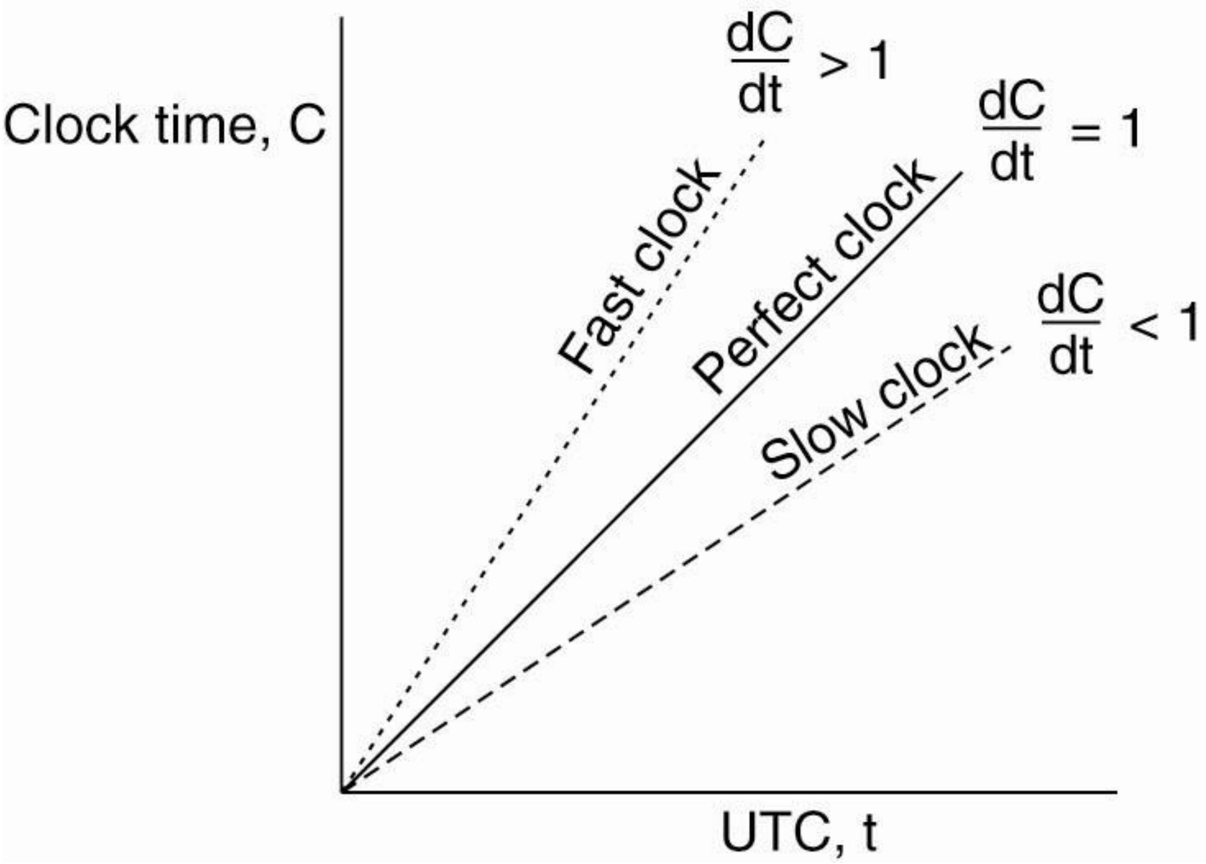
# Possible to synchronize all clocks in a DS?

- It’s surprisingly complicated
- Computers suffer from clock skew (aka drift)
  - In system of *n* computers, very likely each has different time *(even if started out same)*
- UTC (Coordinated Universal Time)
  - Time standard by which world regulates clocks & time
  - Based on the use of cesium 133 atomic clocks
  - Shortwave radio stations in several countries broadcast short pulse at start of a UTC second to receivers that need precise time

# Model

- Each machine has timer that causes interrupt H times/sec
- On interrupt, add 1 to a software clock that tracks number of ticks, C, since some agreed-upon time in the past
- Ideally, when UTC=*t*, C = *t*

# Clock Synchronization Algorithms

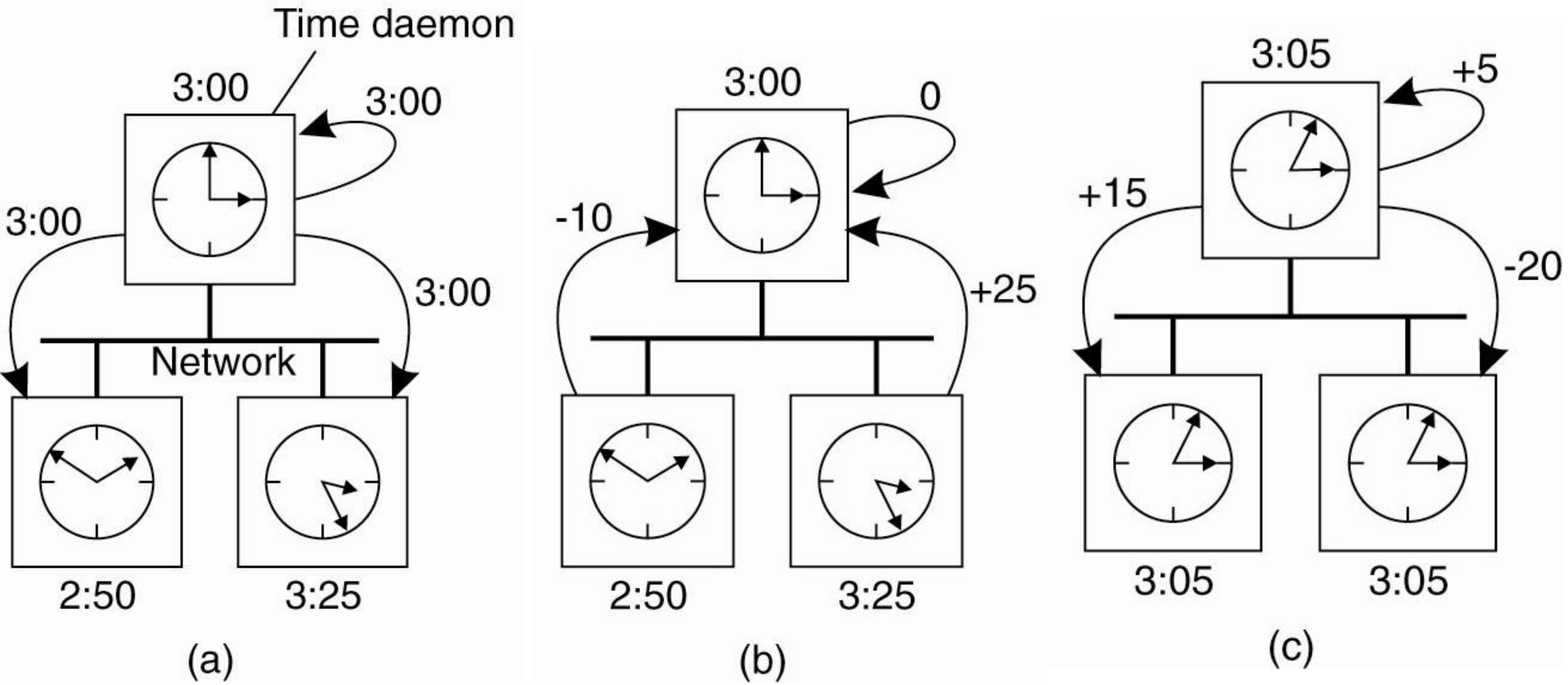


- The relation between clock time and UTC when clocks tick at different rates.

# Clock Synchronization Algorithms

- Internal synchronization
  - Goal: each process tracks its own time, try to keep all processes together
    - Every pair of processes in group have clocks within bound D
    - |C(i)-c(k)| < D at all times, for all processes i and k
  - E.g. Berkeley algorithm
- External Synchronization
  - Goal: one process is the timekeeper, try to keep the others synchronized to it
    - Each process C(i)'s clock is within a bound D of a well-known clock S external to the group
    - |C(i) – S| < D at all times
    - External clock S may be connected to UTC (Universal Coordinated Time) or an atomic clock
  - E.g., NTP, Cristian's algorithm

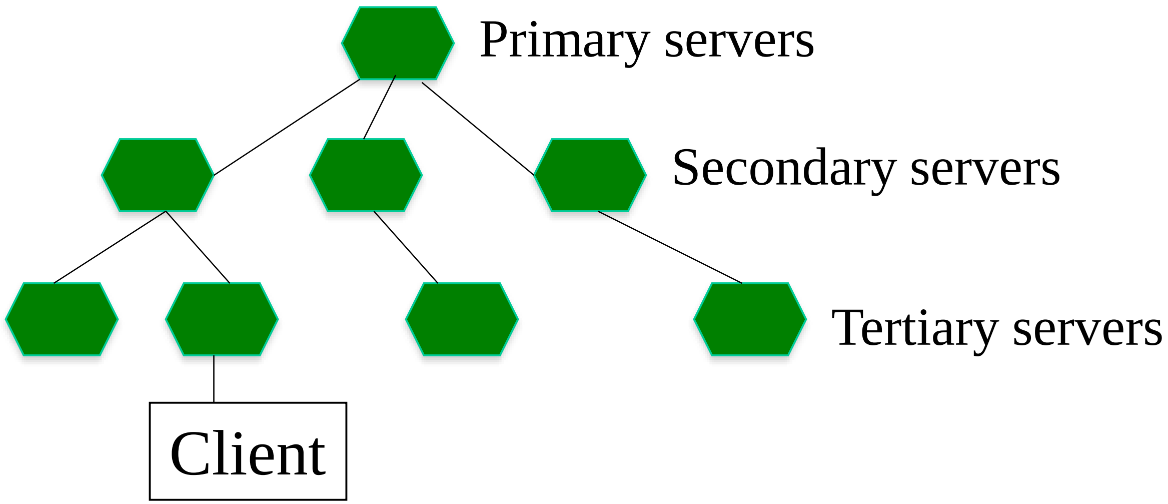
# The Berkeley Algorithm



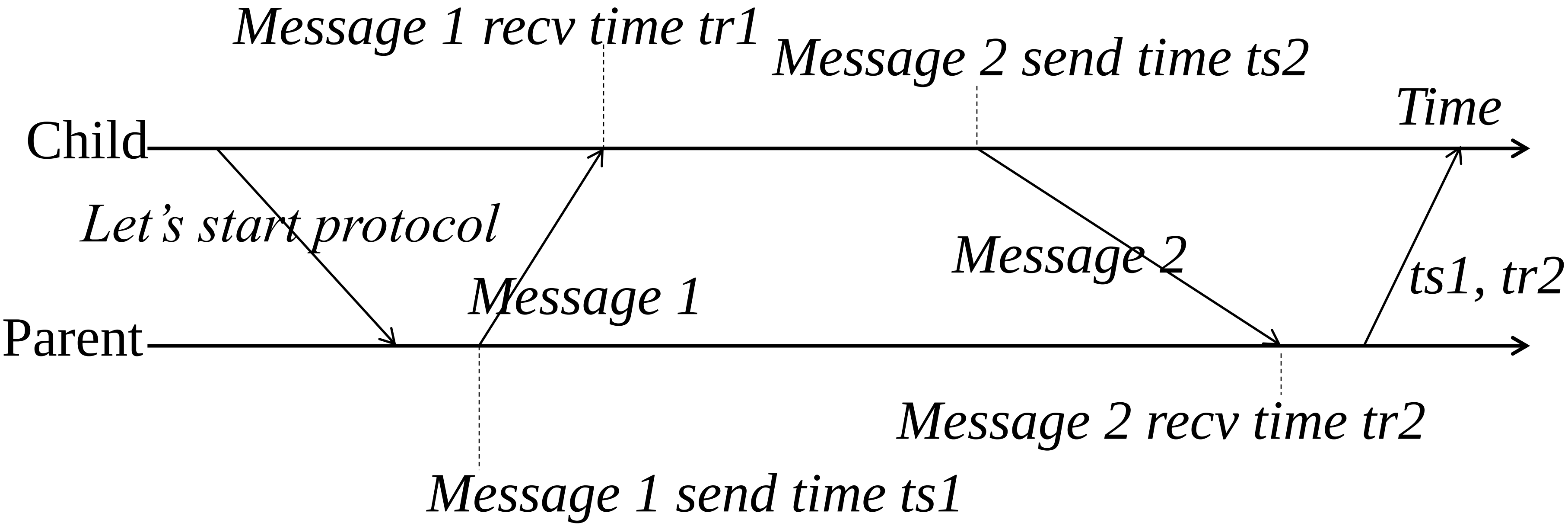
- (a) The time daemon asks all the other machines for their clock values.
- (b) The machines answer.
- (c) The time daemon tells everyone how to adjust their clock.

# 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



# What the Child Does

- Child calculates *offset* between its clock and parent's clock
- Uses $ts1$, $tr1$, $ts2$, $tr2$
- Offset is calculated as
  - $o = (tr1 - tr2 + ts2 - ts1)/2$

# Why o = (tr1 - tr2 + ts2 - ts1)/2?

- **Offset o = $(tr1 - tr2 + ts2 - ts1)/2$**
- **Let's calculate the error**
- **Suppose real offset is *oreal***
  - Child is ahead of parent by *oreal*
  - Parent is ahead of child by -*oreal*
- **Suppose one-way latency of Message 1 is *L1* (*L2* for Message 2)**
- **No one knows *L1* or *L2*!**
- **Then**

$$tr1 = ts1 + L1 + oreal$$

$$tr2 = ts2 + L2 - oreal$$

# Why o = (tr1 - tr2 + ts2 - ts1)/2? (2)

- **Then**

  $tr1 = ts1 + L1 + oreal$

  $tr2 = ts2 + L2 - oreal$

- **Subtracting second equation from the first**

  $oreal = (tr1 - tr2 + ts2 - ts1)/2 + (L2 - L1)/2$

  $\Rightarrow oreal = o + (L2 - L1)/2$

  $\Rightarrow |oreal - o| = |(L2 - L1)/2| < |(L2 + L1)/2|$

  - Thus, the error is bounded by the round-trip-time

# And so…

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

# Ordering Events in a Distributed System

- Often apps need to agree on the **order** in which events occur
- 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

# Lamport’s algorithm (1978)

- Key idea: synchronization need not be based on time (real or virtual)
  - For *make,* what counts is whether *input*.c is older or newer than *input*.o, not their absolute modification times
- Often apps need only agree on the **order** in which events occur
- Lamport’s algorithm synchronizes **logical clocks**
  - Used in almost all distributed systems since then

# Lamport's Logical Clocks (1)

- Define a logical relation "**happens-before**" among pairs of events
- The **"happens-before"** relation (denoted as → ) can be observed directly in two situations:
  - **If *a* and *b* are events in the same process, and *a* occurs before *b*,** then *a* → *b* is true.
  - **If a is the event of a message being sent by one process, and *b* is the event of the message being received by another process,** then *a* → *b*.
- Happens-before is transitive
  - If *a* → *b* and *b* → c then *a* → c
- Creates a *partial order* among events
  - Not all events related to each other via →

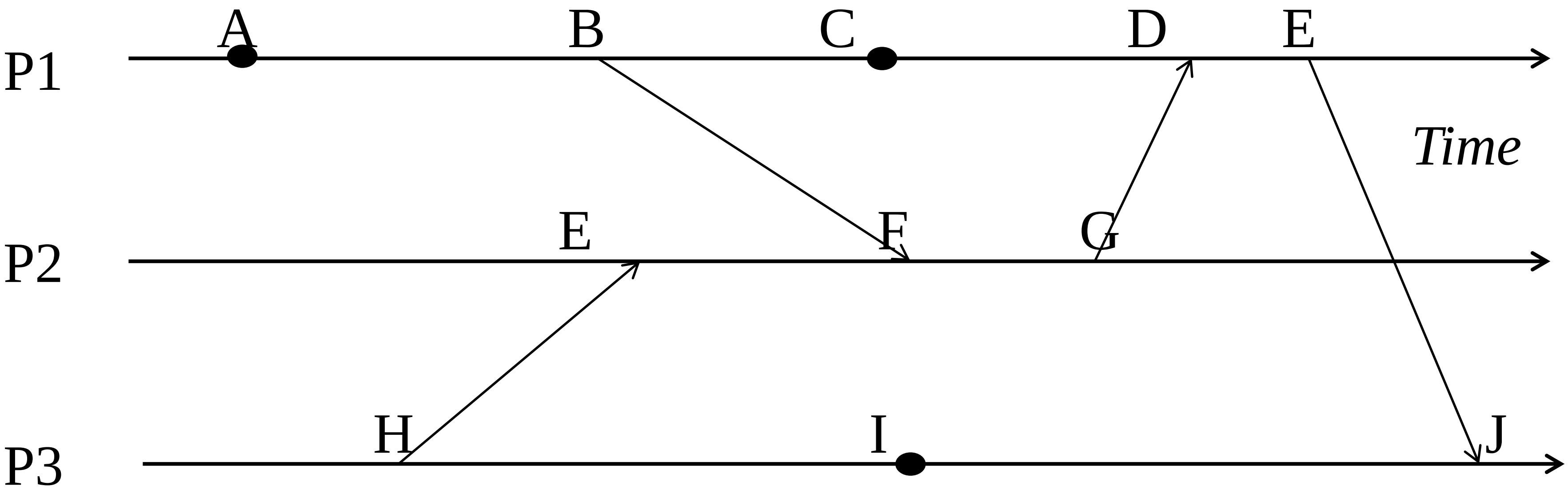
# Lamport's Logical Clocks (2)

- If events x and y **happen in different processes that do not exchange messages** (not even indirectly via third parties), then

  **x → y NOT true**

  - **x and y are concurrent**
  - Nothing can be said (or need be said) about when the events happened or which one happened first

# Example

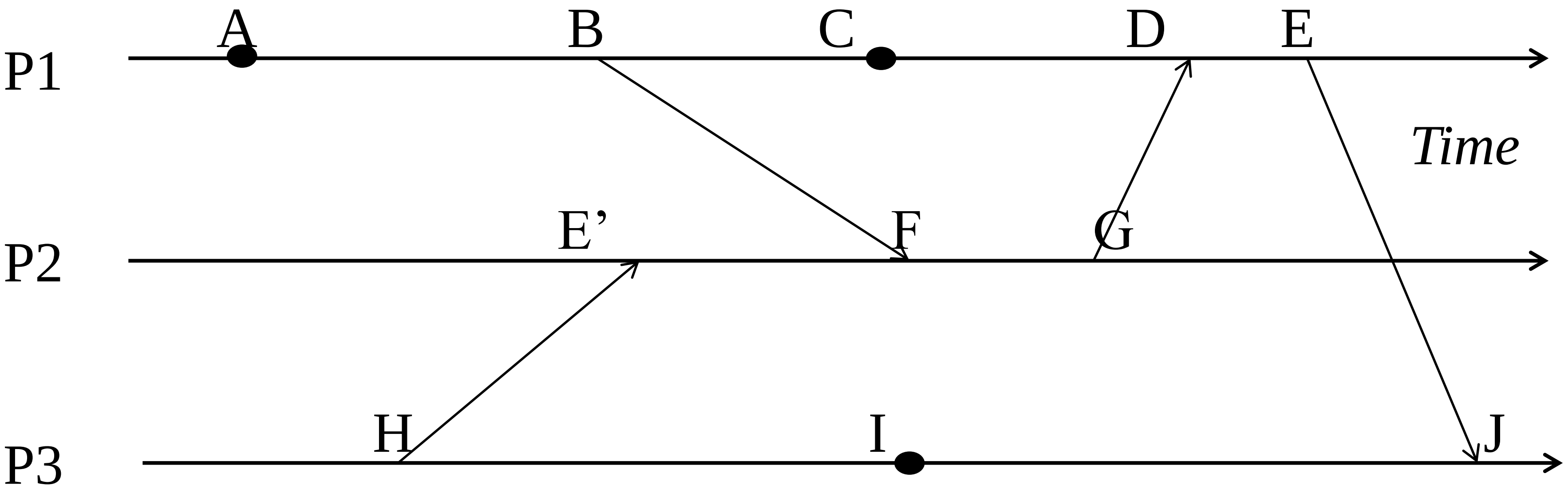


While P1 and P2 each have an event labeled E, these are different events as they occur at different processes.

● *Instruction or step*

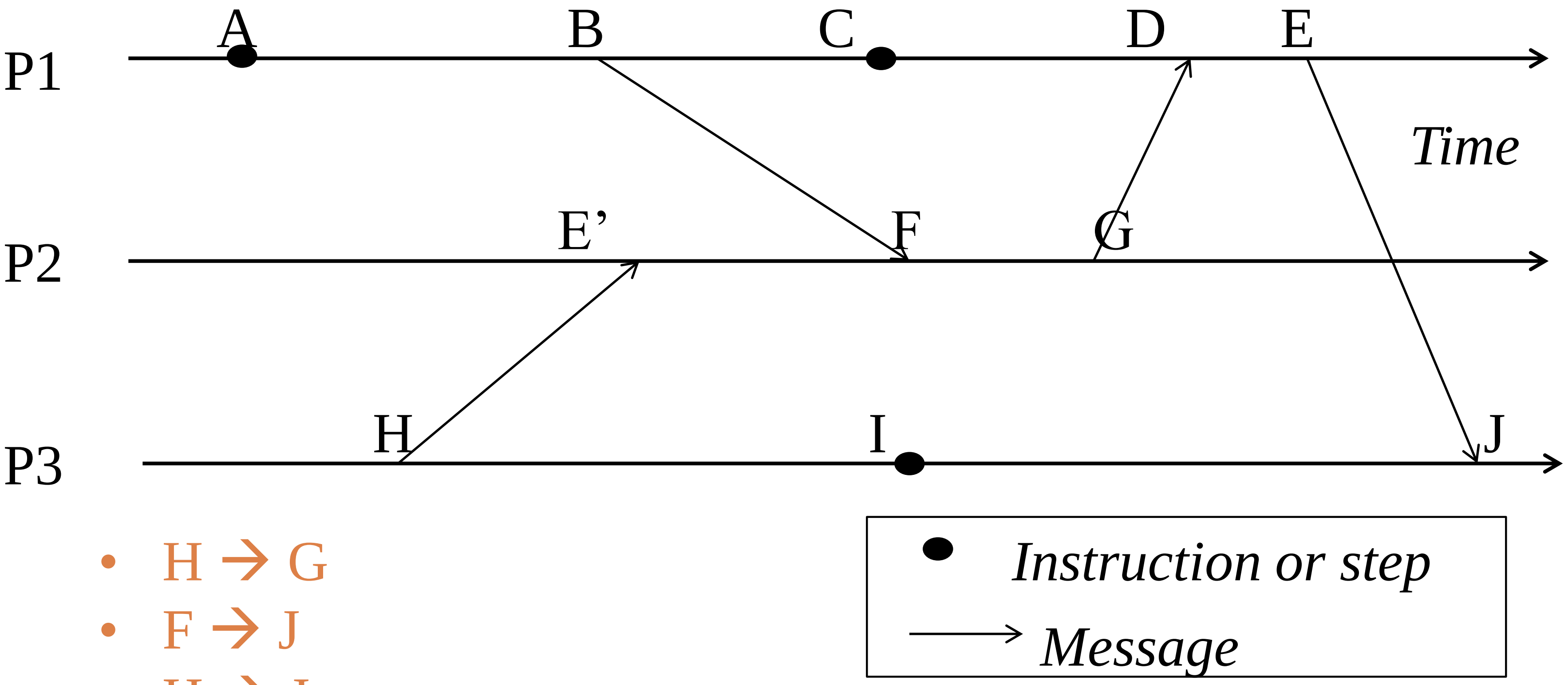
→ *Message*

# Happens-Before



- A → B
- B → F
- A → F

# Happens-Before (2)

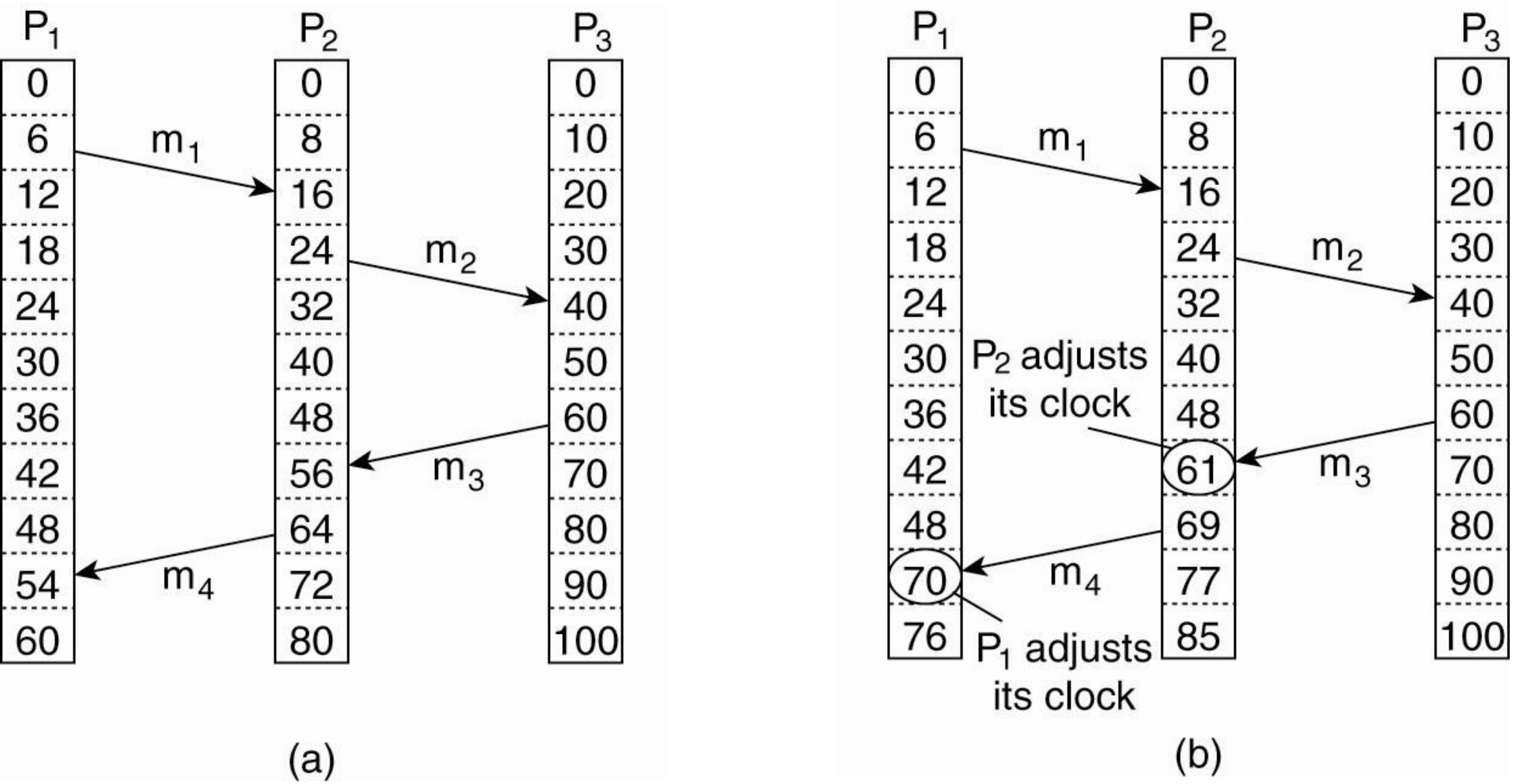


- H → G
- F → J
- H → J
- C → J

# Lamport's Logical Clocks (3)

- For every event a, we can assign it a (logical) time value C(a) on which all processes agree.
- such that:
  - If a->b then C(a) < C(b)
  - Clock time C must always go forward, never decrease
- Lamport's algorithm assigns logical times to events while respecting these properties

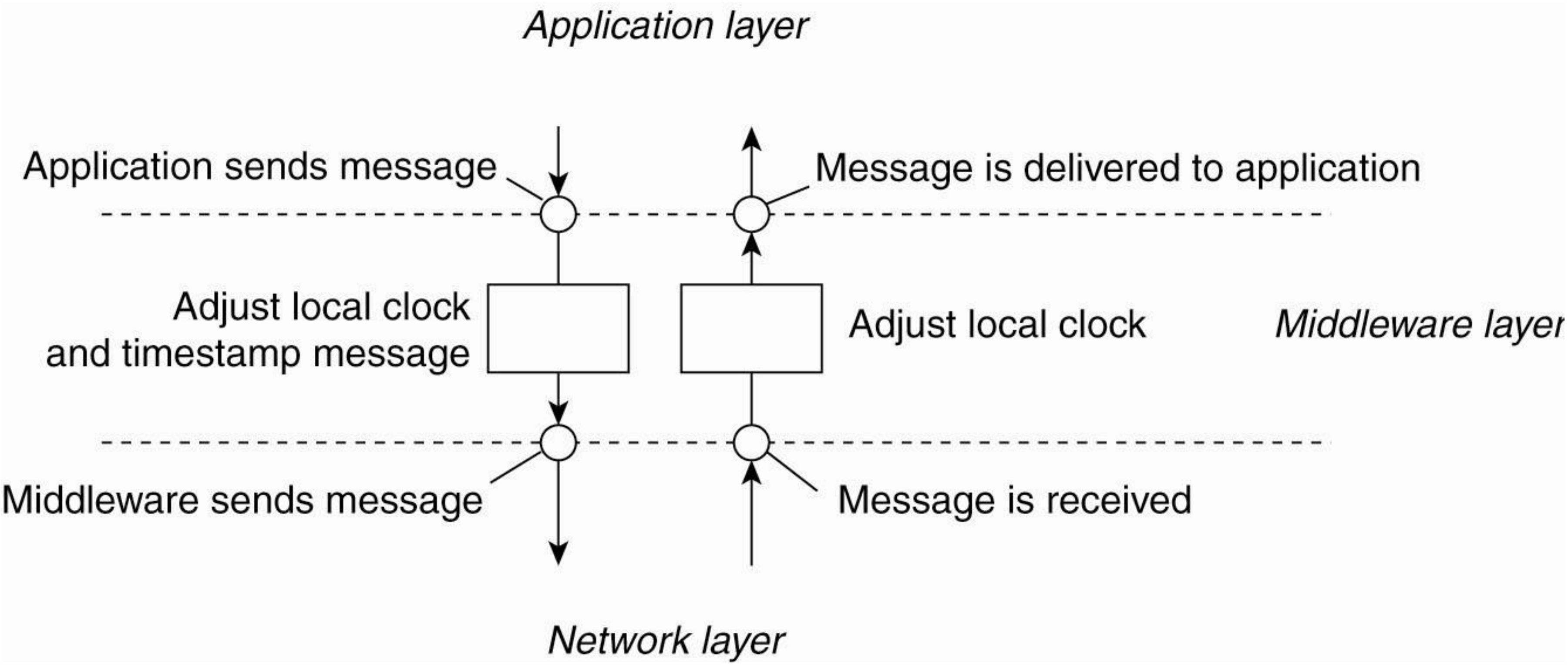
# Lamport's Logical Clocks (4)



(a) (b)

- (a) Three processes, each with its own clock. The clocks run at different rates. (b) Lamport's algorithm corrects the clocks.

# Lamport’s Logical Clocks (5)



- The positioning of Lamport’s logical clocks in distributed systems.

# Lamport's Logical Clocks (6)

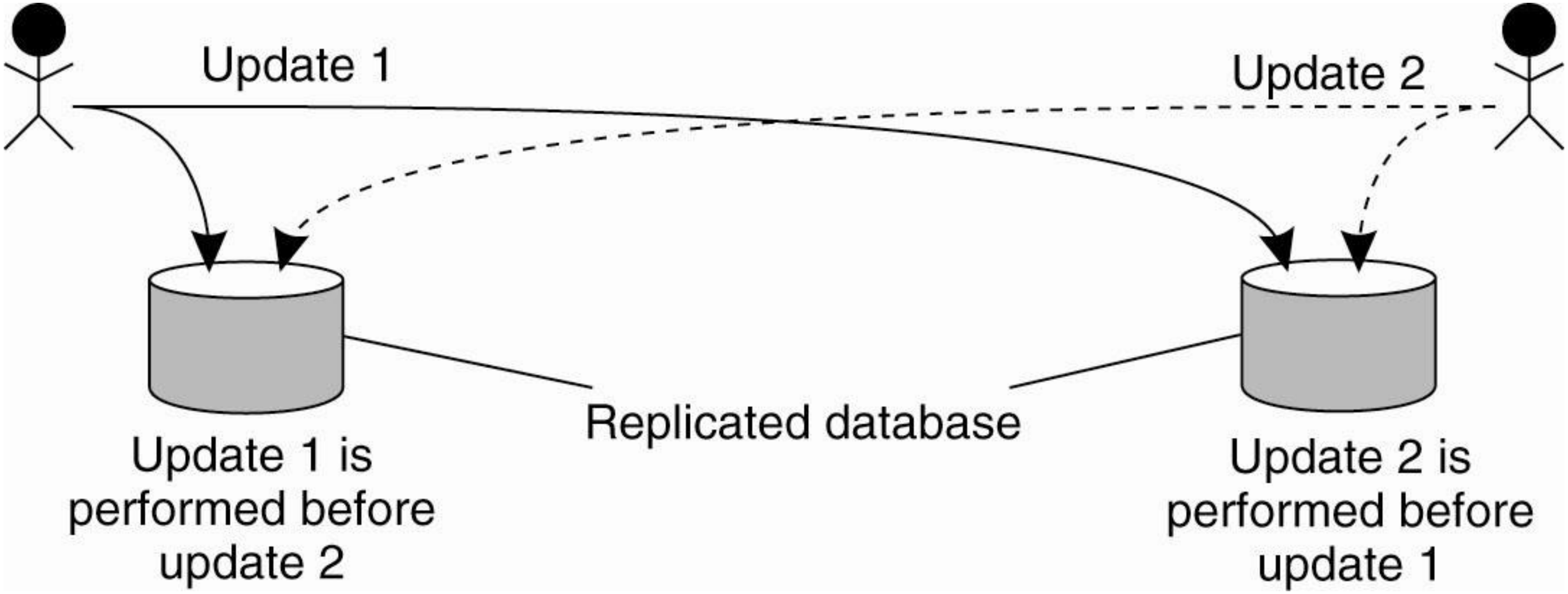
Each process $P_i$ maintains a *local* counter $C_i$

1. Before executing an event (e.g., send msg over net, deliver msg to app, or some internal event), $P_i$ executes $C_i \leftarrow C_i + 1$.
2. When process $P_i$ sends a message m to $P_z$, it sets *m*'s timestamp *ts*(*m*) equal to $C_i$.
3. Upon receipt of message *m*, process $P_z$ adjusts its own local counter as
   $C_z \leftarrow \max\{C_z , ts(m)\}, + 1,$ and delivers the message to the application.

# Lamport's Logical Clocks (7)

- We can attach the number/ID of the process in which the event occurs to the event's timestamp
  - E.g., event at time 40 at $P_i$ is timestamped with 40.i
- When we assign $C(a) = C_i(a)$, if *a* happened at process $P_i$ at time $C_i(a)$, we get a distributed implementation of the global time value of all events

# Example: Totally Ordered Multicasting



- Updating a replicated database and leaving it in an inconsistent state. Bank example: add $100 to account in SF copy while increasing with 1% interest the amount in NY copy
- Need **totally-ordered multicast:** all msgs delivered in same order to each node

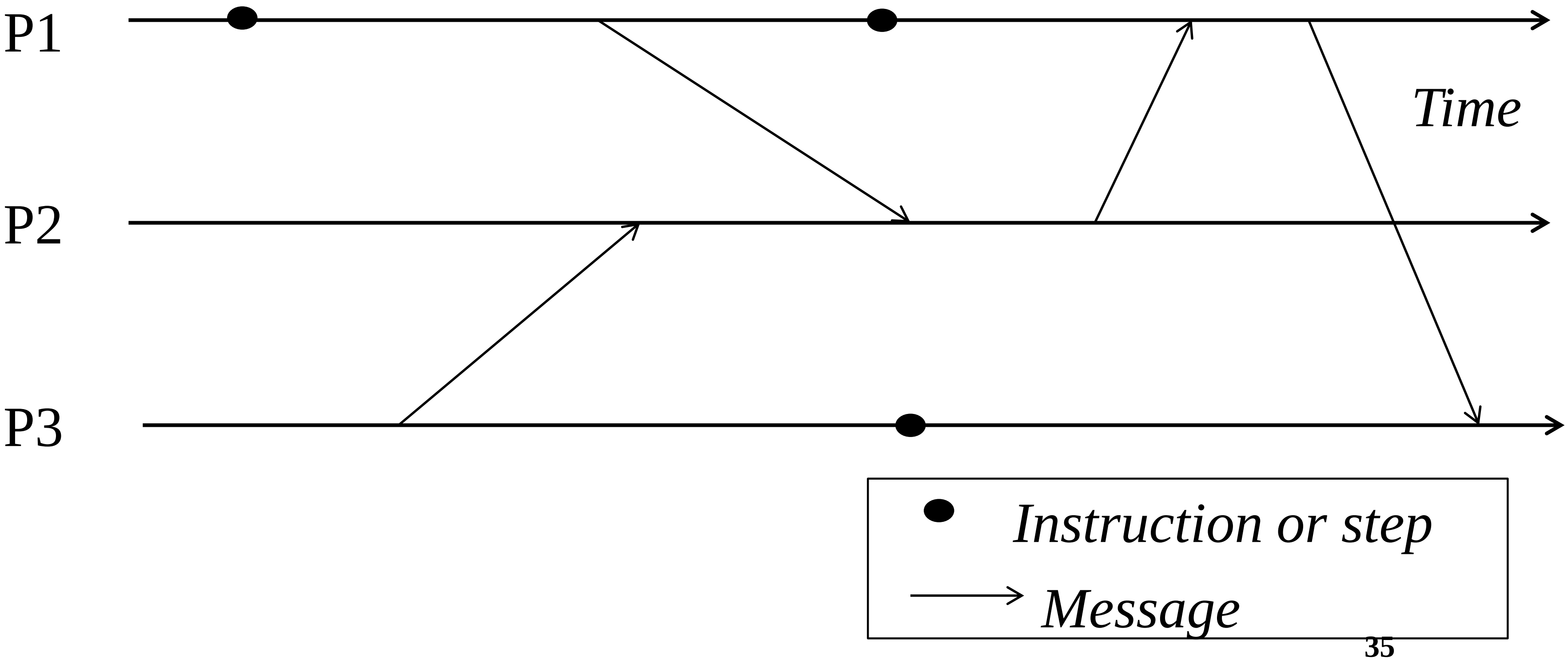
# Totally-ordered multicast

- Goal: all msgs delivered in same order to each node.
- Lamport's clocks can be used to implement totally-ordered multicast in a distributed fashion.
- When process receives msg, puts in local queue, ordered according to timestamp
- Receiver multicasts ack to other processes (Note: ack has higher timestamp than msg)
- **Eventually all processes will have the same copy of the local queue (provided no msgs are removed)**
  - A process delivers a queued msg to app only when msg is at head of queue and has been acknowledged by all others
  - Thus, all msgs are delivered in same order everywhere
- **Aka state machine replication**

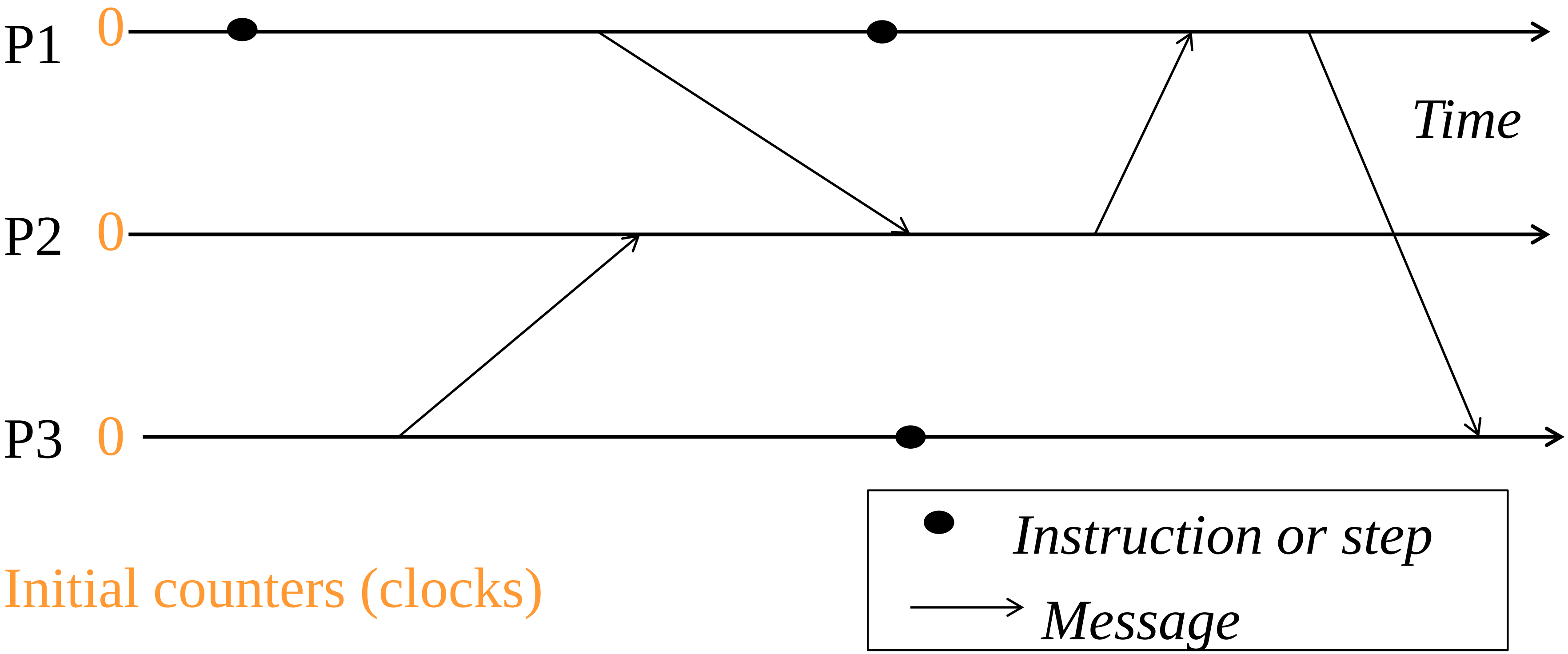
# With Lamport's Clocks…

- All events in a distributed system are totally ordered with property that
    - If *a* happened before *b*, then *a* will be positioned in that ordering before *b* (i.e., C(a) < C(b))
- However, converse not necessarily true
    - If C(a) < C(b), does not necessarily mean that *a* indeed happened before *b*
    - So we can't simply compare time values to determine if *a* happened before *b*

# Example: Lamport timestamps

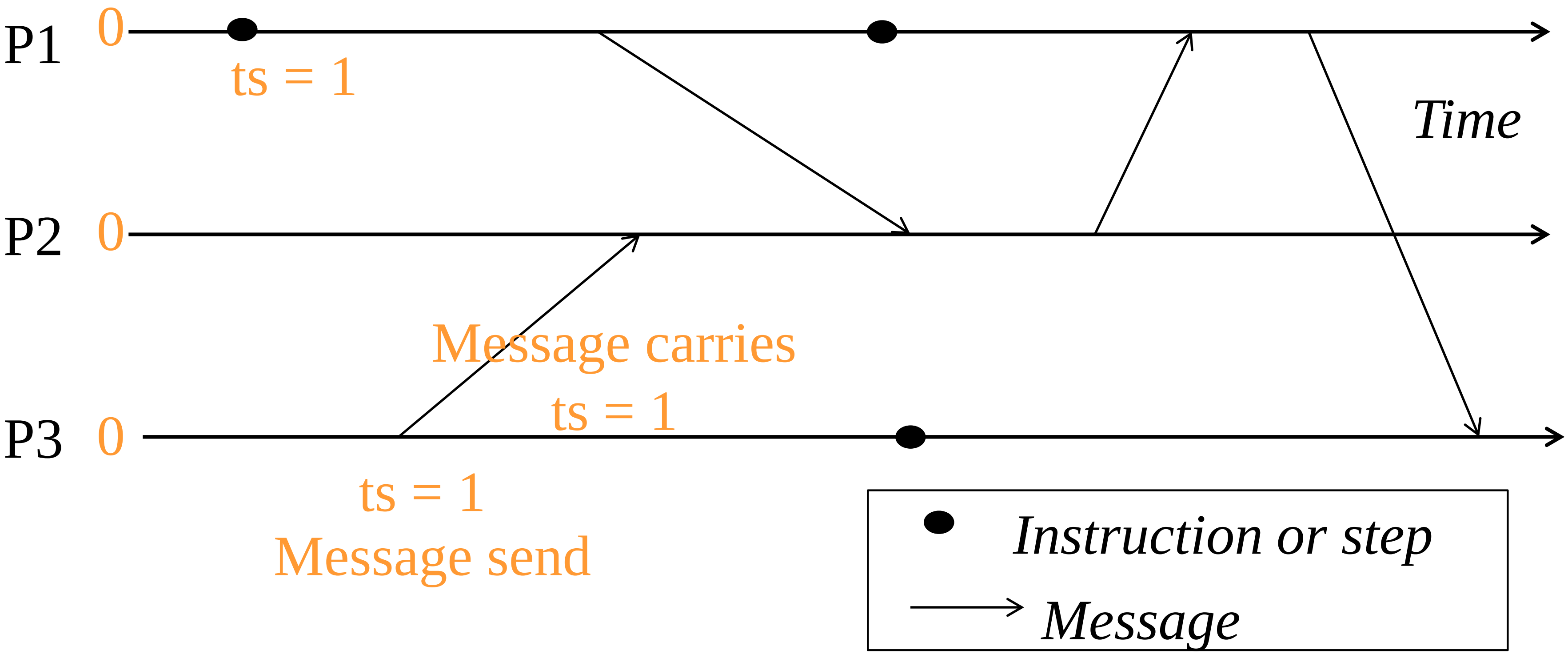


# Example: Lamport timestamps



Initial counters (clocks)

# Example Lamport timestamps



P1 0

ts = 1

*Time*

P2 0

Message carries
ts = 1

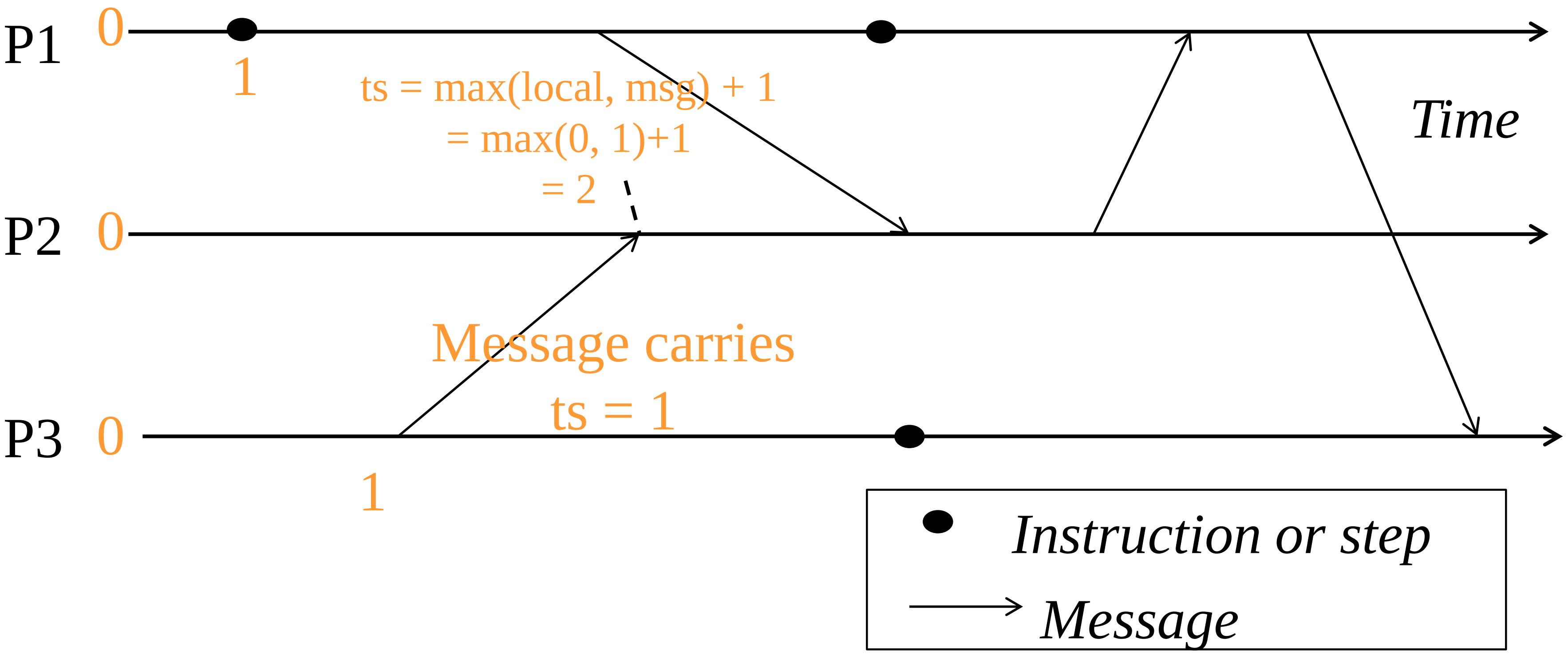
P3 0

ts = 1
Message send

● *Instruction or step*

→ *Message*

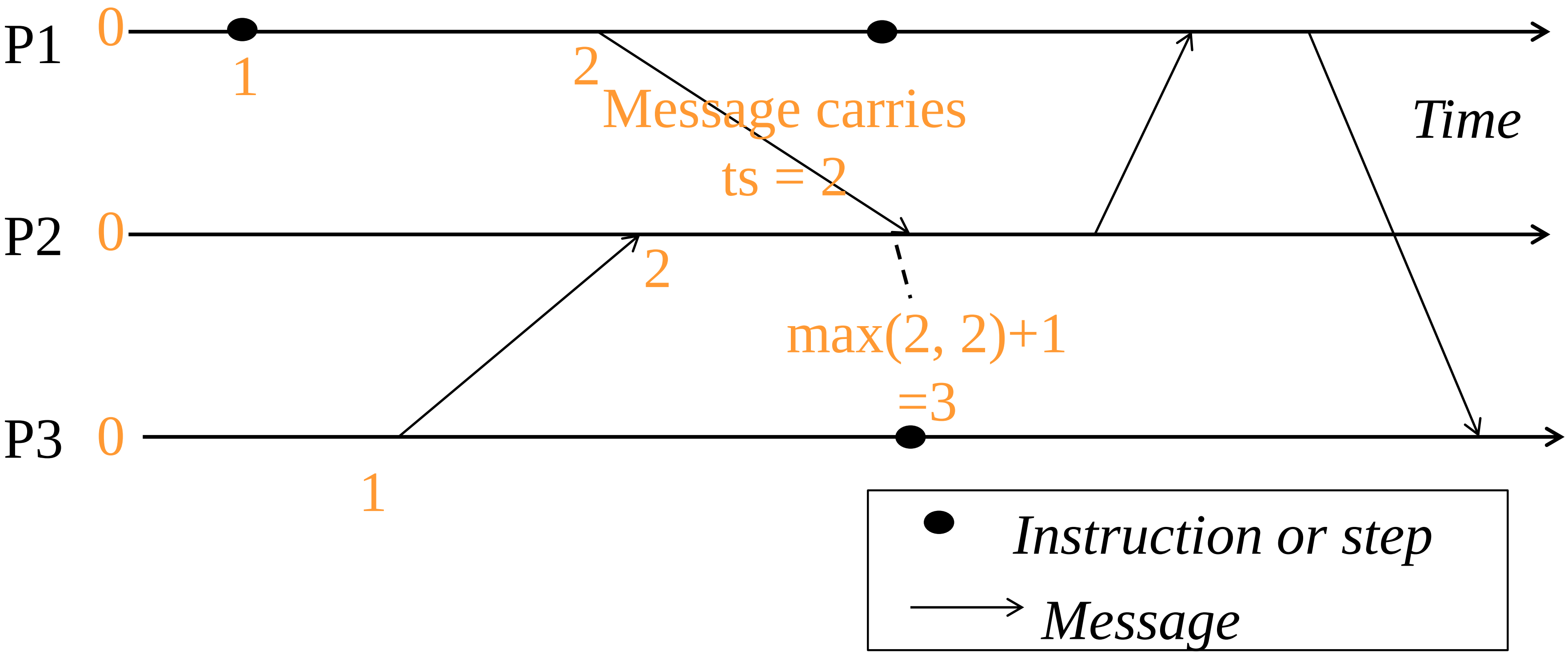
# Example Lamport timestamps



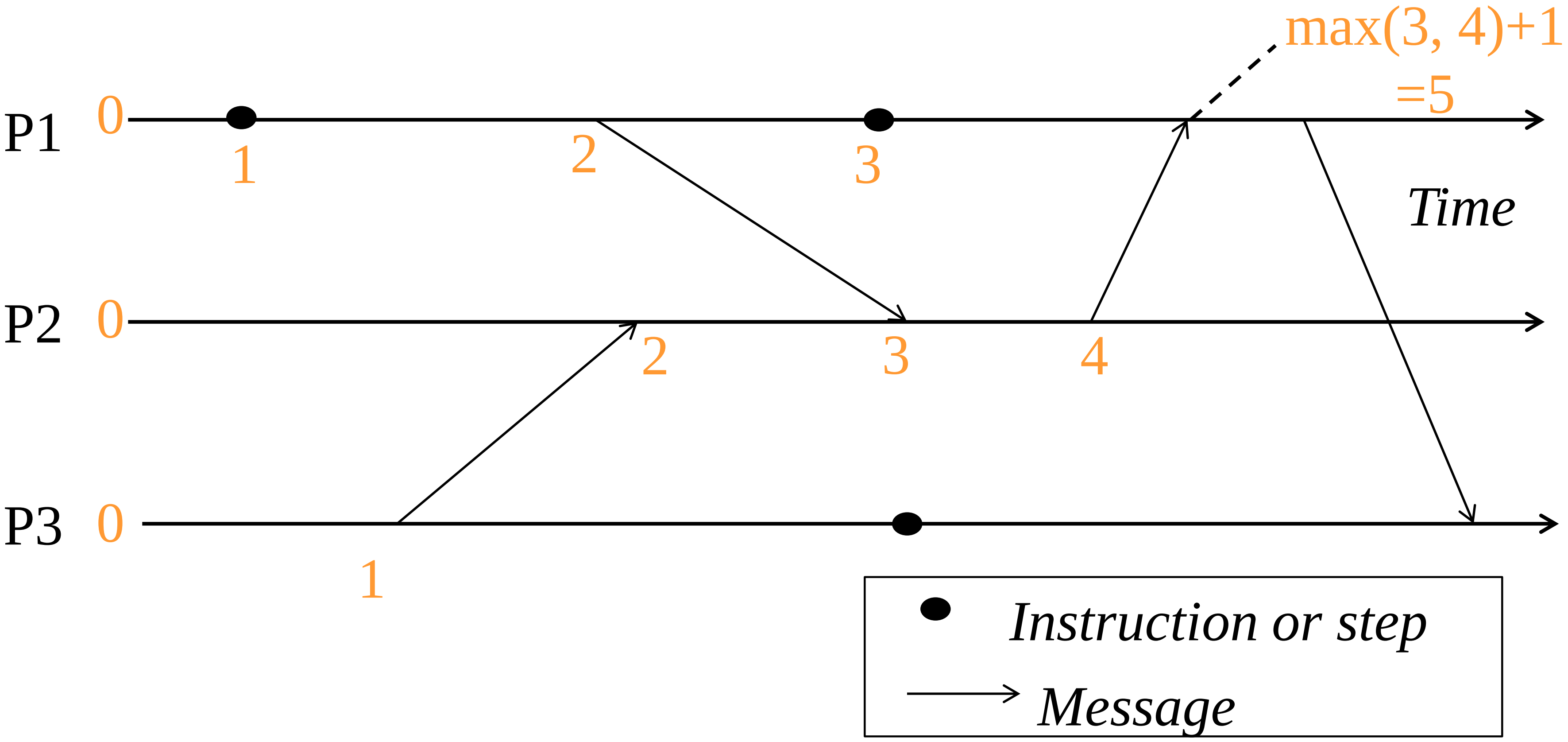
ts = max(local, msg) + 1
= max(0, 1)+1
= 2

Message carries
ts = 1

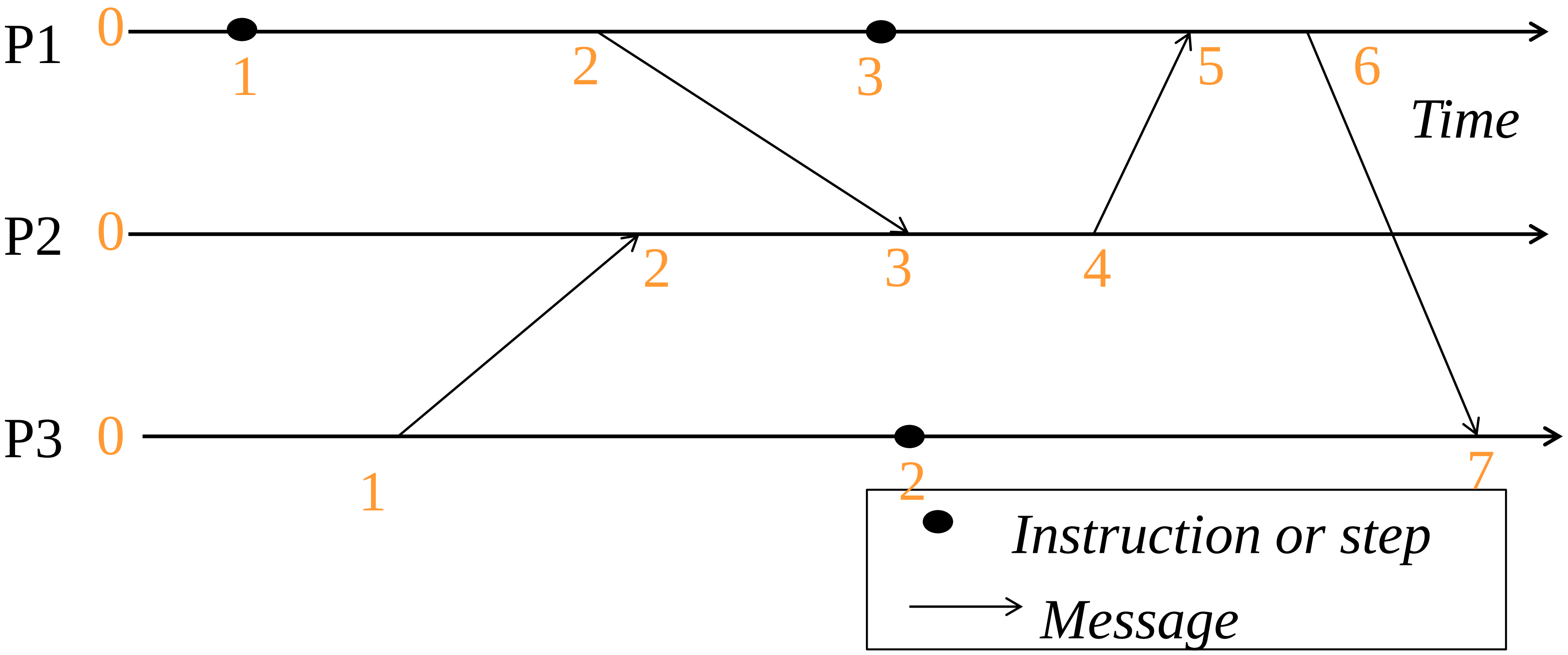
# Example: Lamport timestamps



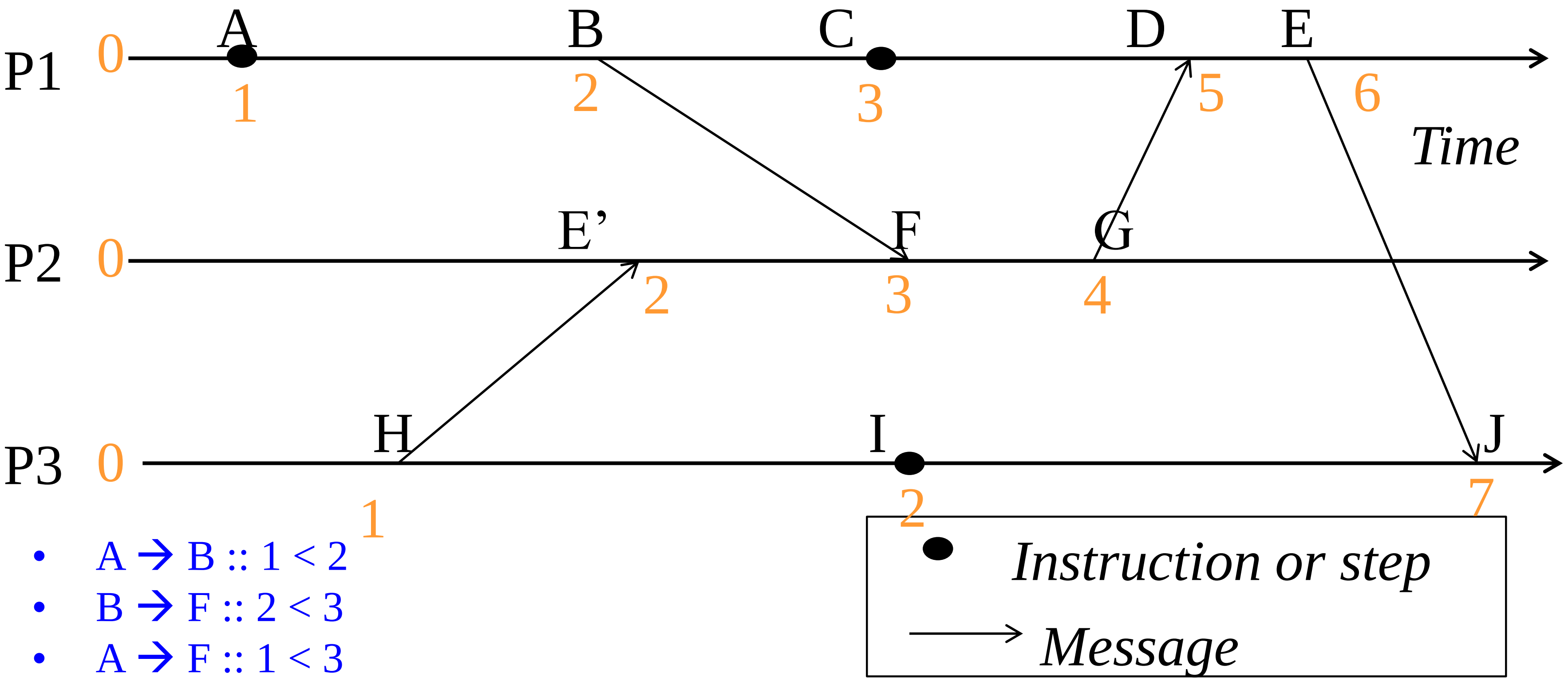
# Example: Lamport timestamps



# Example: Lamport timestamps

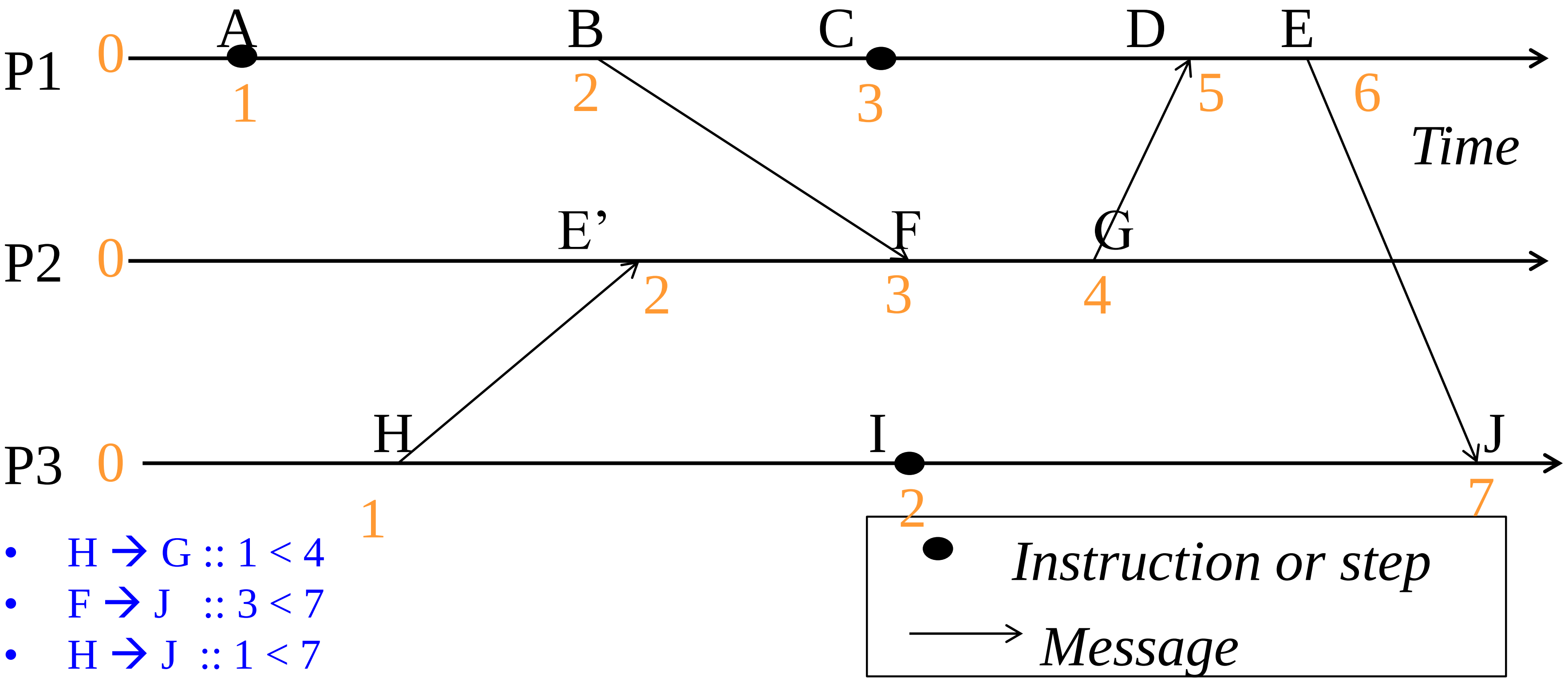


# Obeying Causality



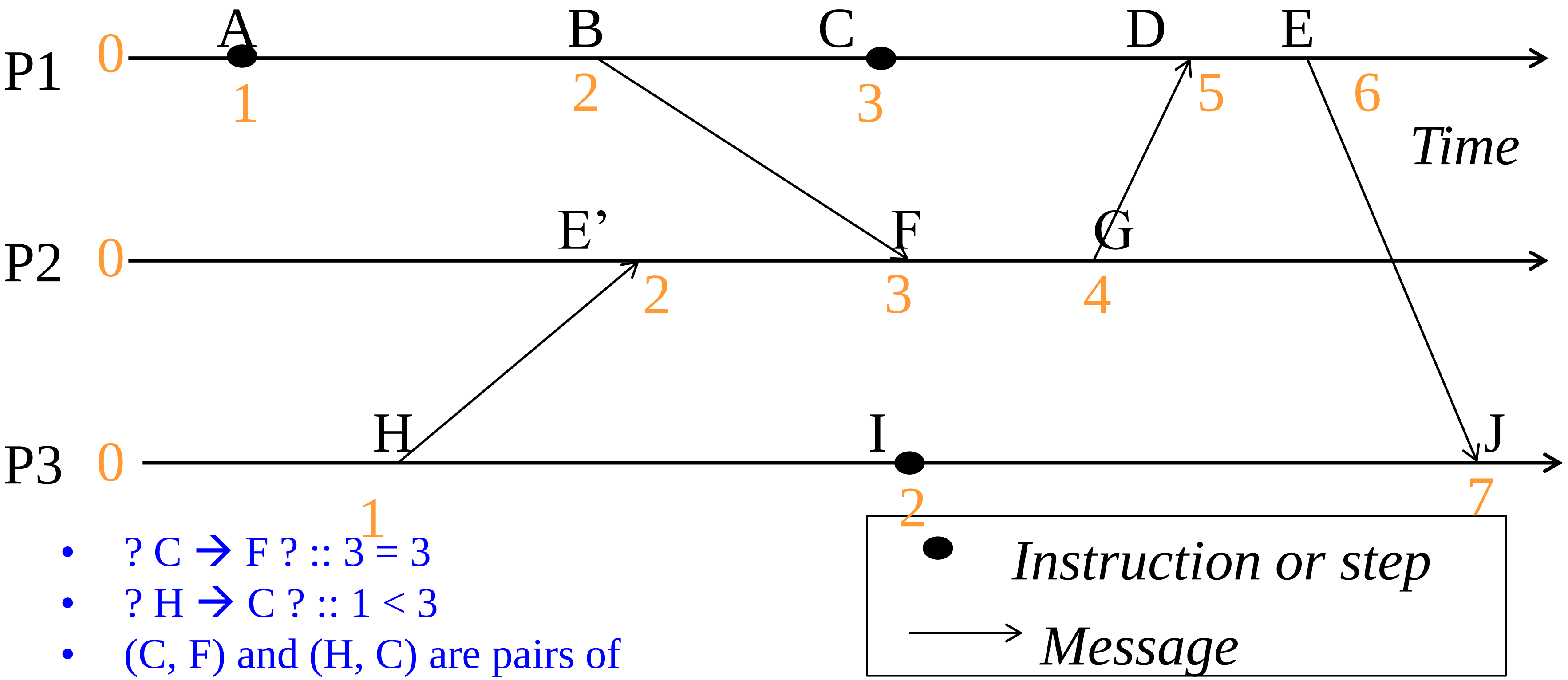
- A → B :: 1 < 2
- B → F :: 2 < 3
- A → F :: 1 < 3

# 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

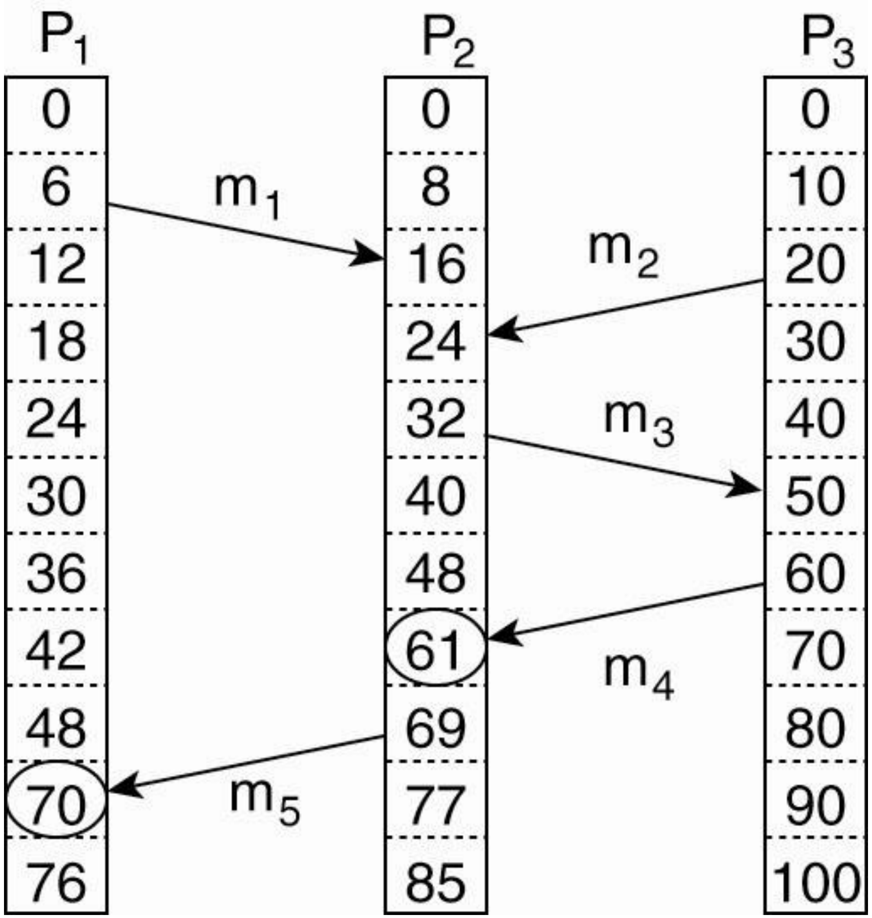
- **A pair of concurrent events doesn't have a causal path from one event to another (either way, in the pair)**
- **Lamport timestamps not guaranteed to be ordered or unequal for concurrent events**
- **Ok, since concurrent events are not causality related!**
- **Remember**

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

timestamp(E1) < timestamp (E2) ⇒

{E1 → E2} OR {E1 and E2 concurrent}

# Lamport clocks do not capture causality (Example 2)



- Concurrent message transmission using logical clocks. Note: Lamport's clocks do not capture **causality.**
- Sending m3 might depend on what was received through m1
- **S**ending of m2 (by P3) **definitely** has nothing to do with receipt of m1, so even though $T_{rcv}(m1) < T_{snd}(m2)$, can't be sure that m1 was indeed received before m2 was sent

# Next

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

# Vector Clocks (1)

- Causality can be captured by vector clocks
- Vector clocks are constructed by letting each process $P_i$ maintain a vector $VC_i$ with the following two properties:

1. $VC_i[i]$ is the number of events that have occurred so far at $P_i$. In other words, $VC_i[i]$ is the local logical clock at process $P_i$ .
2. If $VC_i[z] = k$ then $P_i$ knows that k events have occurred at $P_z$. It is thus $P_i$'s knowledge of the local time at $P_z$.

Property 1 attained by incrementing $VC_i[i]$ at every new event at process $P_i$.

# Vector Clocks (2)

- Steps carried out to accomplish property 2 of previous slide:

1. Before executing an instruction or send event $P_i$ executes $VC_i[i] \leftarrow VC_i[i] + 1$.
2. When process $P_i$ sends a message m to $P_z$, it sets *m*'s (vector) timestamp $ts(m)$ equal to $VC_i$.
3. Upon receipt of a message m, process $P_z$ adjusts its own vector by setting:
   $VC_z[k] \leftarrow \max\{VC_z[k], ts(m)[k]\}$ for k !=z
   $VC_z[k] \leftarrow VC_z[k] + 1$ for k=z .

# Vector Clocks (3)

- If event *a* has ts(a), then ts(a)[i]-1 = # events processed at $P_i$ that causally precede *a*
- When $P_z$ receives msg from $P_i$ with ts(m), it knows
  - # events that have occurred at $P_i$ that causally preceded the sending of m  AND
  - #events at *other* processes that took place before $P_i$ sent msg *m*
  - Hence, ts(m) tells $P_z$ the # events in other processes that preceded the sending of *m* and on which *m* may causally depend

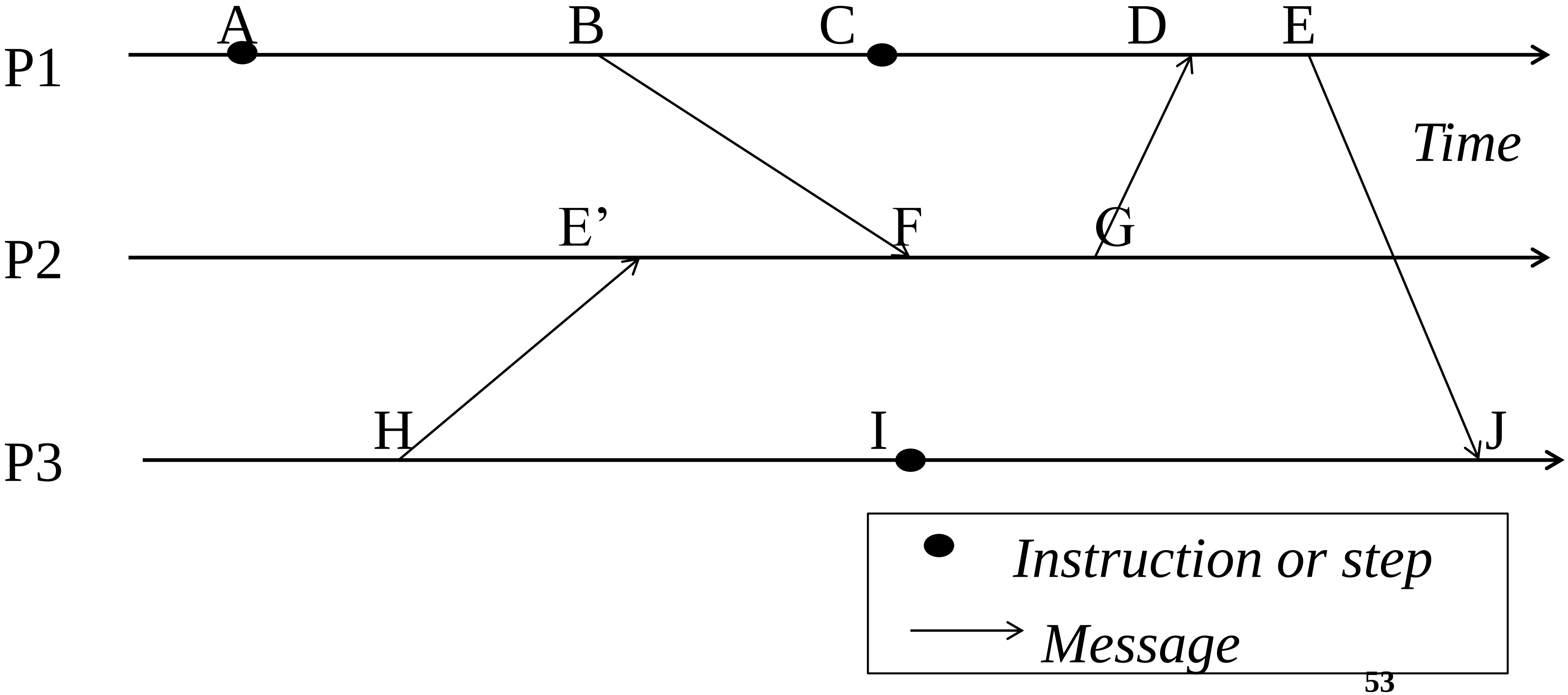
# Enforcing causal communication

- With vector clocks, we can ensure that a message is delivered only if all messages that causally precede it have also been received
- Assumptions
  - messages are multicast within a group of processes
  - Clocks are adjusted only when sending/receiving messages
- Causally-ordered multicasting is weaker than totally-ordered multicasting
  - If 2 messages unrelated, we do not care about the order they are delivered to apps

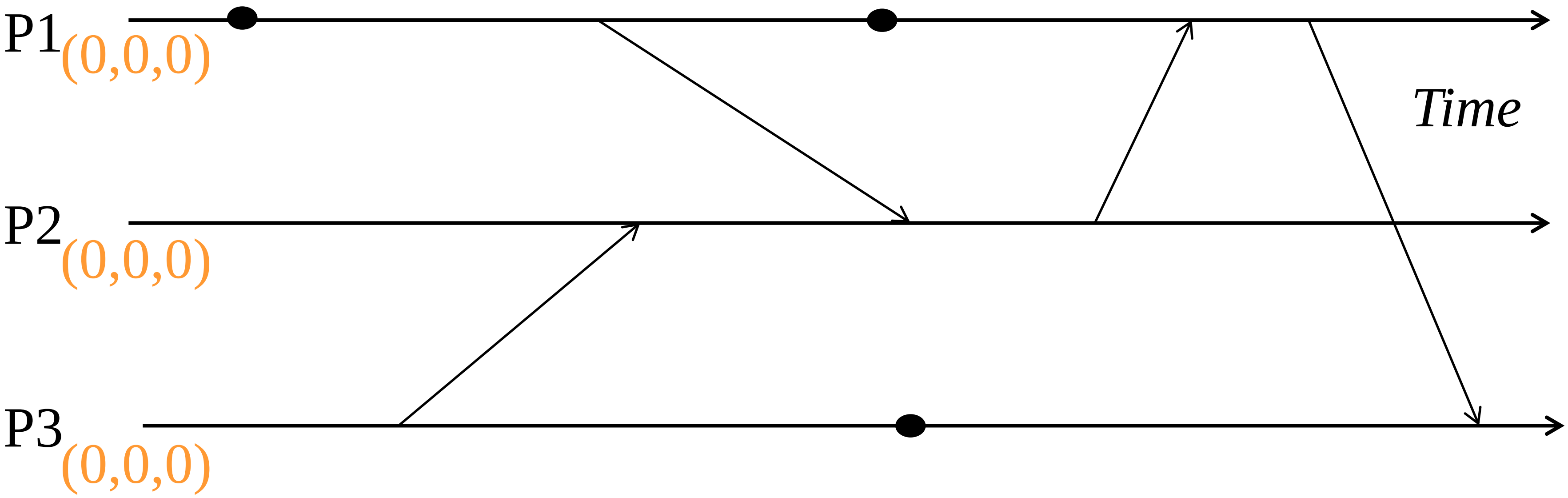
# Enforcing causal communication

- Suppose $P_z$ receives *m* from $P_i$ with (vector) timestamp ts(*m*)
- Delivery of message *m* to the application is delayed until following conditions are met:
  - ts(*m*)[i] = $VC_z$[i] + 1 [i.e., *m* is the next message that $P_z$ was expecting from process $P_i$]
  - for all k != i, ts(*m*)[k] <= $VC_z$[k] [i.e., $P_z$ has seen all the messages that have been seen by $P_i$ when it sent message *m*]

# Vector Timestamps Example



# Vector Timestamps Example



P1 (0,0,0)

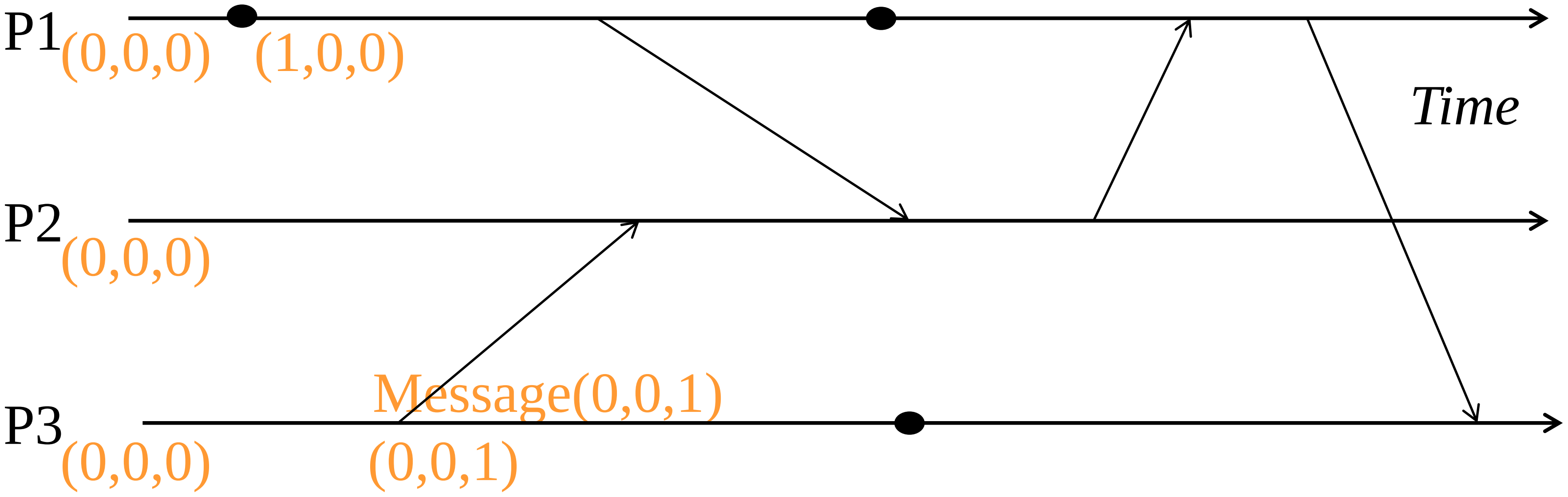
P2 (0,0,0)

P3 (0,0,0)

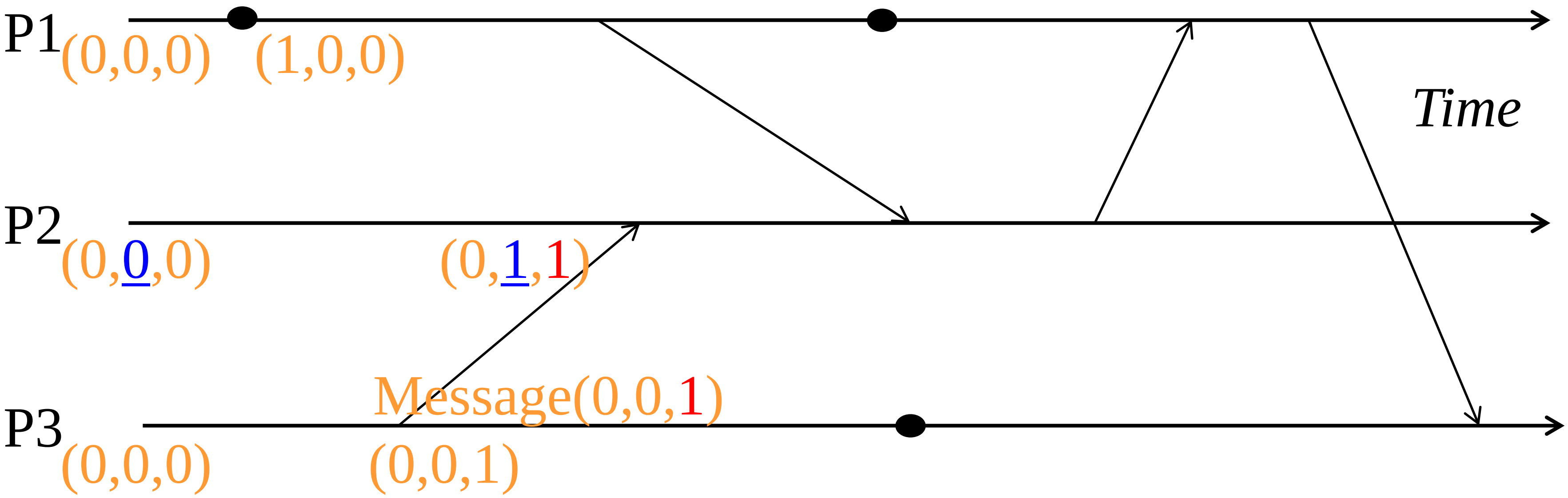
*Time*

Initial counters (clocks)

# Vector Timestamps Example



# Vector Timestamps Example



P1 (0,0,0) (1,0,0)

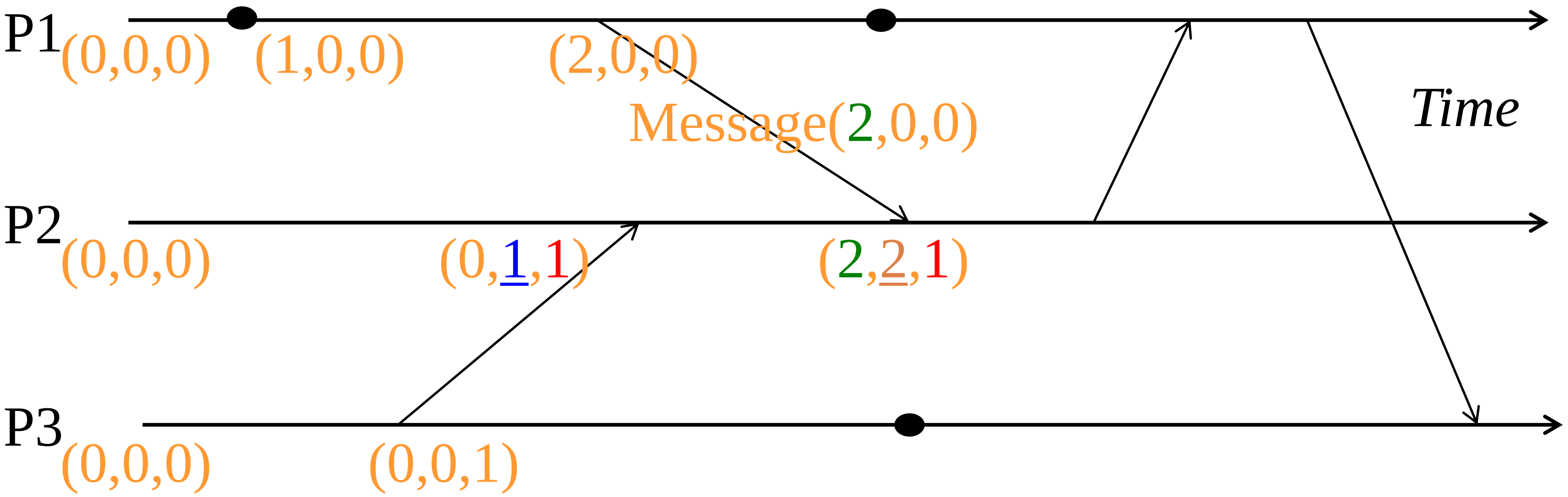
Time

P2 (0,0,0) (0,1,1)

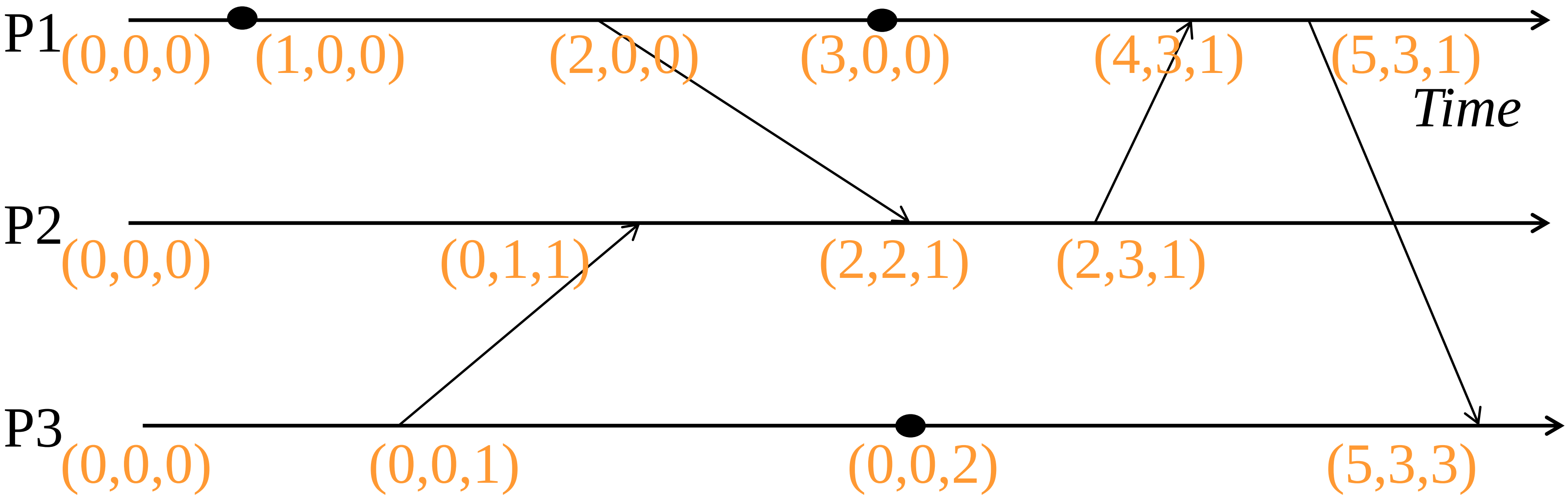
Message(0,0,1)

P3 (0,0,0) (0,0,1)

# Vector Timestamps Example



# Vector Timestamps Example



# 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

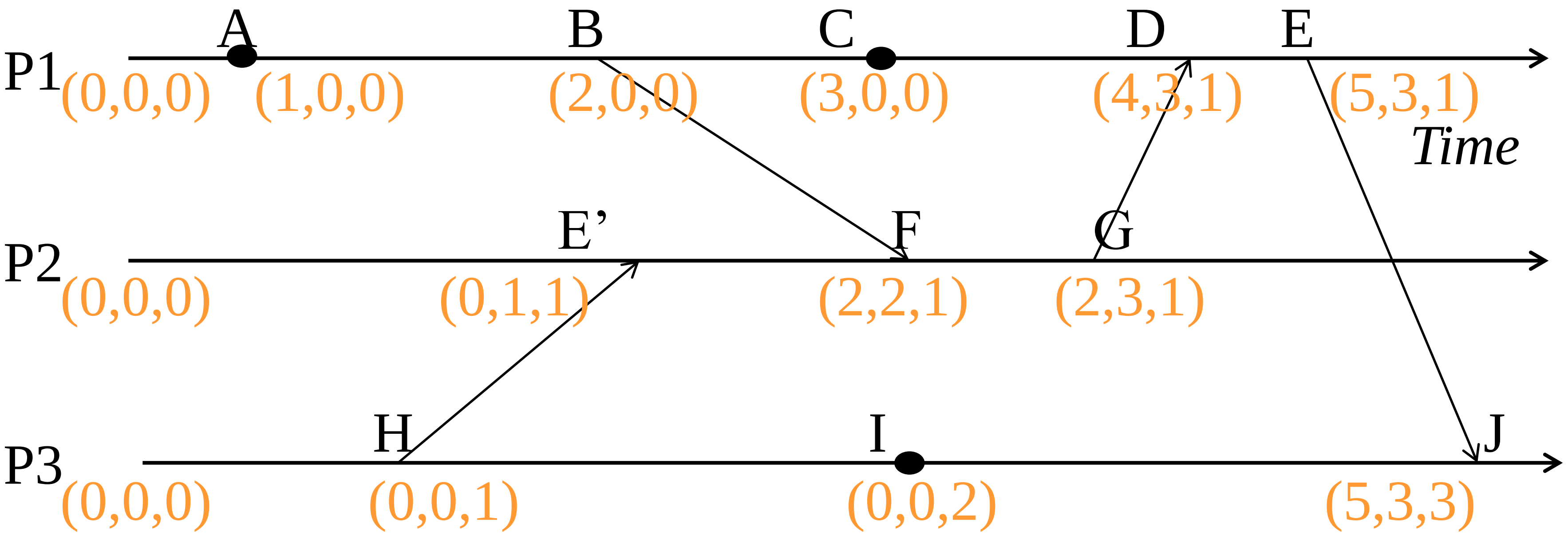
- Two events $VT_1$ and $VT_2$ are concurrent

  *iff*

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

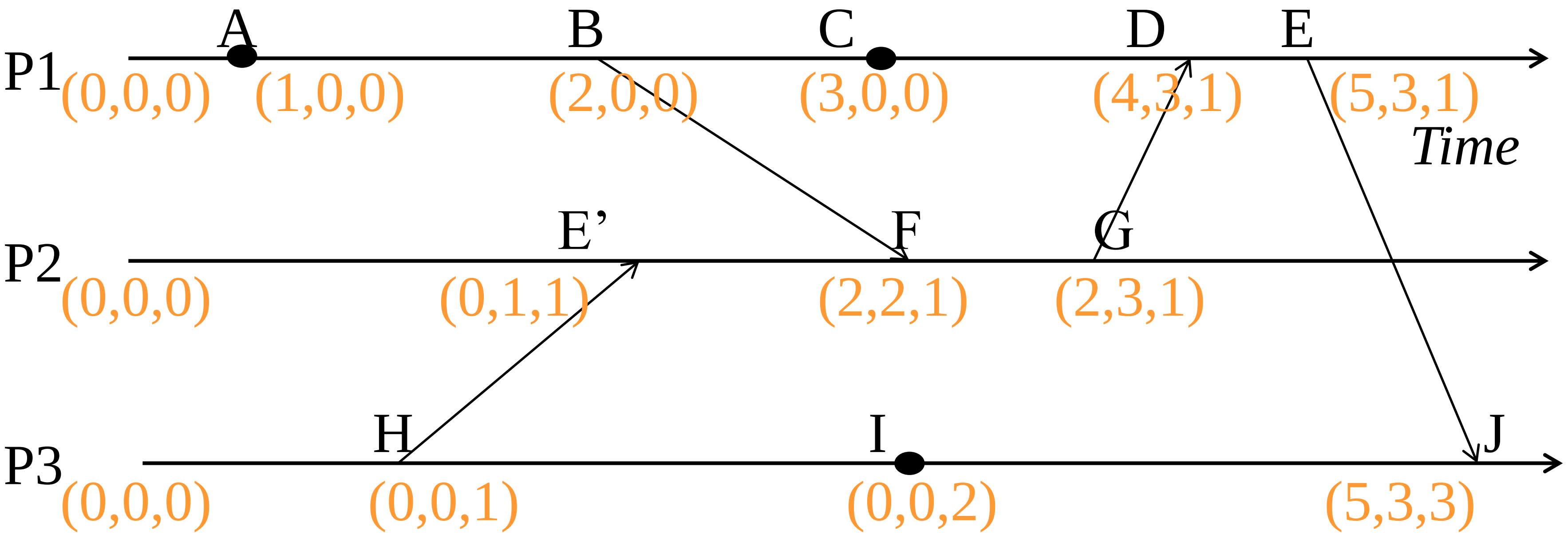
  We'll denote this as $VT_2 \mid\mid\mid VT_1$

# Obeying Causality



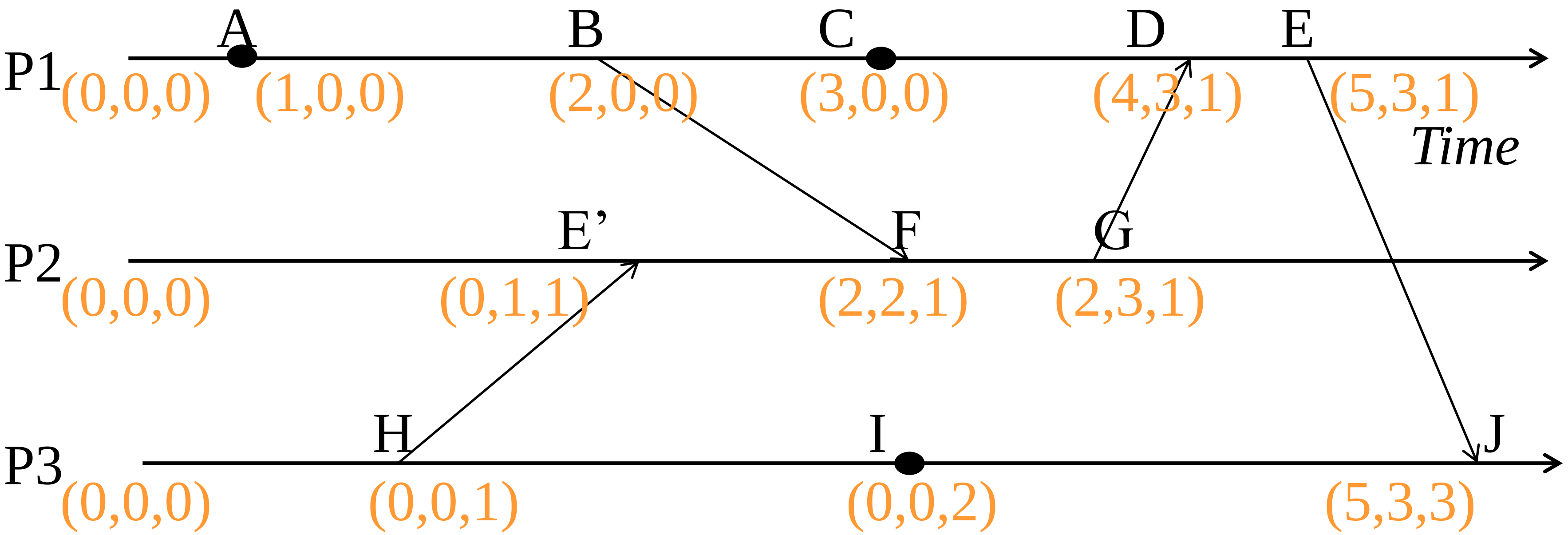
- A → B :: (1,0,0) < (2,0,0)
- B → F :: (2,0,0) < (2,2,1)
- A → F :: (1,0,0) < (2,2,1)

# 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

# CATOCS controversy

- CATOCS (Causal and Totally Ordered Communication Service) middleware toolkits are available
- Should support for causally and totally ordered multicasting be provided by middleware or should apps handle ordering of messages?
  - Middleware cannot tell what a message contains, so only *potential* causality is captured → overly restrictive
  - Middleware cannot catch all causality
    - Electronic bulletin board example – Bob posts response to Alice's article after having heard over phone about it from Alice
  - Again, some argue application knows best (E2E)

# Time and Ordering Summary

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

# Time and Ordering Summary (2)

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