Time and Clock Synchronization in Cloud Data Centers

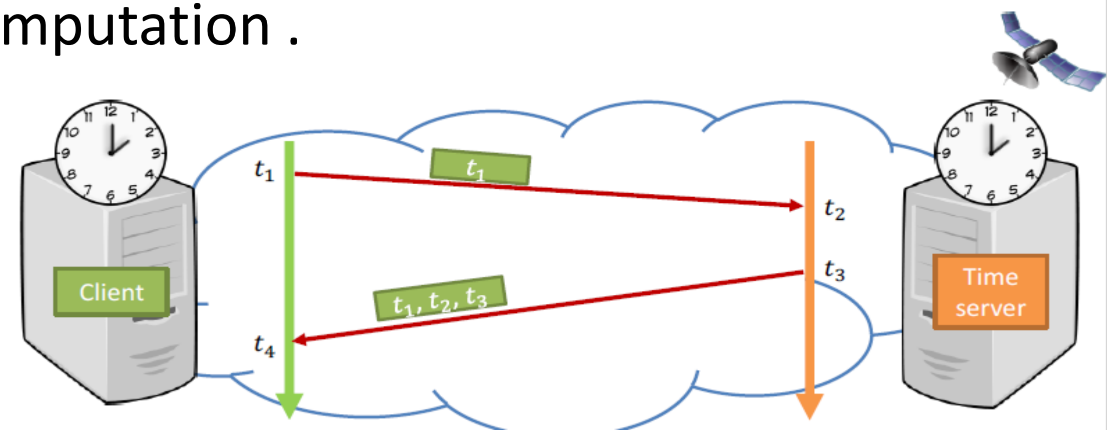
Dr. Rajiv Misra
Associate Professor
Dept. of Computer Science & Engg.
Indian Institute of Technology Patna
rajivm@iitp.ac.in

Preface

Content of this Lecture:

- In this lecture, we will discuss the fundamentals of clock synchronization in cloud and its different algorithms.

- We will also discuss the causality and a general framework of logical clocks and present two systems of logical time, namely, lamport and vector, timestamps to capture causality between events of a distributed computation .



Need of Synchronization

- **You want to catch a bus at 9.05 am, but your watch is off by 15 minutes**
  - What if your watch is Late by 15 minutes?
    - You'll miss the bus!
  - What if your watch is Fast by 15 minutes?
    - You'll end up unfairly waiting for a longer time than you intended

- **Time synchronization is required for:**
  - **Correctness**
  - **Fairness**

Time and Synchronization

- **Time and Synchronization**

  (“There’s is never enough time…”)

- **Distributed Time**
  - The notion of time is well defined (and measurable) at each single location
  - But the relationship between time at different locations is unclear

- **Time Synchronization is required for:**
  - **Correctness**
  - **Fairness**

Synchronization in the cloud

Example: Cloud based airline reservation system:

- Server X receives, a client request, to purchase the last ticket on a flight, say PQR 123.
- Server X timestamps the purchase using its local clock as **6h:25m:42.55s**. It then logs it. Replies ok to the client.
- That was the very last seat, Server X sends a message to Server Y saying the "flight is full".
- Y enters, "Flight PQR 123 is full" + its own local clock value, (which happens to read **6h:20m:20.21s**).
- Server Z, queries X's and Y's logs. Is confused that a client purchased a ticket at X after the flight became full at Y.
- **This may lead to full incorrect actions at Z**

Key Challenges

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

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 large 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.

Space-time diagram

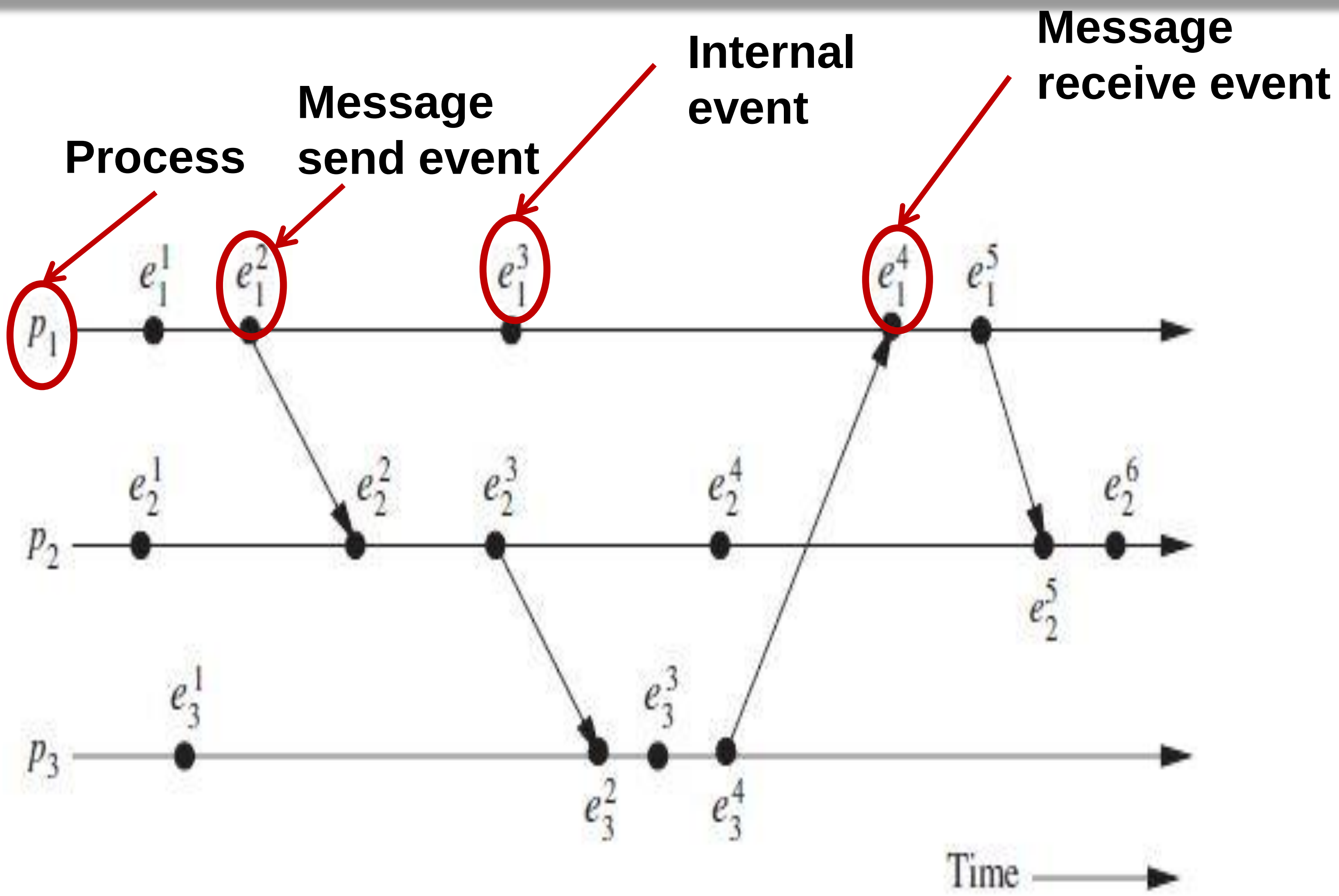


Figure : The space-time diagram of a distributed execution.

Clock Skew vs. 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 road.
  - **Clock Drift = Relative difference in clock frequencies (rates) of two processes**
    - Like difference in speeds of two vehicles on the road.
- **A non-zero clock skew implies clocks are not synchronized**
- **A non-zero clock drift causes skew increases (eventually).**
  - If faster vehicle is ahead, it will drift away.
  - If faster vehicle is behind, it will catch up and then drift away.

Clock Inaccuracies

- Clocks that must not only be synchronized with each other but also have to adhere to physical time are termed **physical clocks**.

- Physical clocks are synchronized to an accurate real-time standard like **UTC (Universal Coordinated Time).**

- However, due to the clock inaccuracy, **a timer (clock)** is said to be working within its specification if (where **constant ρ** is the **maximum skew rate** specified by the manufacturer)

$$1 - \rho \leq \frac{dc}{dt} \leq 1 + \rho$$

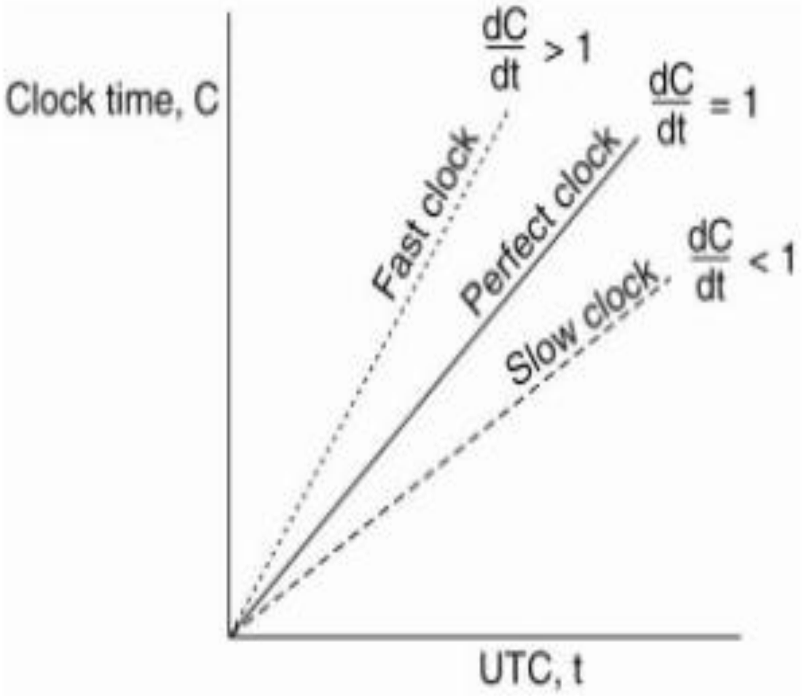


Figure: The behavior of fast, slow, and perfect clocks with respect to UTC.

How often to Synchronize

- **Maximum Drift rate (MDR) of a clock**
- Absolute MDR is defined to relative coordinated universal Time (UTC). UTC is the correct time at any point of time.
  - MDR of any process depends on the environment.
- Maximum drift rate between two clocks with similar MDR is **2*MDR**.
- Given a maximum acceptable skew M between any pair of clocks, need to synchronize at least once every:
  M/ (2* MDR) time units.
  - Since time = Distance/ Speed.

External vs Internal Synchronization

- Consider a group of processes
- **External synchronization**
    - Each process C(i)’s clock is within a bounded D of a well-known clock S external to the group
    - |C(i)- S|< D at all times.
    - External clock may be connected to UTC (Universal Coordinated Time) or an atomic clock.
    - **Example: Christian’s algorithm, NTP**
- **Internal Synchronization**
    - Every pair of processes in group have clocks within bound D
    - |C(i)- C(j)|< D at all times and for all processes i,j.
    - **Example: Berkley Algorithm, DTP**

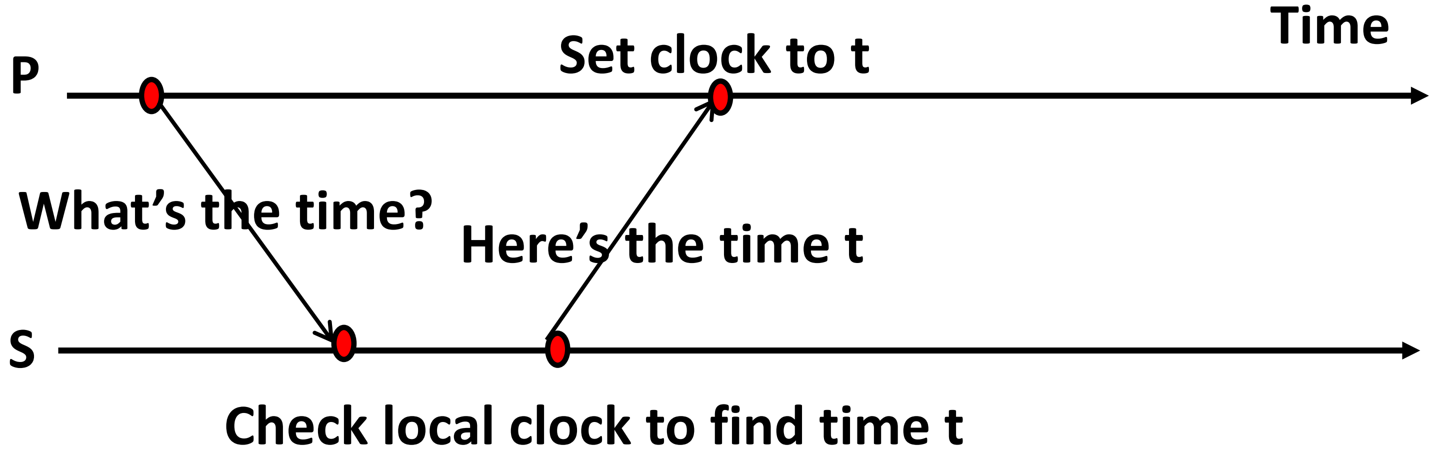
External vs Internal Synchronization

- **External synchronization with D => Internal synchronization with 2*D.**

- **Internal synchronization does not imply External Synchronization.**
  - In fact, the entire system may drift away from the external clock S!

Basic Fundamentals

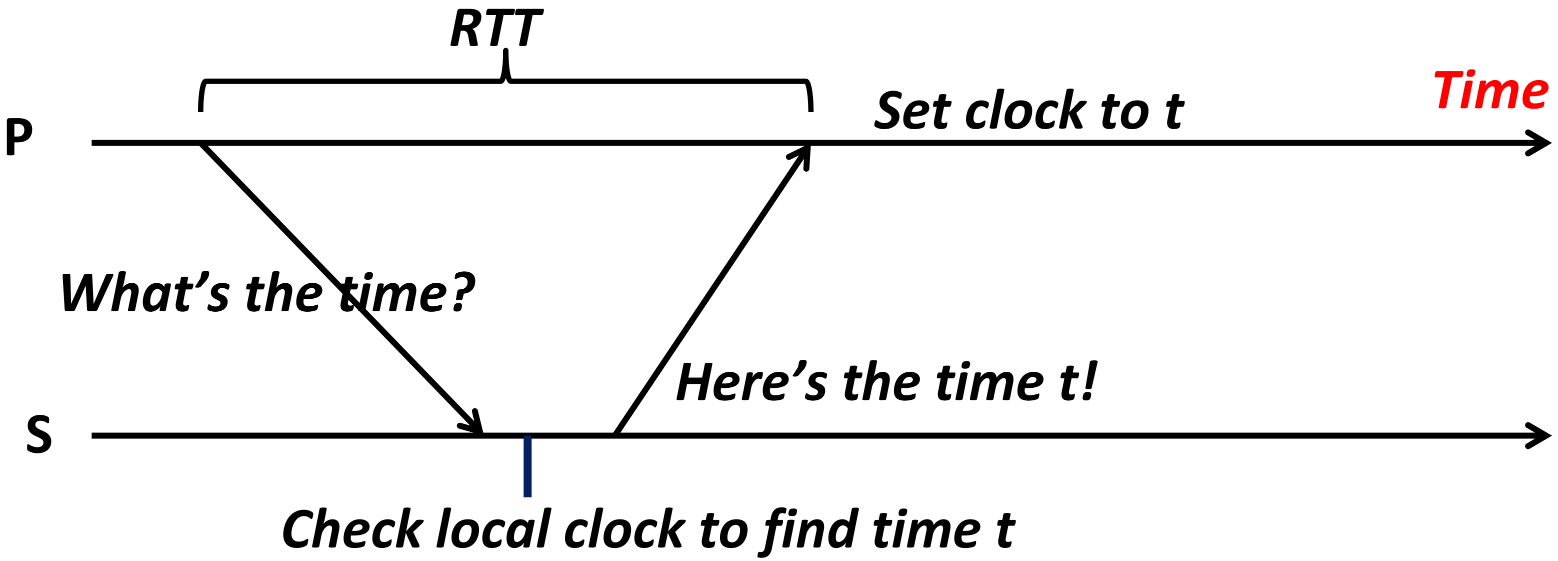
- External time synchronization
- All processes P synchronize with a time server S.



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

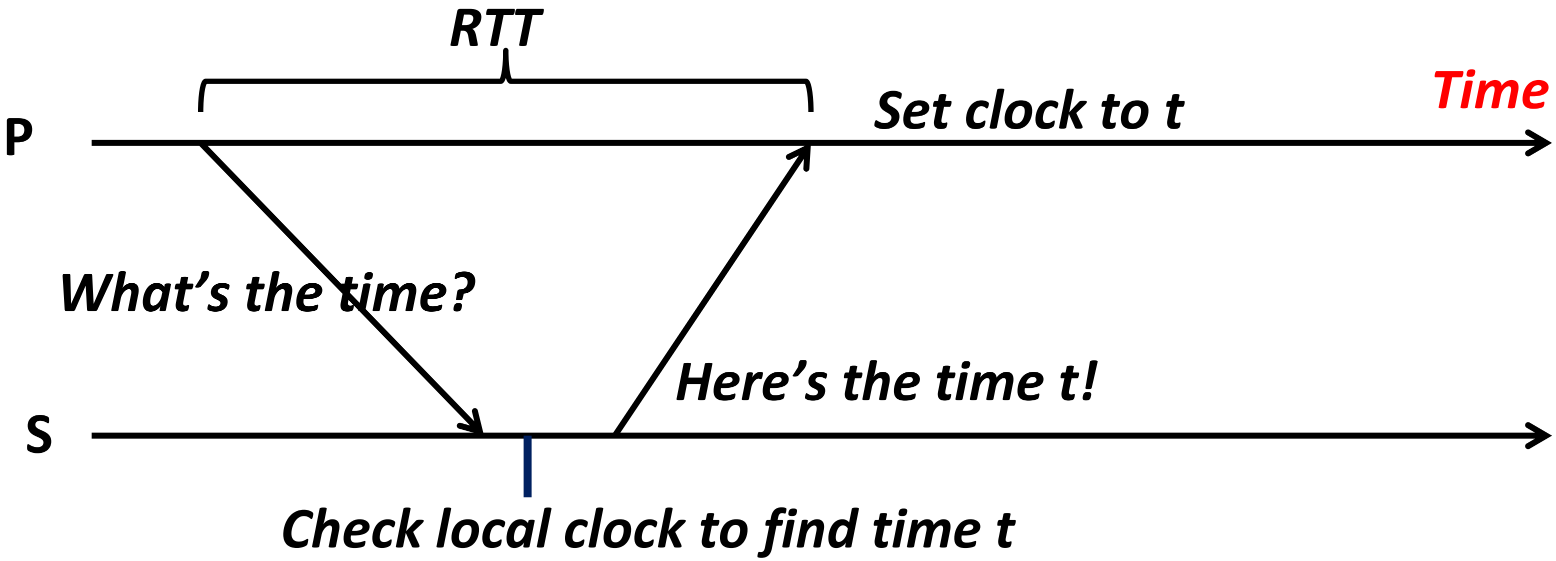
(i) Christians Algorithm

- 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 depends on the OS 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**]

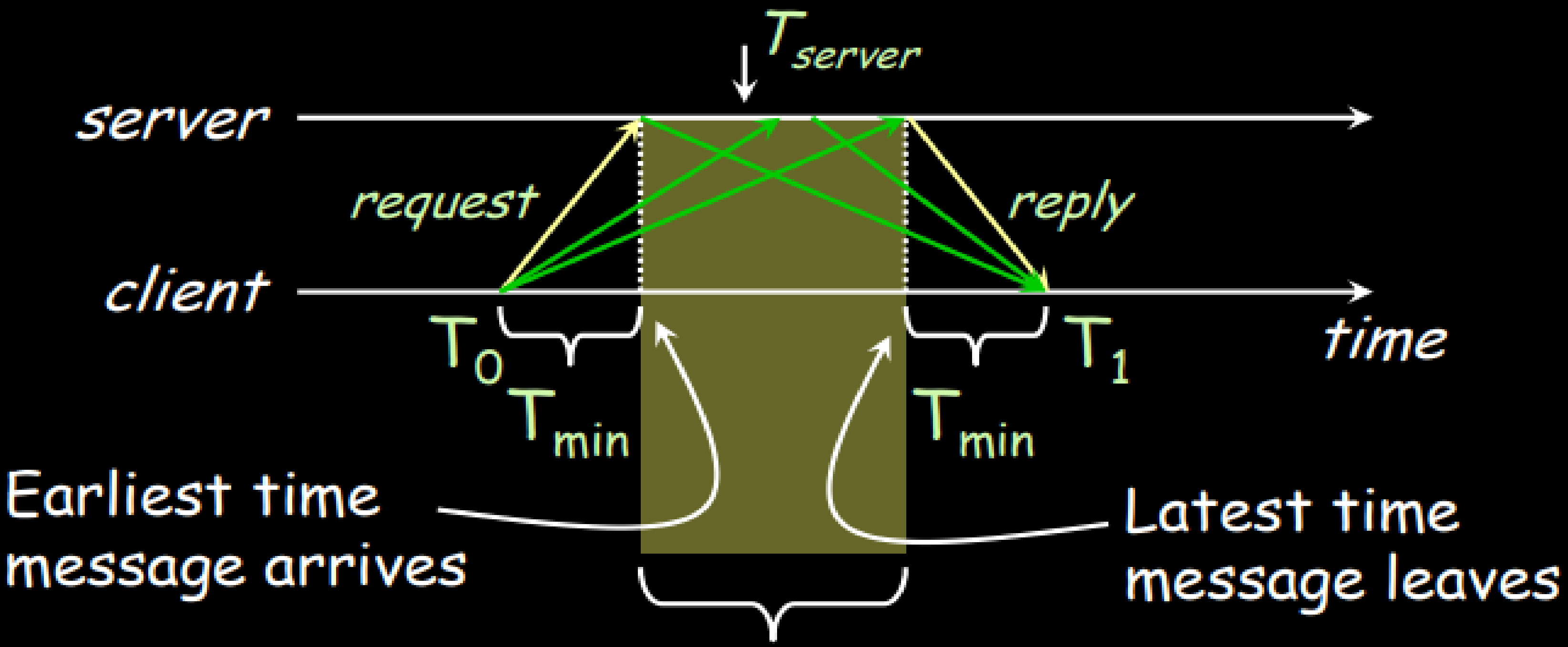


Christians Algorithm

- 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



Error Bounds



range = $T_1 - T_0 - 2T_{min}$

$$\boxed{\text{accuracy of result} = \pm \frac{T_1 - T_0}{2} - T_{min}}$$

Error Bounds

- **Allowed to increase clock value but should never decrease clock value**
  - May violate ordering of events within the same process.

- **Allowed to increase or decrease speed of clock**

- **If error is too high, take multiple readings and average them**

Christians Algorithm: Example

- Send request at 5:08:15.100 (T_0)
- Receive response at 5:08:15.900 (T_1)
  - – Response contains 5:09:25.300 (Tserver)
- Elapsed time is $T_1 - T_0$
  - 5:08:15.900 - 5:08:15.100 = 800 msec
- Best guess: timestamp was generated
  - 400 msec ago
- Set time to Tserver+ elapsed time
  - 5:09:25.300 + 400 = 5:09.25.700

$$\boxed{Error = \pm\frac{900-100}{2} - 200 = \pm\frac{800}{2} - 200 = \pm 200}$$

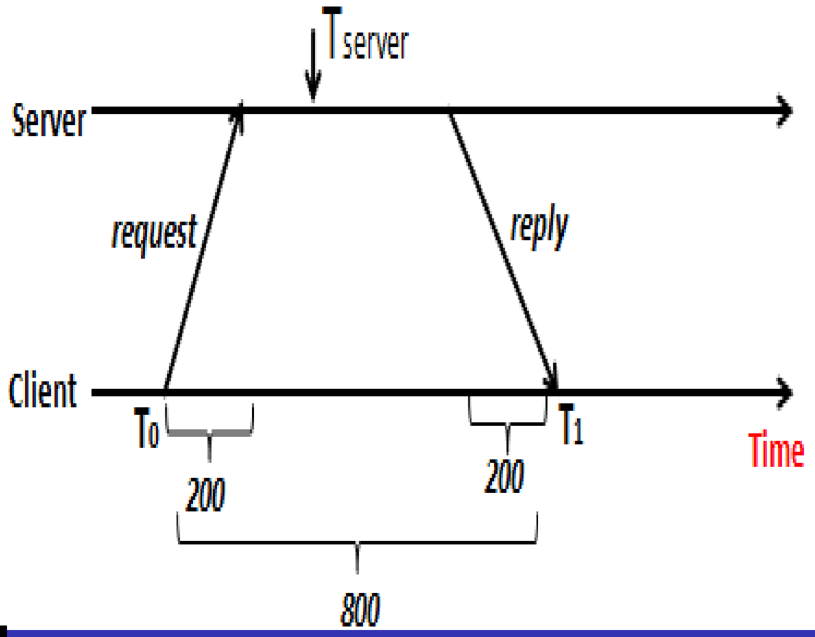
- If best-case message time=200 msec

T_0 = 5:08:15.100

T_1 = 5:08:15.900

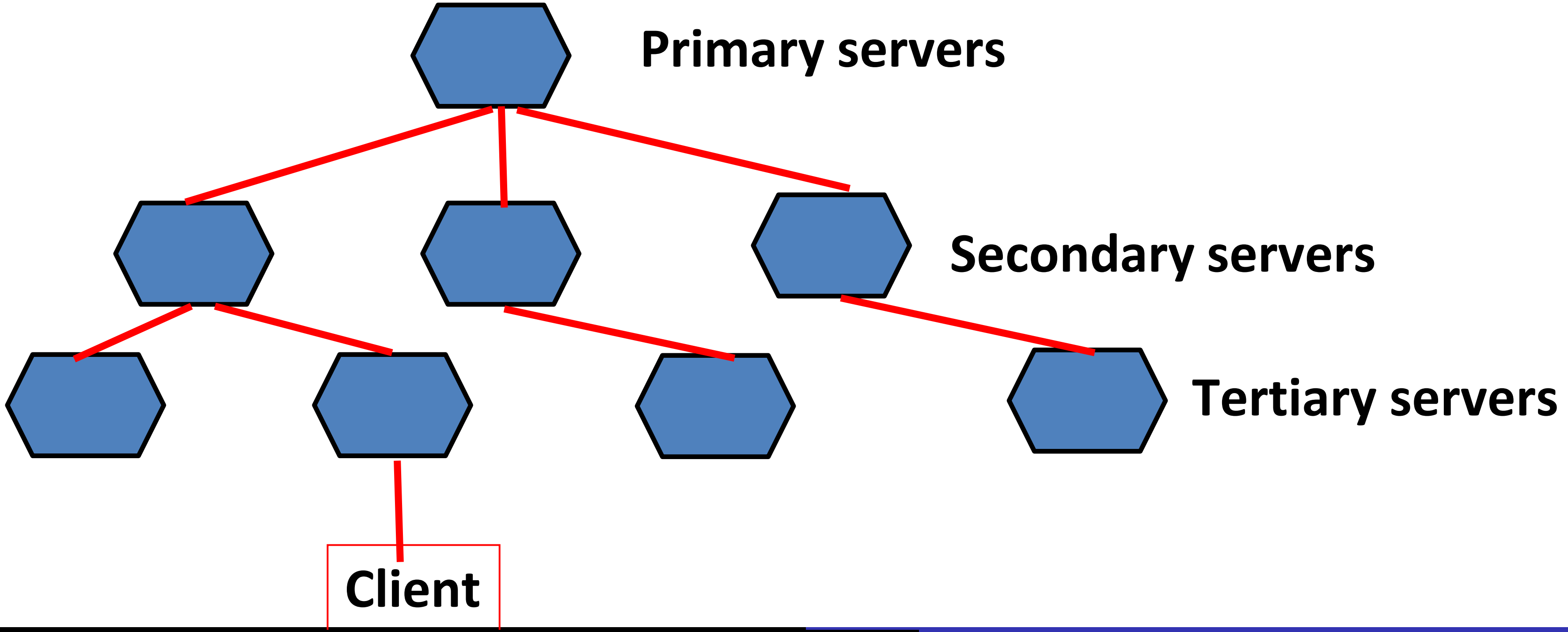
Tserver= 5:09:25:300

Tmin = 200msec

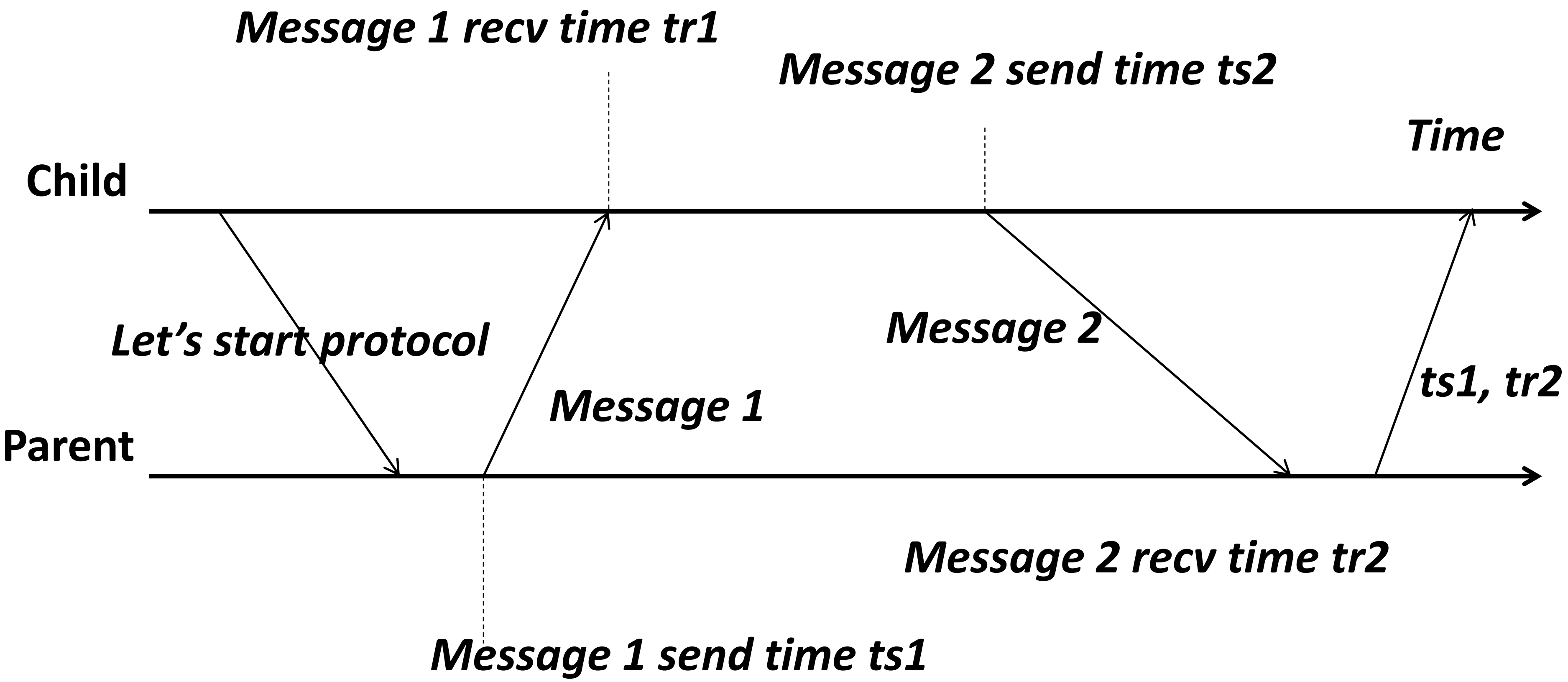


(ii) NTP: Network time protocol

- (1991, 1992) Internet Standard, version 3: RFC 1305
- **NTP servers organized in a tree.**
- Each client = a leaf of a tree.
- Each node synchronizes with its tree parent



NTP Protocol



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

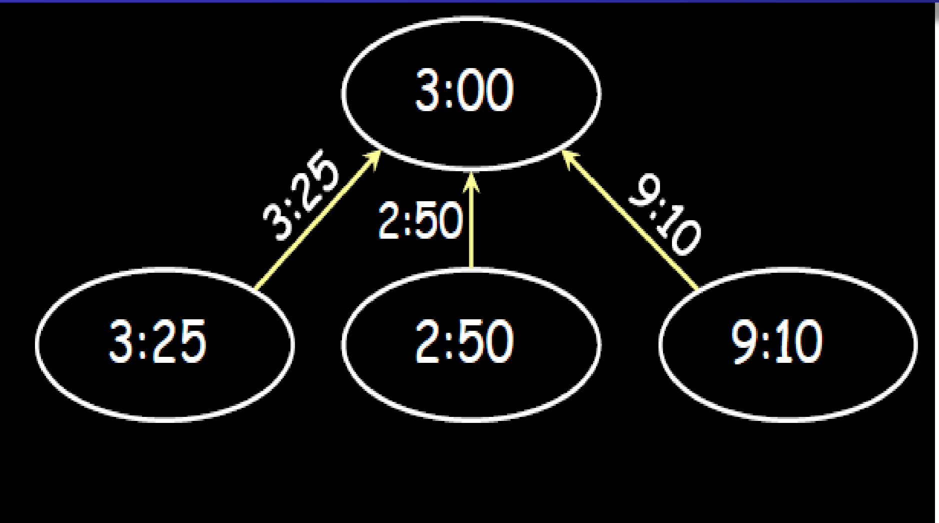
Why $o = (tr1-tr2 + ts2- ts1)/2$?

- **Then**
  - $tr1 = ts1 + L1 + oreal$.
  - $tr2 = ts2 + L2 – oreal$.
- **Subtracting second equation from 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 (RTT)

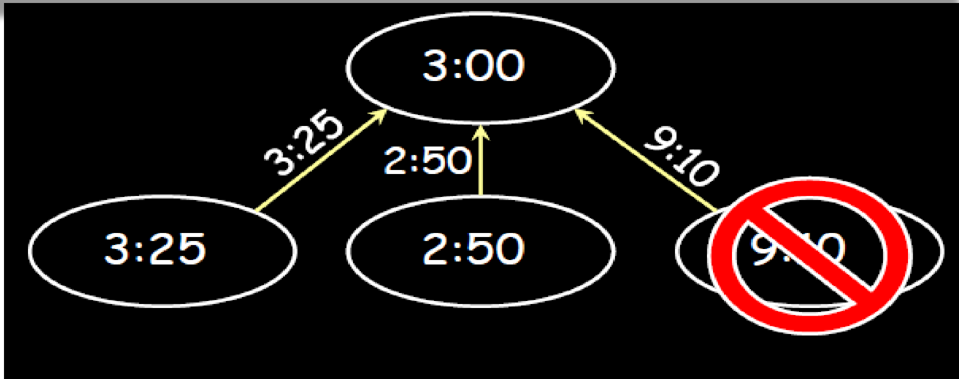
(iii) Berkley's Algorithm

- **Gusella & Zatti, 1989**
- Master poll's each machine periodically
  - Ask each machine for time
    - Can use Christian's algorithm to compensate the network's latency.
- When results are in compute,
  - Including master's time.
- **Hope: average cancels out individual clock's tendency to run fast or slow**
- Send offset by which each clock needs adjustment to each slave
  - Avoids problems with network delays if we send a time-stamp.

Berkley's Algorithm : Example

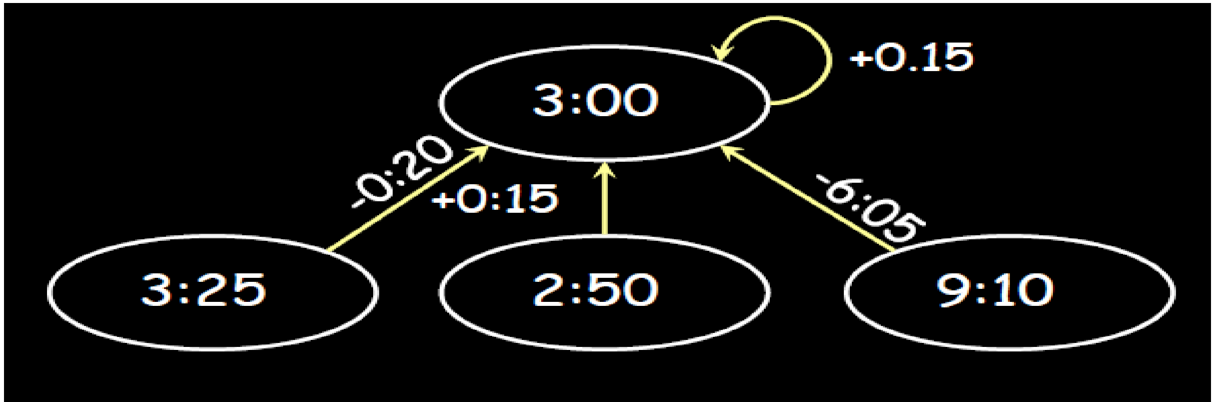


1. Request timestamps from all slaves



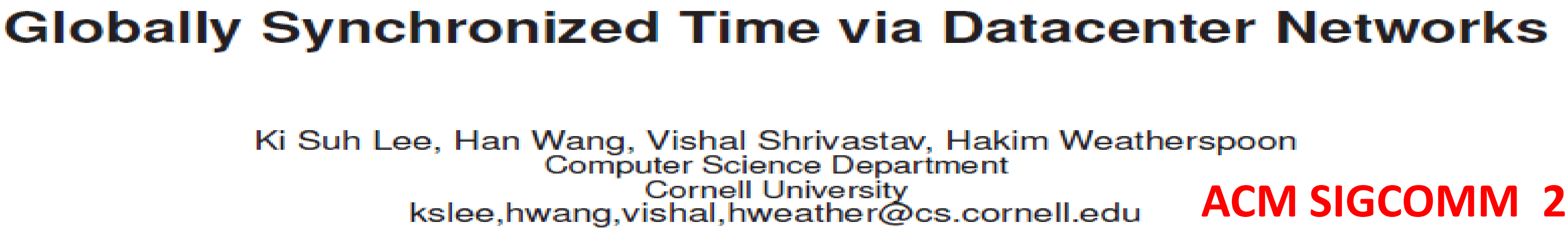
2. Compute fault-tolerant average:

$$\frac{3{:}25 + 2{:}50 + 3{:}00}{3} = 3{:}05$$



3. Send offset to each client

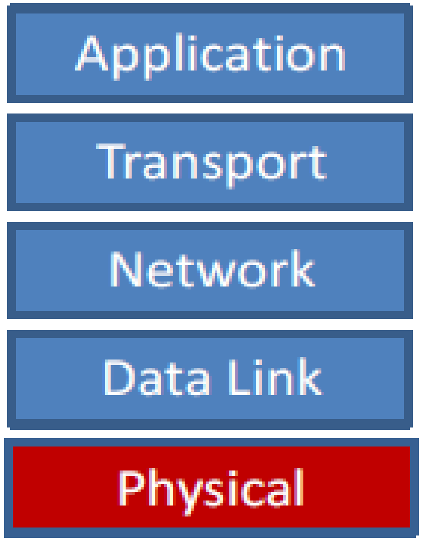
(iv) DTP: Datacenter Time Protocol



Globally Synchronized Time via Datacenter Networks

Ki Suh Lee, Han Wang, Vishal Shrivastav, Hakim Weatherspoon
Computer Science Department
Cornell University
kslee,hwang,vishal,hweather@cs.cornell.edu

ACM SIGCOMM 2016



Application

Transport

Network

Data Link

Physical

- DTP uses the physical layer of network devices to implement a decentralized clock synchronization protocol.
- ***Highly Scalable with bounded precision!***

– ~25ns (4 clock ticks) between peers

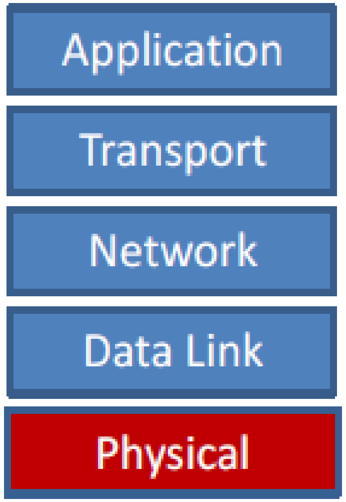
– ~150ns for a datacenter with six hops

– No Network Traffic

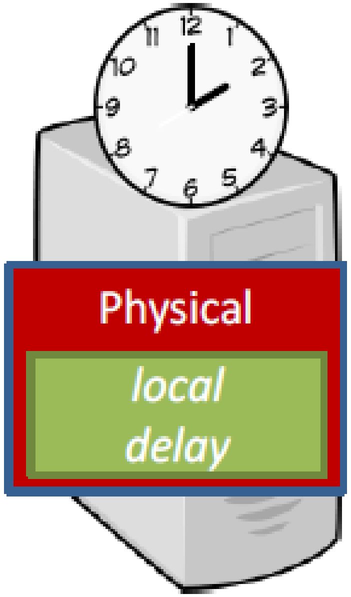
– *Internal Clock Synchronization*

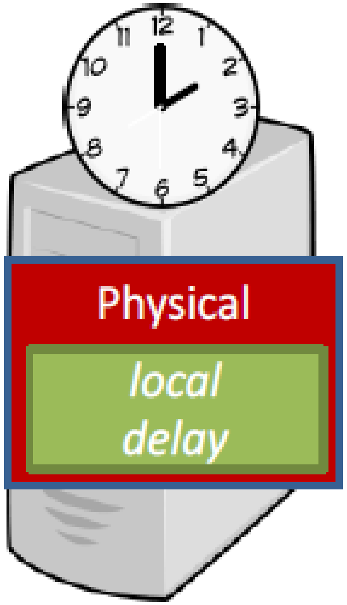
- End-to-End: ~200ns precision!

DTP: Phases



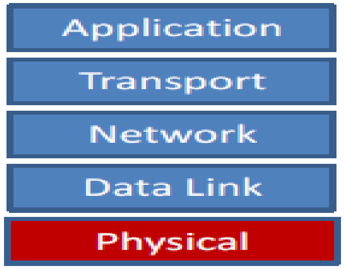
- Runs in two phases between two peers
  - Init Phase: Measuring OWD (one-way delay)
  - Beacon Phase: Re-Synchronization



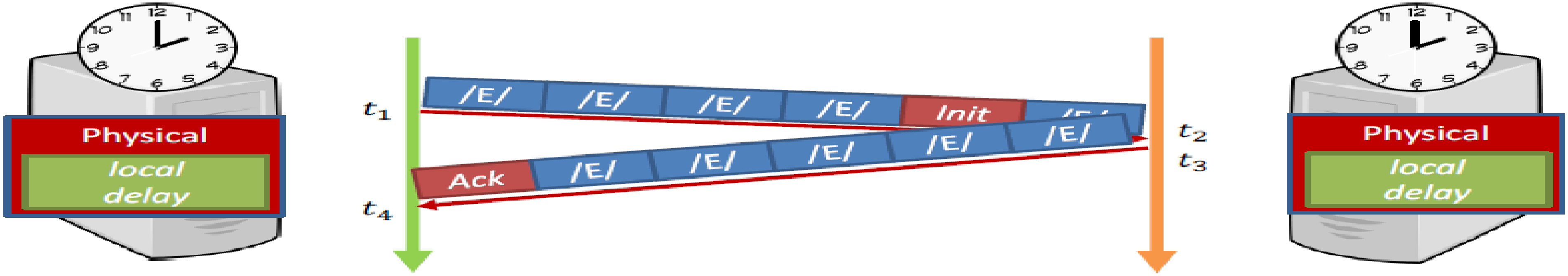


DTP: (i) Init Phase

- **INIT phase:** The purpose of the INIT phase is to measure the one-way delay between two peers. The phase begins when two ports are physically connected and start communicating, i.e. when the link between them is established.
- Each peer measures the one-way delay by measuring the time between sending an INIT message and receiving an associated INIT-ACK message, i.e. measure RTT, then divide the measured RTT by two.

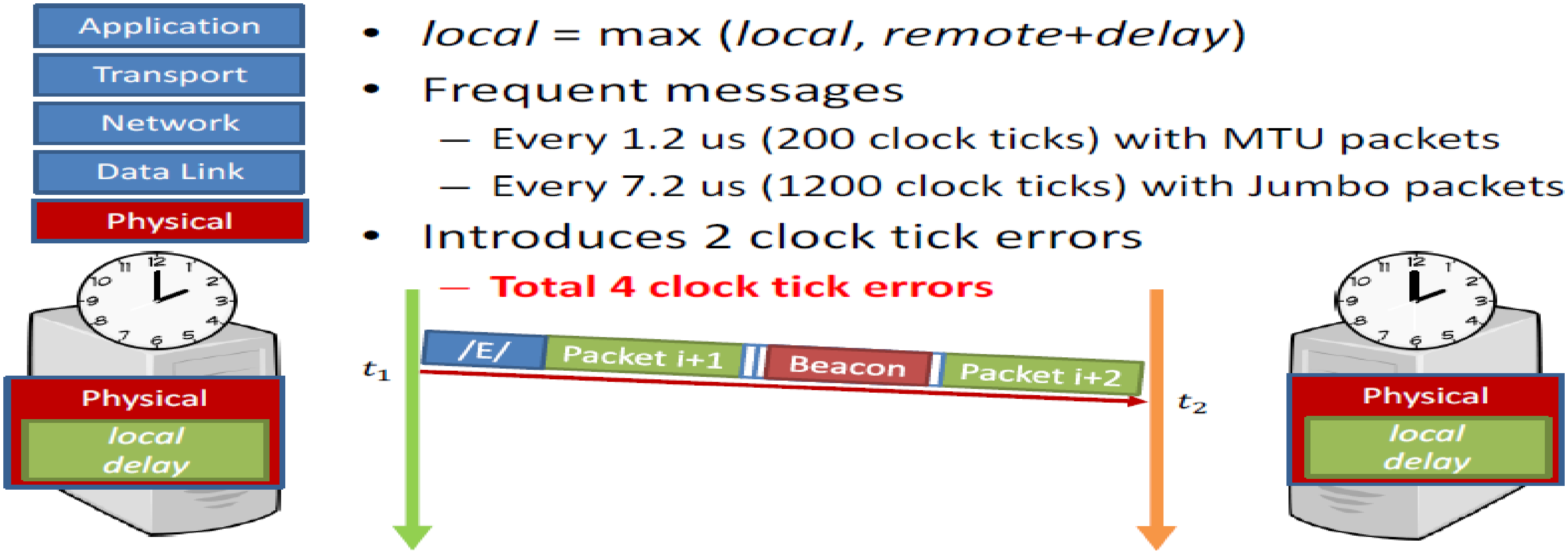


- $delay = (t_4 - t_1 - \alpha)/2$
  - α=3: Ensure *delay* is always less than actual delay
- Introduce 2 clock tick errors
  - Due to oscillator skew, timing and Sync FIFO



DTP: (ii) Beacon Phase

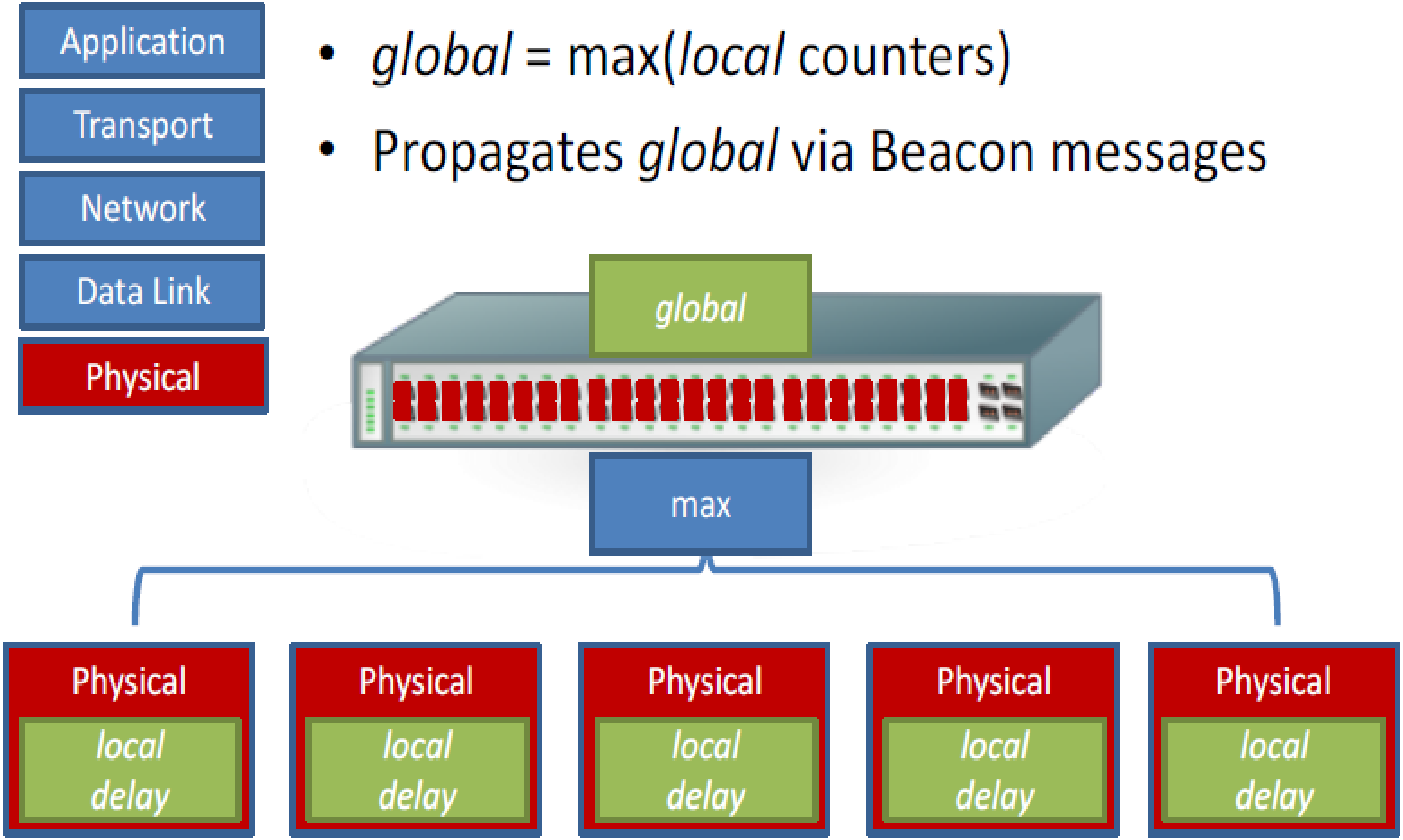
- **BEACON phase:** During the BEACON phase, two ports periodically exchange their local counters for resynchronization. Due to oscillator skew, the offset between two local counters will increase over time. A port adjusts its local counter by selecting the maximum of the local and remote counters upon receiving a BEACON message from its peer. Since BEACON messages are exchanged frequently, hundreds of thousands of times a second (every few microseconds), the offset can be kept to a minimum.



- *local* = max (*local*, *remote*+*delay*)
- Frequent messages
  - Every 1.2 us (200 clock ticks) with MTU packets
  - Every 7.2 us (1200 clock ticks) with Jumbo packets
- Introduces 2 clock tick errors
  - Total 4 clock tick errors

DTP Switch

- *global* = max(*local* counters)
- Propagates *global* via Beacon messages



DTP Property

- DTP provides bounded precision and scalability

- Bounded Precision in hardware
  - Bounded by 4T (=25.6ns, T=oscillator tick is 6.4ns)
  - Network precision bounded by 4*TD*

    D is network diameter in hops

- Requires NIC and switch modifications

But Yet…

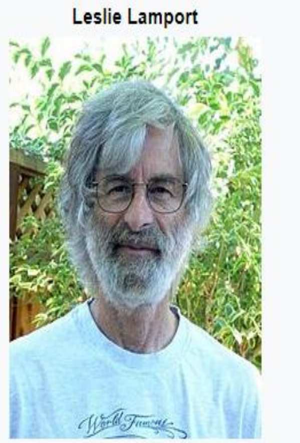
- **We still have a non-zero error!**
- **We just can’t seem to get rid of error**
  - Can’t as long as messages latencies are non-zero.
- **Can we avoid synchronizing clocks altogether, and still be able to order events ?**

Ordering events in a distributed system

- To order events across processes, trying to synchronize clocks is an approach.
- What if we instead assigned timestamps to events that were not absolute time ?
- As long as those timestamps obey **causality**, that would work
  - If an event A causally happens before another event B, then timestamp(A) < timestamp (B)
  - Example: Humans use causality all the time
    - I enter the house only if I unlock it
    - 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.



- **Leslie B. Lamport** (born February 7, 1941) is an American computer scientist. Lamport is best known for his seminal work in distributed systems and as the initial developer of the document preparation system LaTeX. Leslie Lamport was the winner of the **2013 Turing Award** for imposing clear, well-defined coherence on the seemingly chaotic behavior of distributed computing systems, in which several autonomous computers communicate with each other by passing messages.

Lamport’s research contributions

- **Lamport’s research contributions have laid the foundations of the theory of distributed systems. Among his most notable papers are**
  - “Time, Clocks, and the Ordering of Events in a Distributed System”, which received the PODC Influential Paper Award in 2000,
  - “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs”,which defined the notion of Sequential consistency,
  - “The Byzantine Generals' Problem”,
  - “Distributed Snapshots: Determining Global States of a Distributed System” and
  - “The Part-Time Parliament”.
- These papers relate to such concepts as logical clocks (and the *happened-before* relationship) and Byzantine failures. They are among the most cited papers in the field of computer science and describe algorithms to solve many fundamental problems in distributed systems, including:
  - the Paxos algorithm for consensus,
  - the bakery algorithm for mutual exclusion of multiple threads in a computer system that require the same resources at the same time,
  - the Chandy-Lamport algorithm for the determination of consistent global states (snapshot), and
  - the Lamport signature, one of the prototypes of the digital signature.

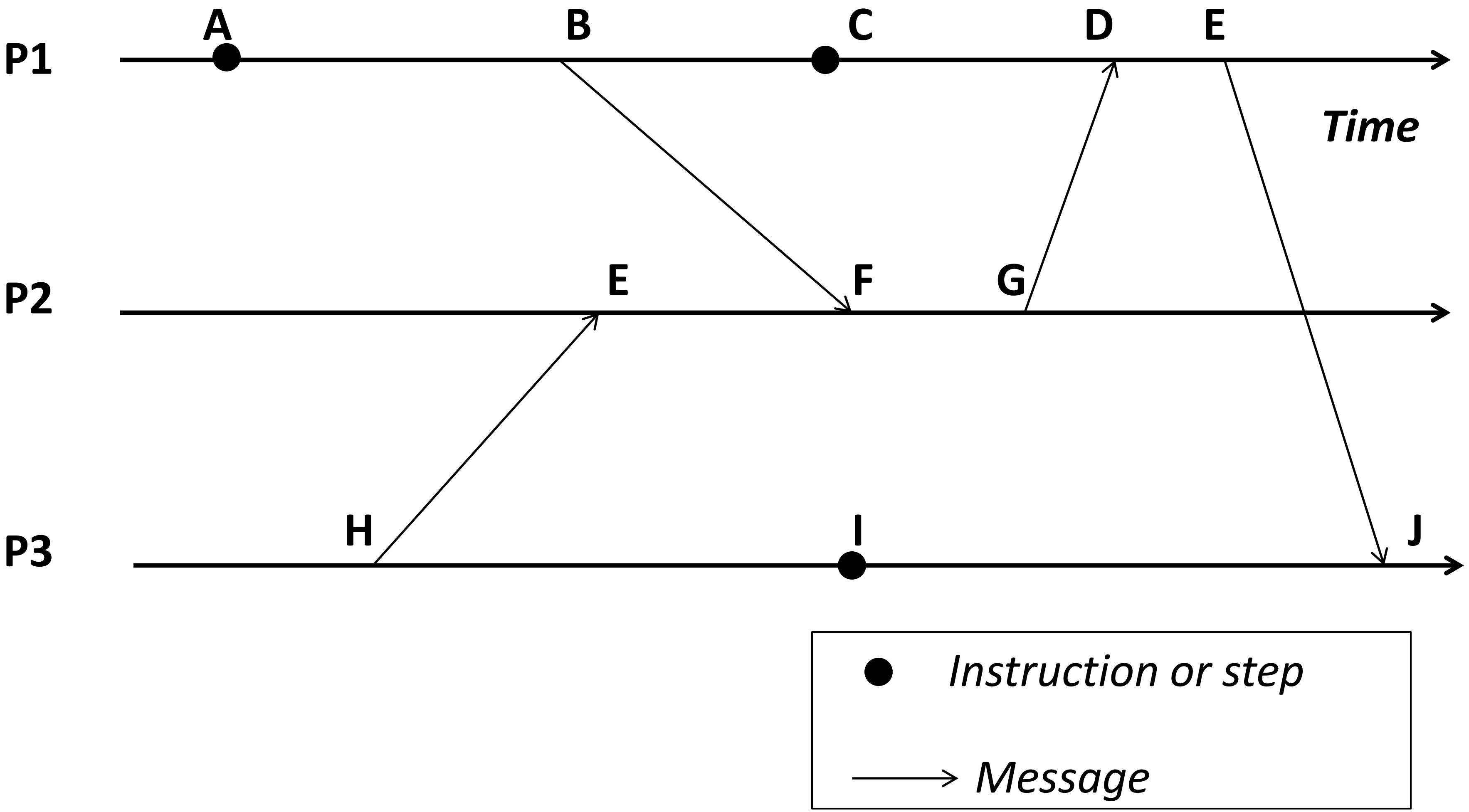
Logical (or Lamport) Ordering(2)

- Define a logical relation ***Happens-Before*** among pairs of events
- **Happens-Before denoted as →**
- **Three rules:**

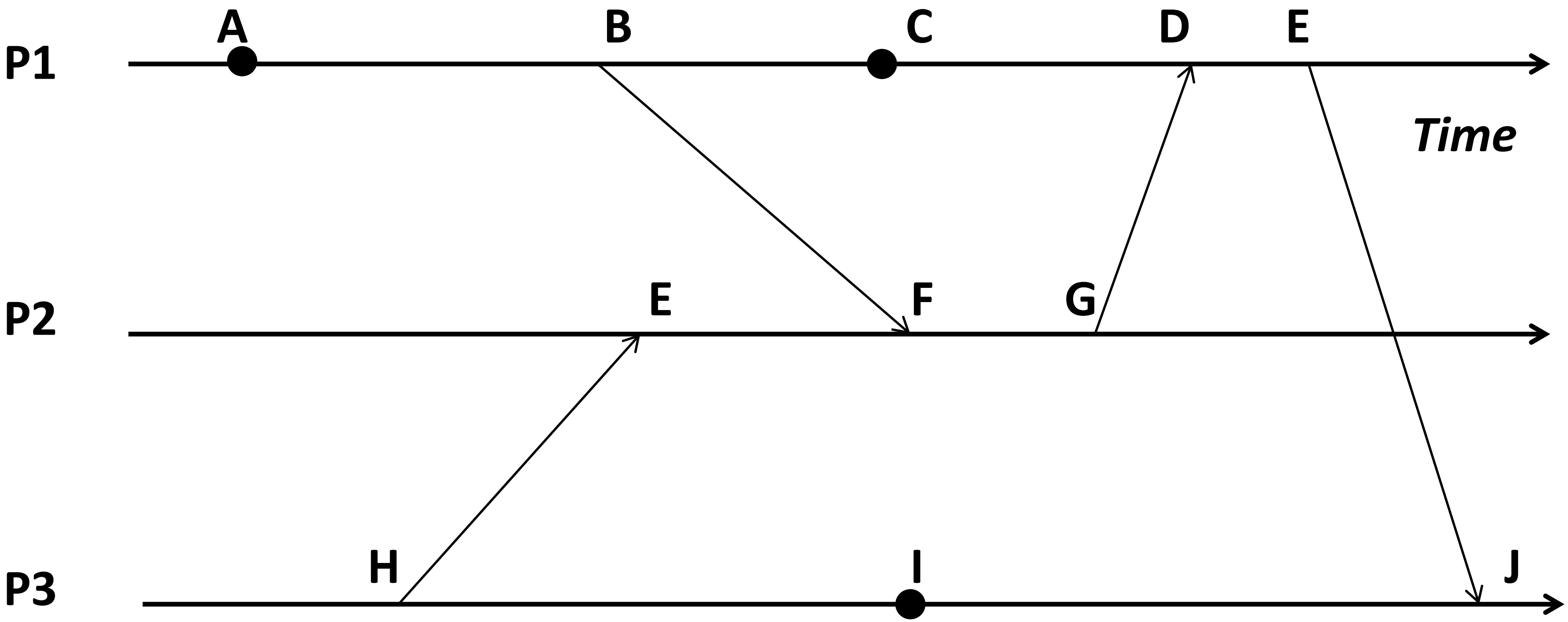
1. On the same process: ***a* → *b*, if *time(a)* < *time(b)*** (using the local clock)
2. If p1 sends *m* to p2: ***send(m)* → *receive(m)***
3. **(Transitivity)** If ***a* → *b and b* → *c*** then ***a* → *c***

- Creates a *partial order* among events
  - Not all events related to each other via →

Example 1:



Example 1: Happens-Before

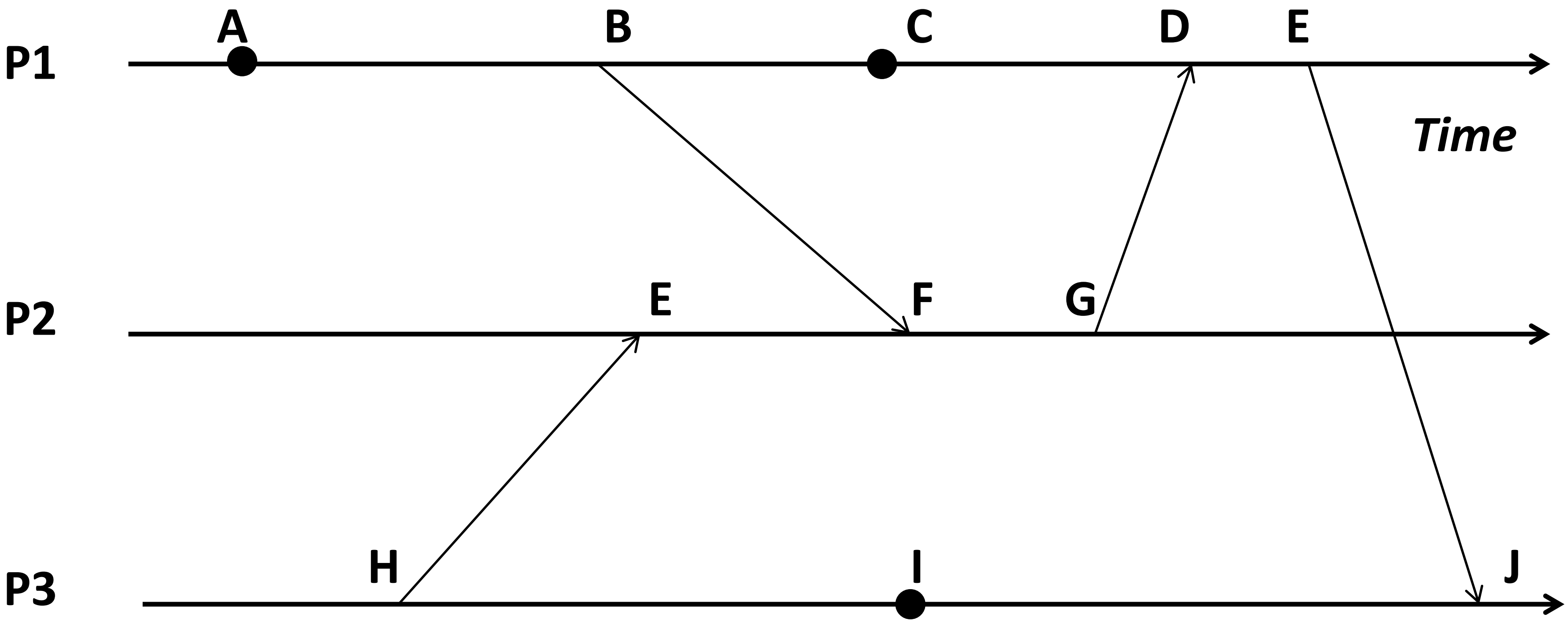


- A → B
- B → F
- A → F

● *Instruction or step*

⟶ *Message*

Example 2: Happens-Before



- H → G
- F → J
- H → J
- C → J

● *Instruction or step*

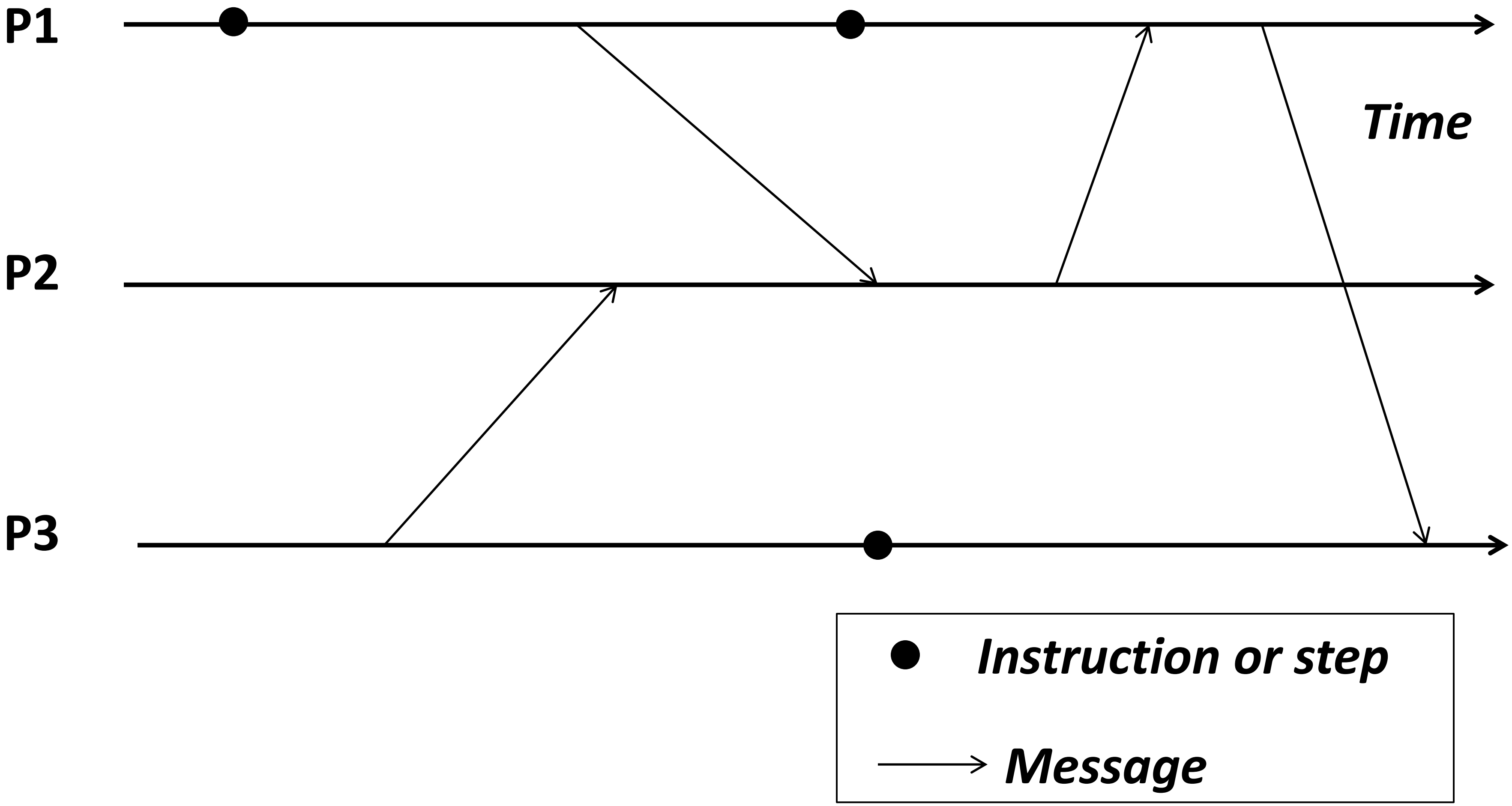
⟶ *Message*

Lamport timestamps

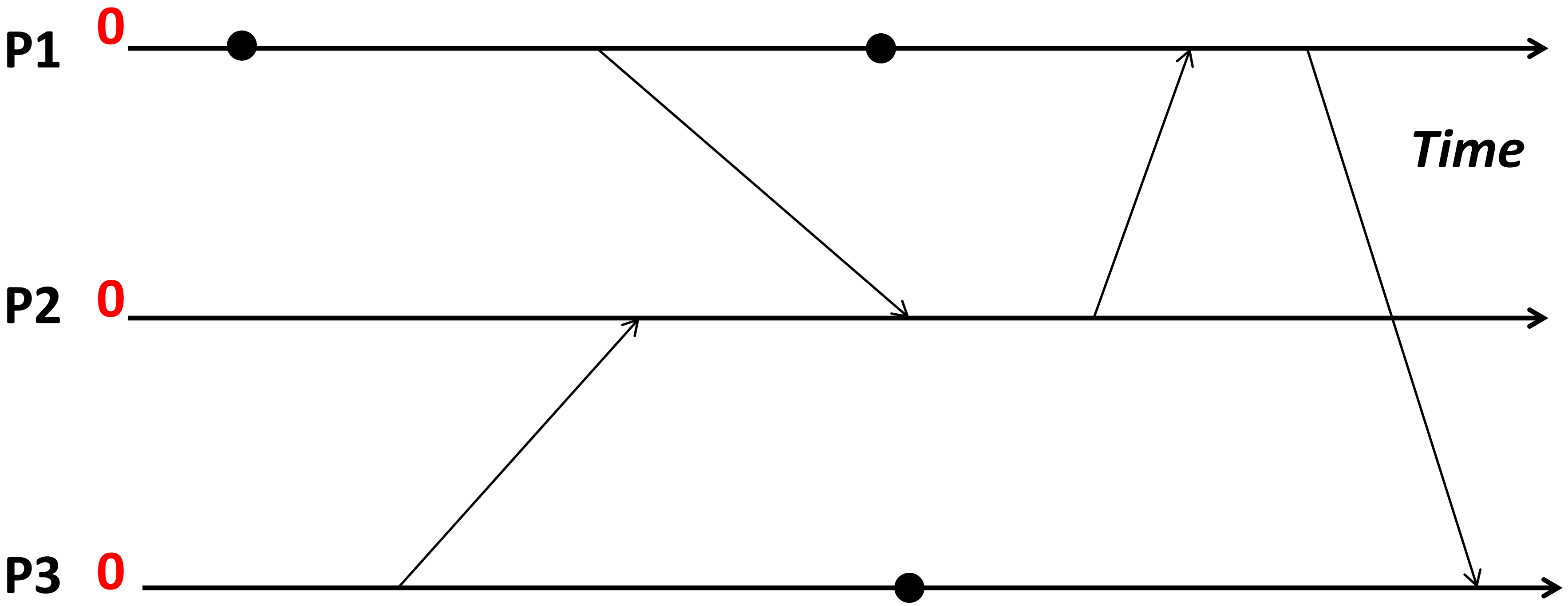
- **Goal: Assign logical (Lamport) timestamp to each event**
- **Timestamps obey causality**
- **Rules**
  - Each process uses a local counter (clock) which is an integer
    - initial value of counter is zero
  - A process increments its counter when a **send** or an **instruction** happens at it. The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

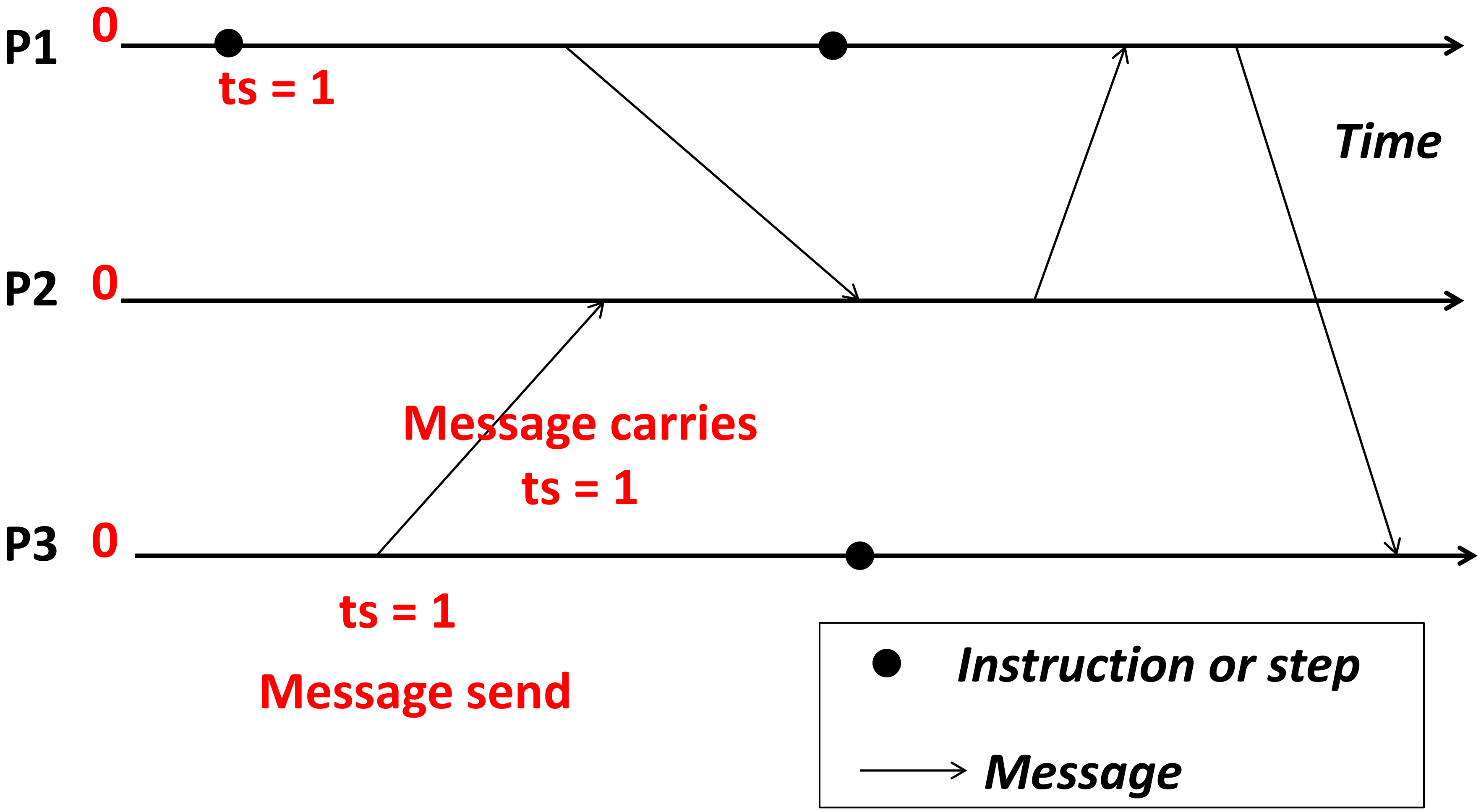


● *Instruction or step*

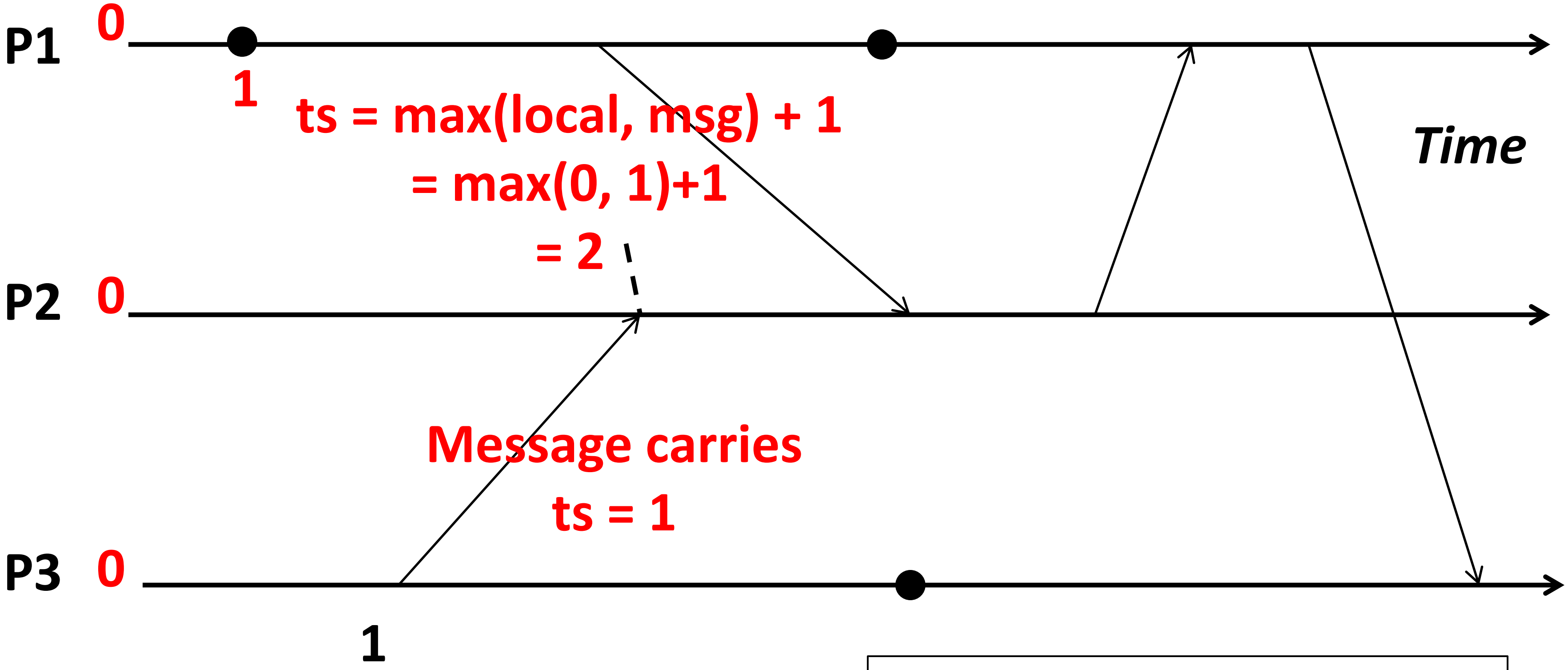
→ *Message*

Initial counters (clocks)

Lamport Timestamps

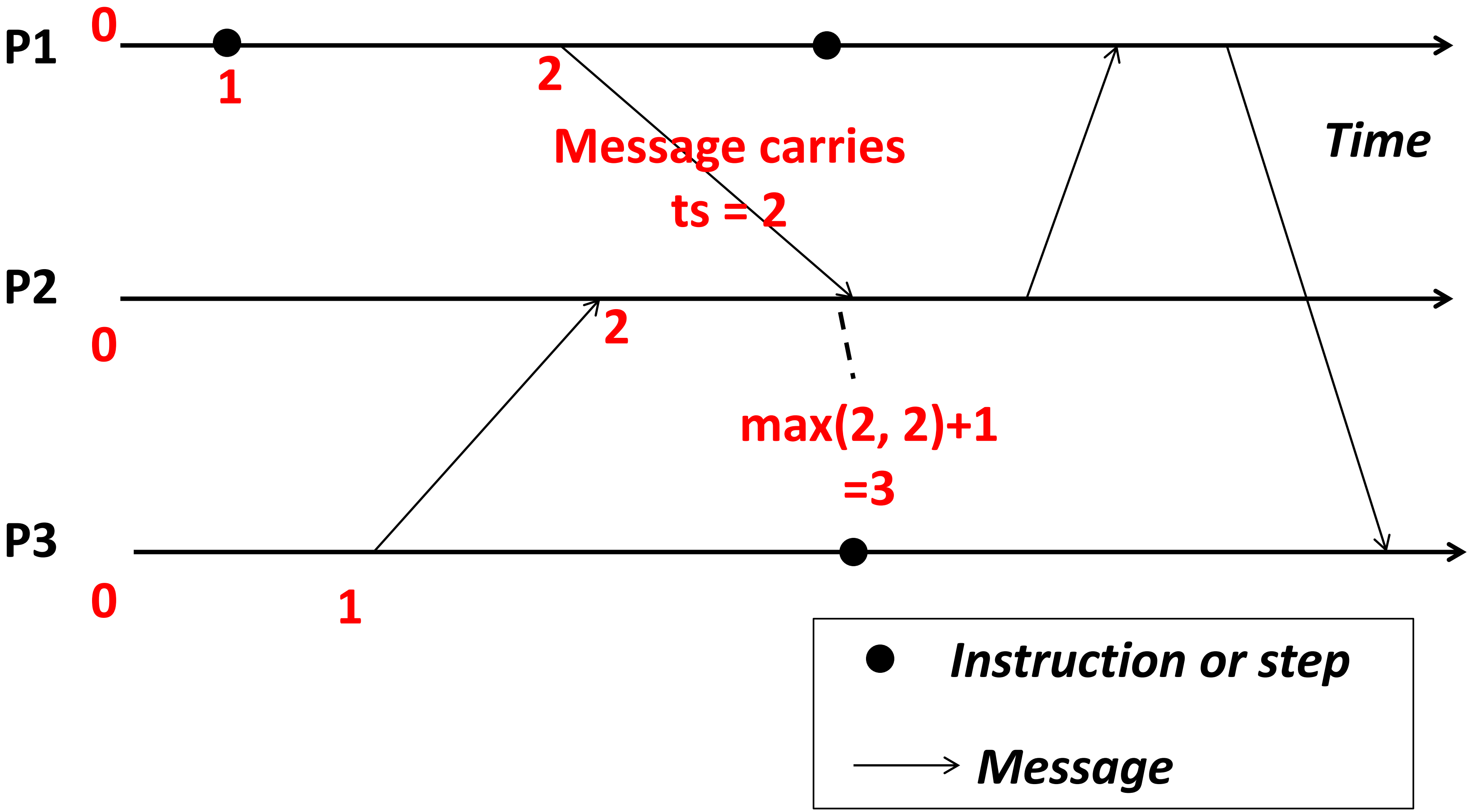


Lamport Timestamps

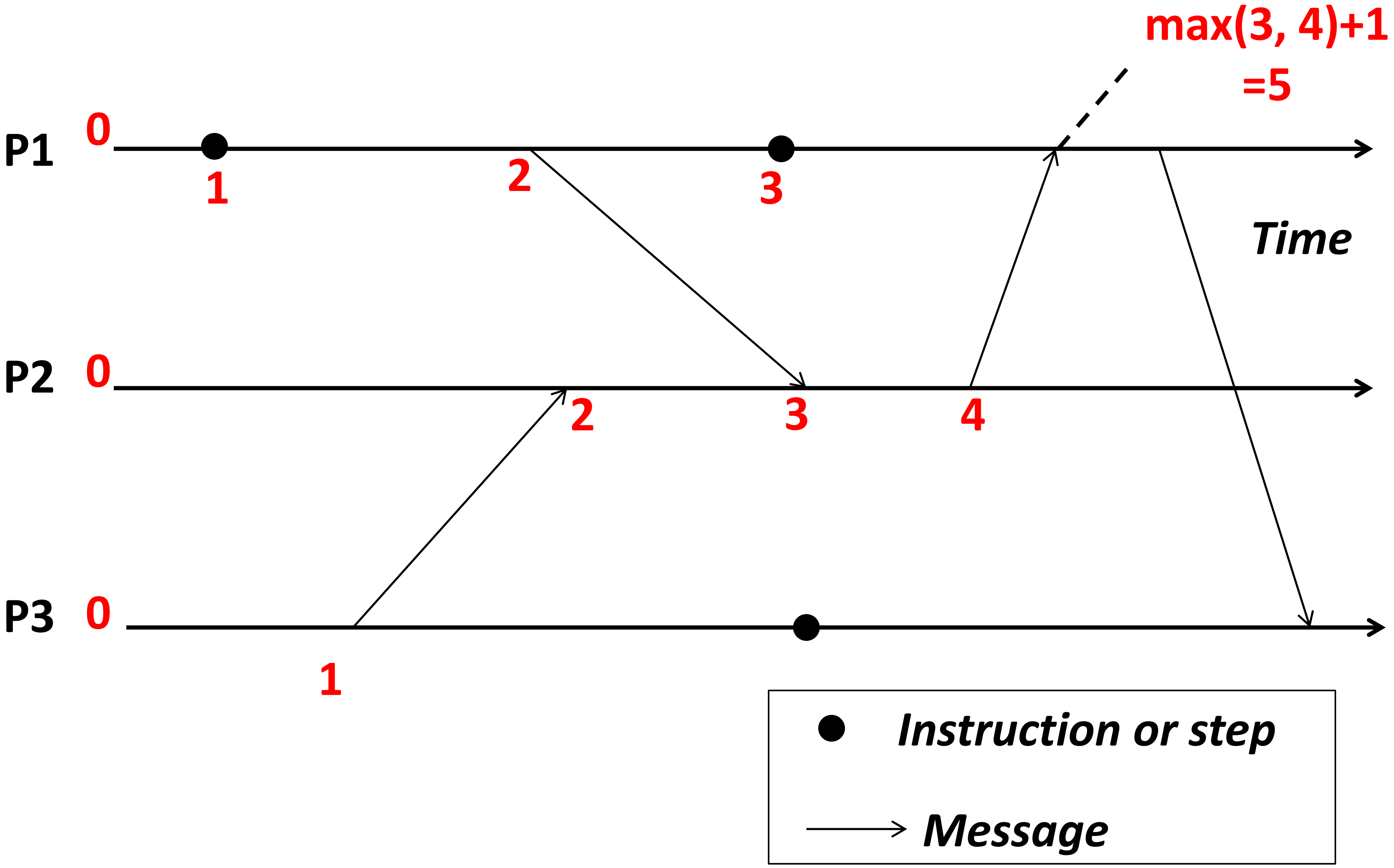


- ● *Instruction or step*
- → *Message*

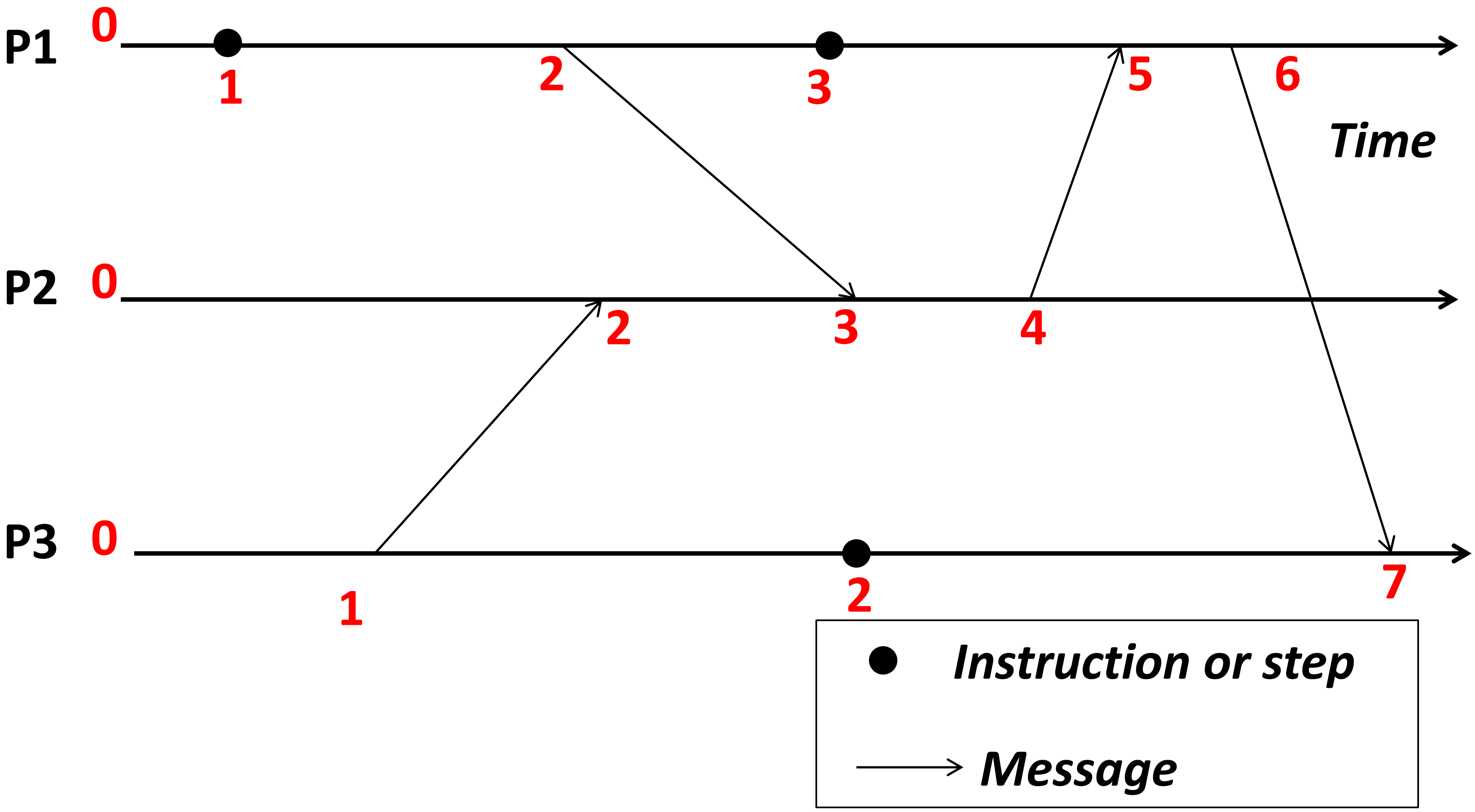
Lamport Timestamps



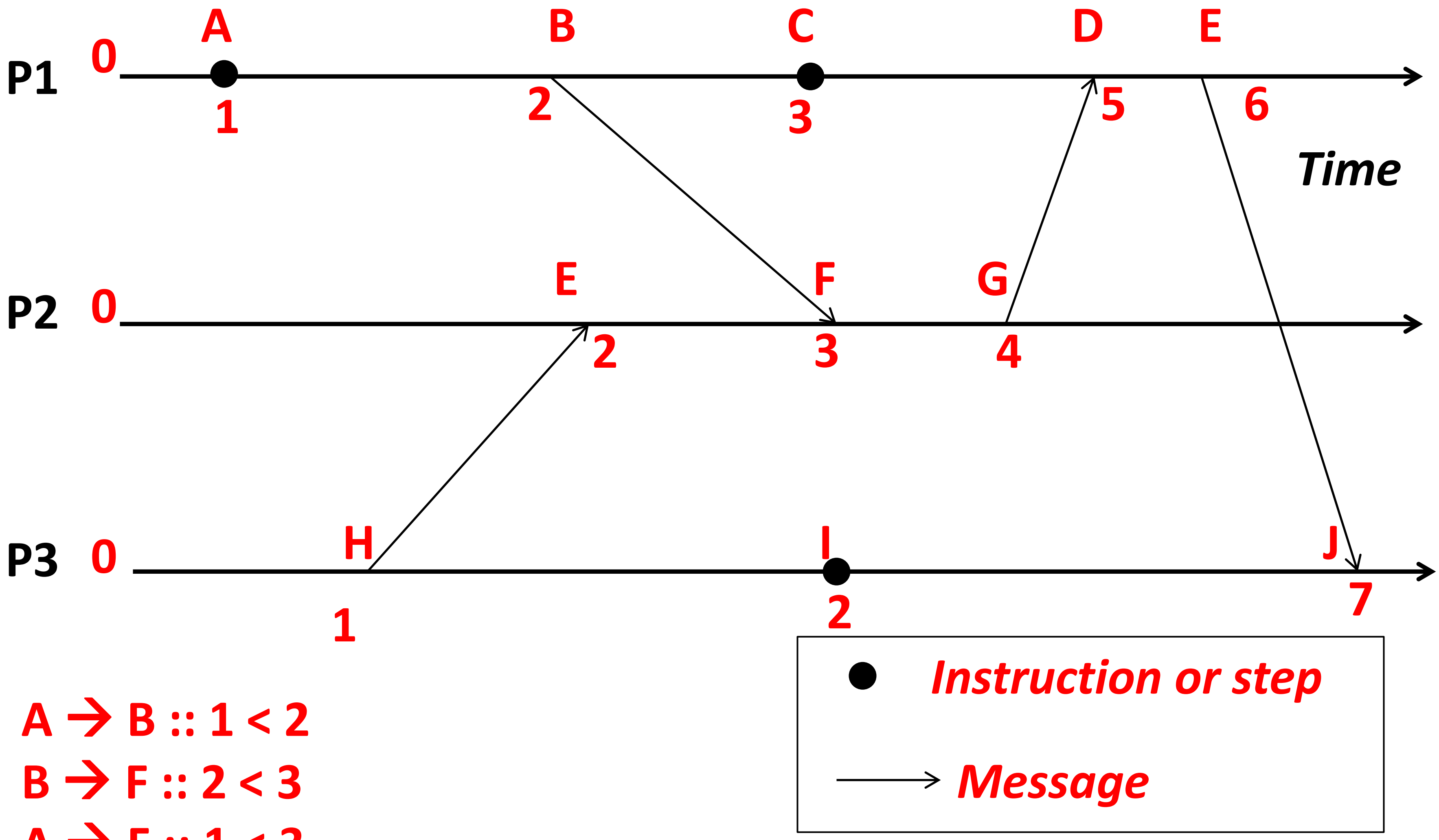
Lamport Timestamps



Lamport Timestamps

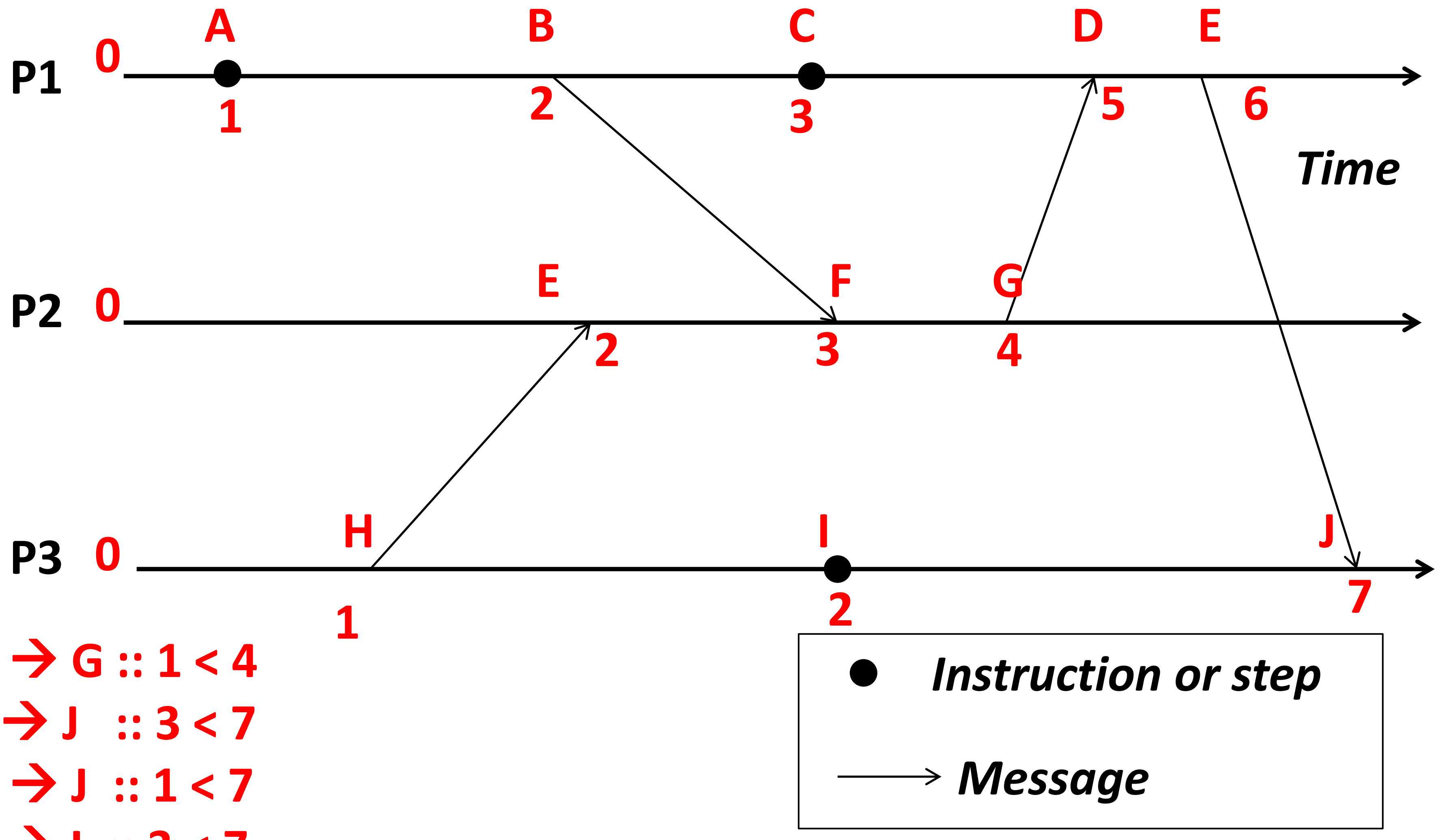


Obeying Causality



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

Obeying Causality (2)



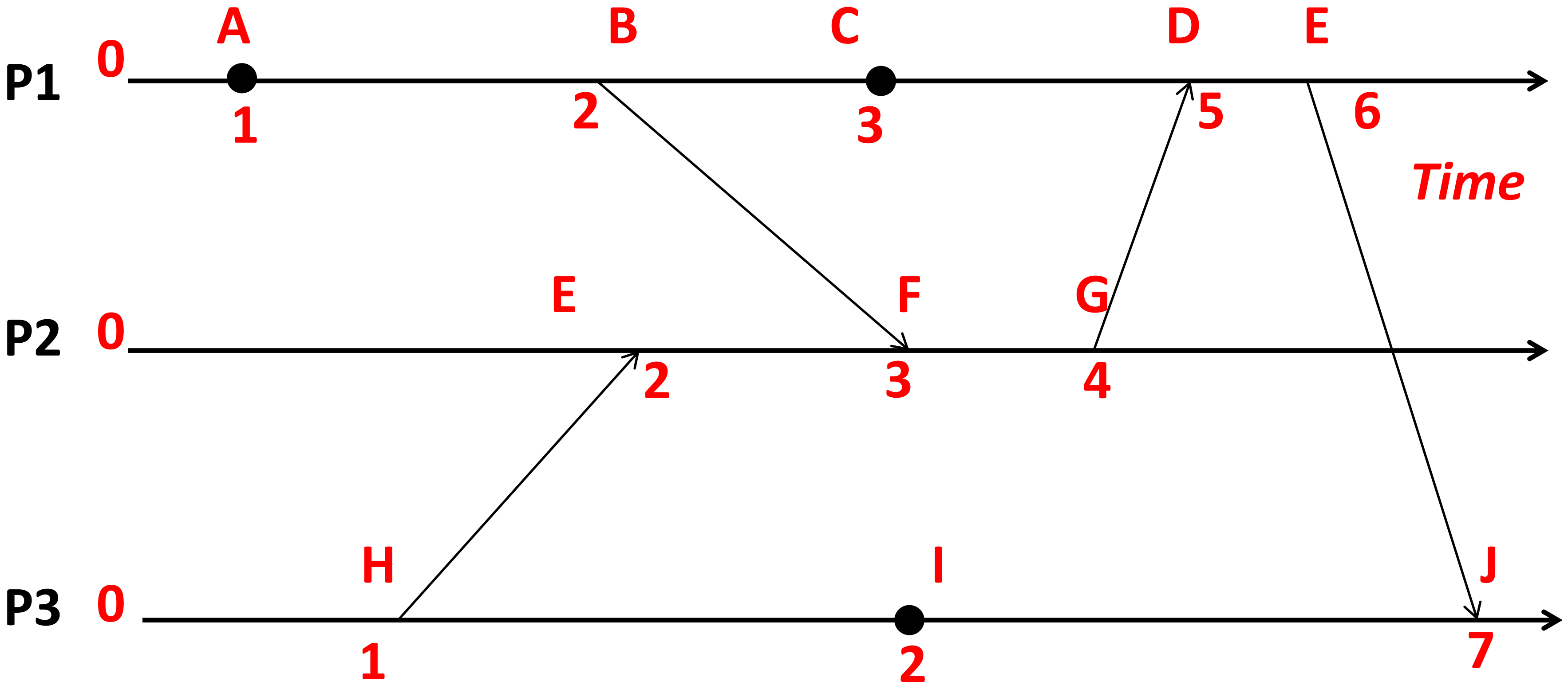
H → G :: 1 < 4

F → J :: 3 < 7

H → J :: 1 < 7

C → J :: 3 < 7

Not always *implying* Causality



● *Instruction or step*

→ *Message*

- ? 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:**

$E1 \rightarrow E2 \Rightarrow \text{timestamp}(E1) < \text{timestamp}(E2)$, **BUT**

$\text{timestamp}(E1) < \text{timestamp}(E2) \Rightarrow$

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

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* **V_i [*1…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

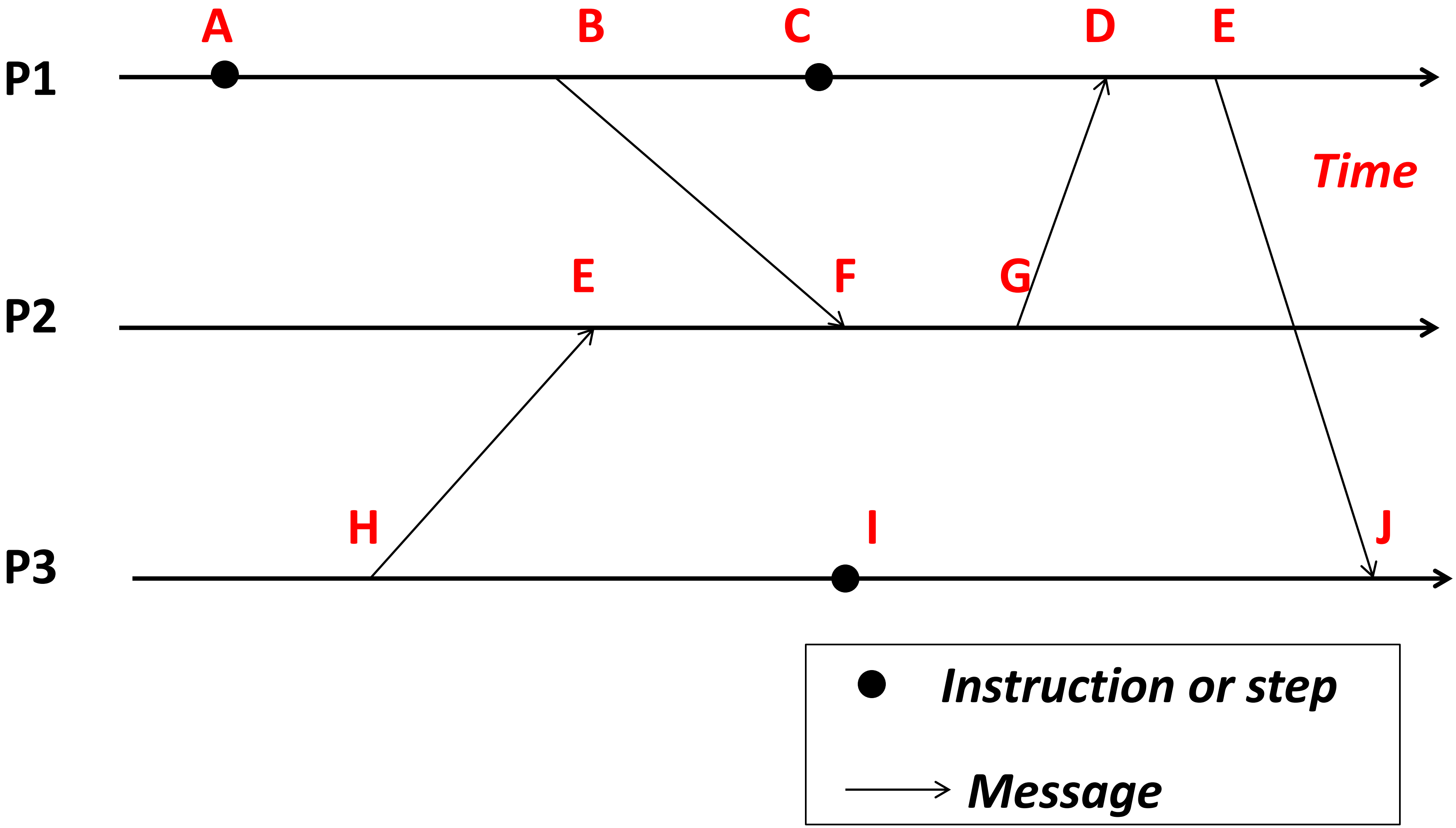
Incrementing vector clocks

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 $V_{message}[1...N]$
3. On receiving a message at process *i*:

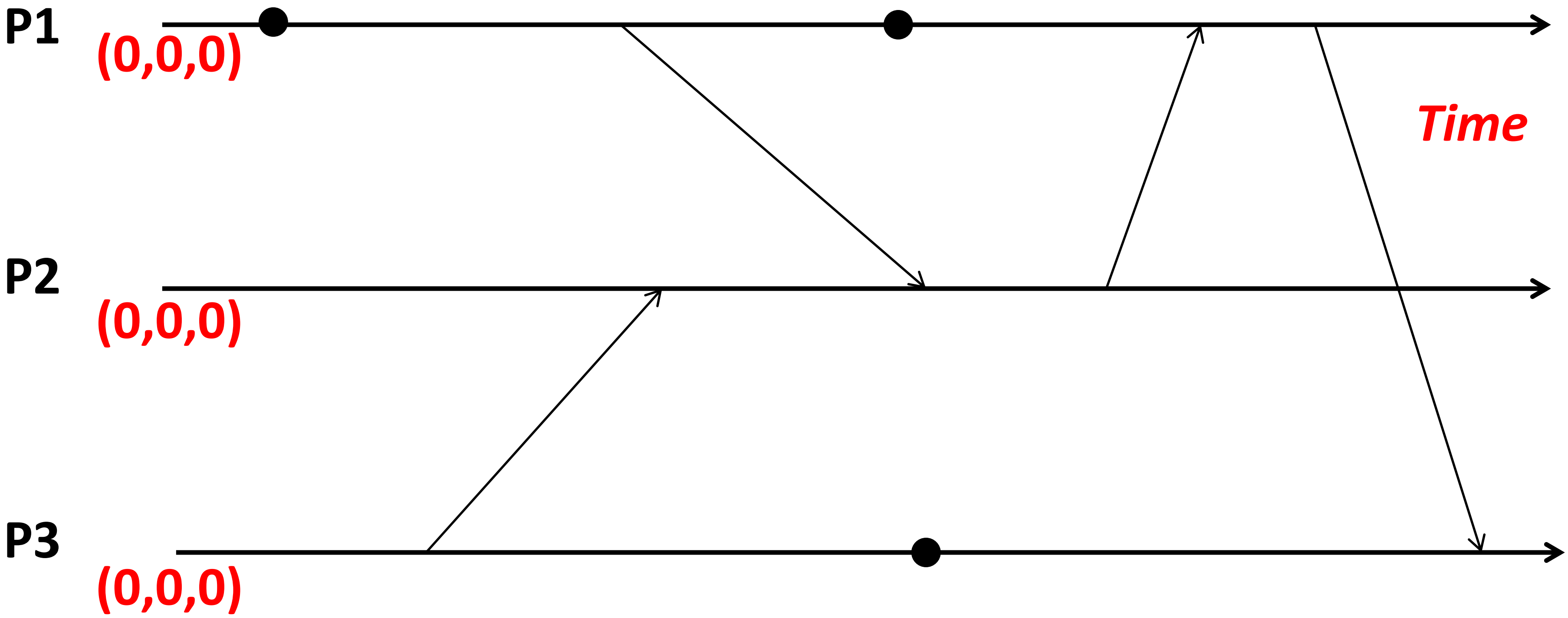
   $V_i[i] = V_i[i] + 1$

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

Example

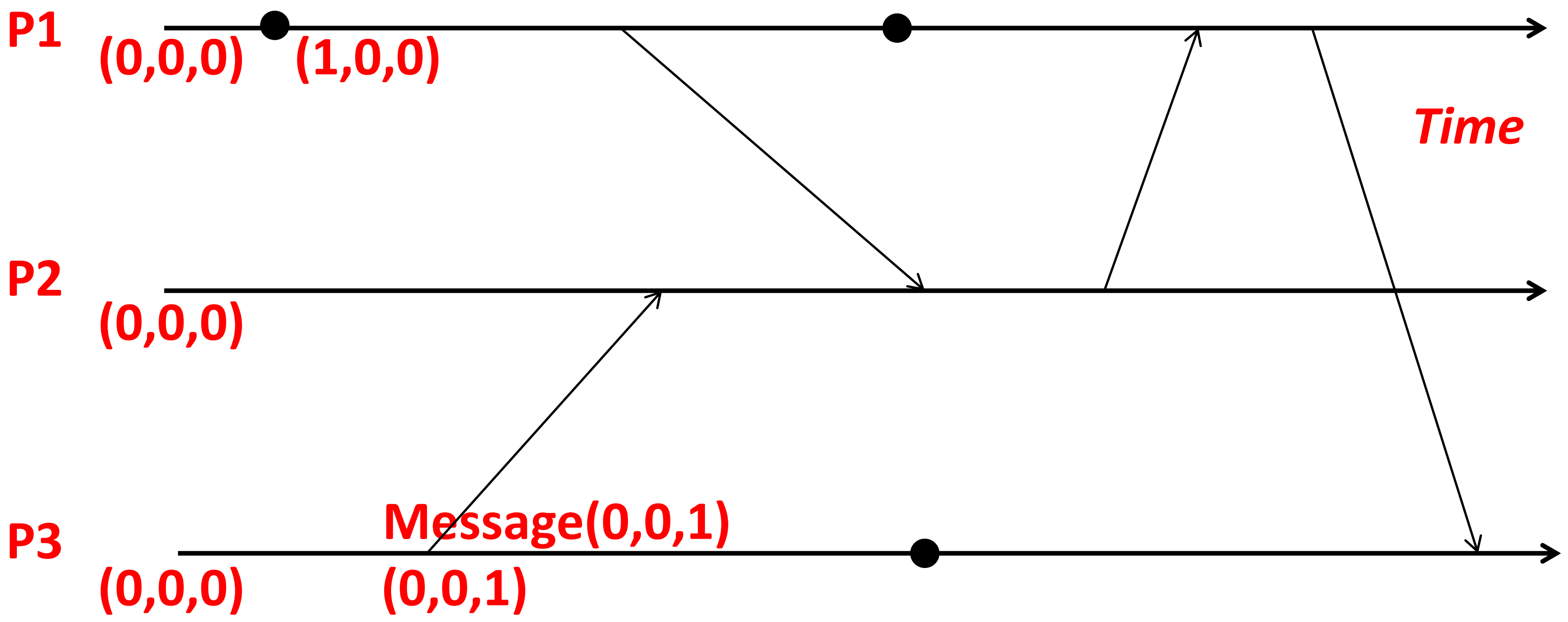


Vector Timestamps

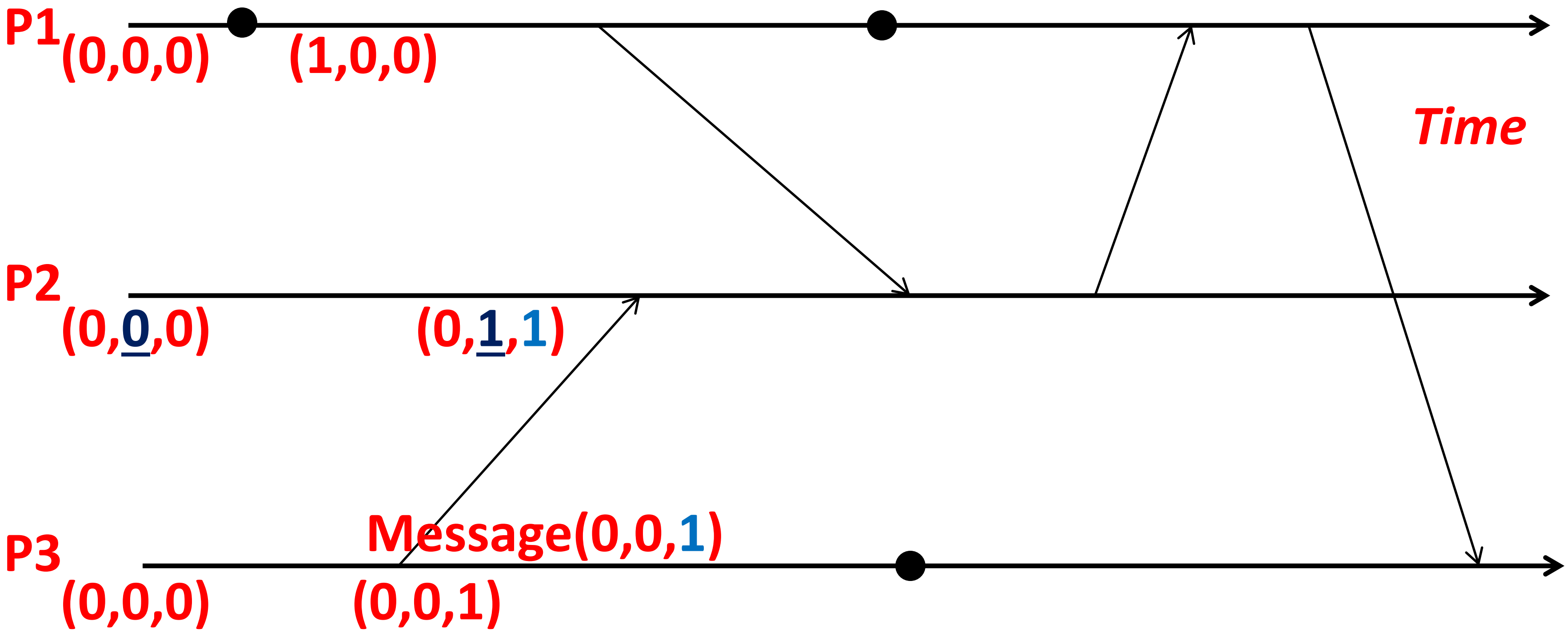


Initial counters (clocks)

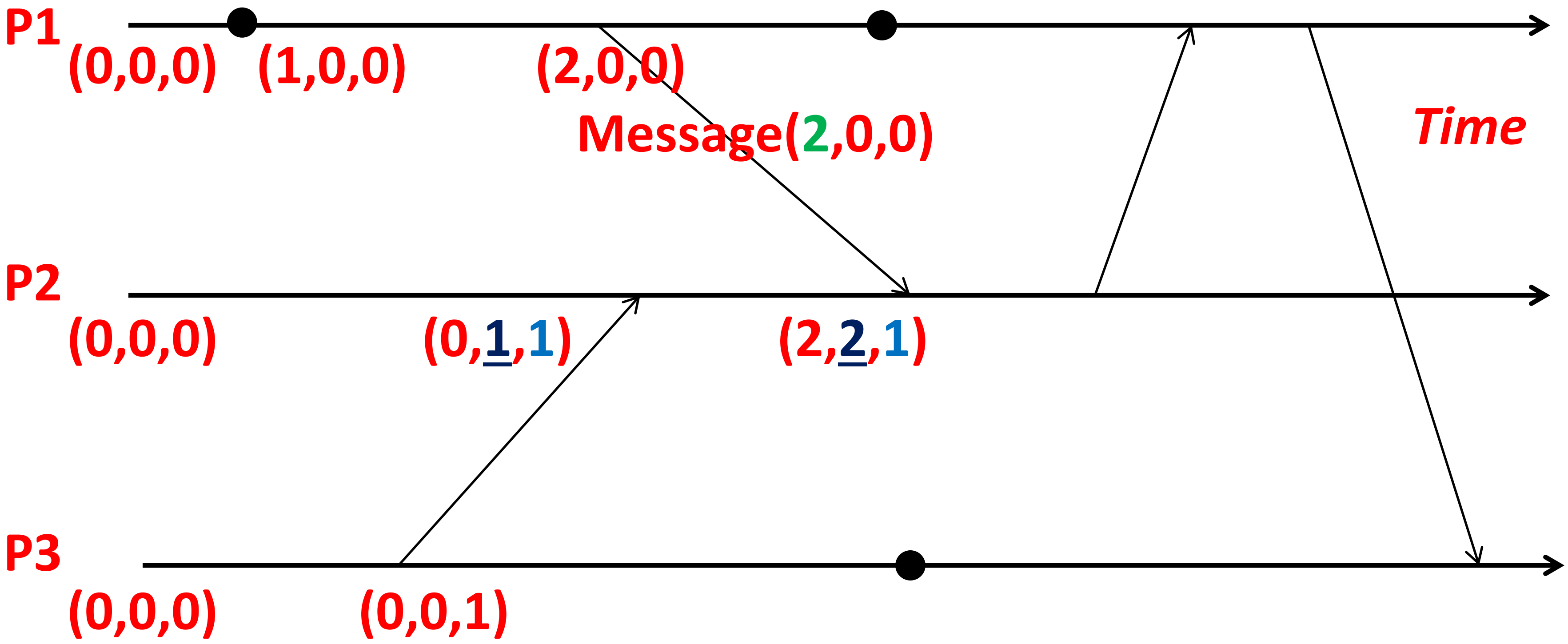
Vector Timestamps



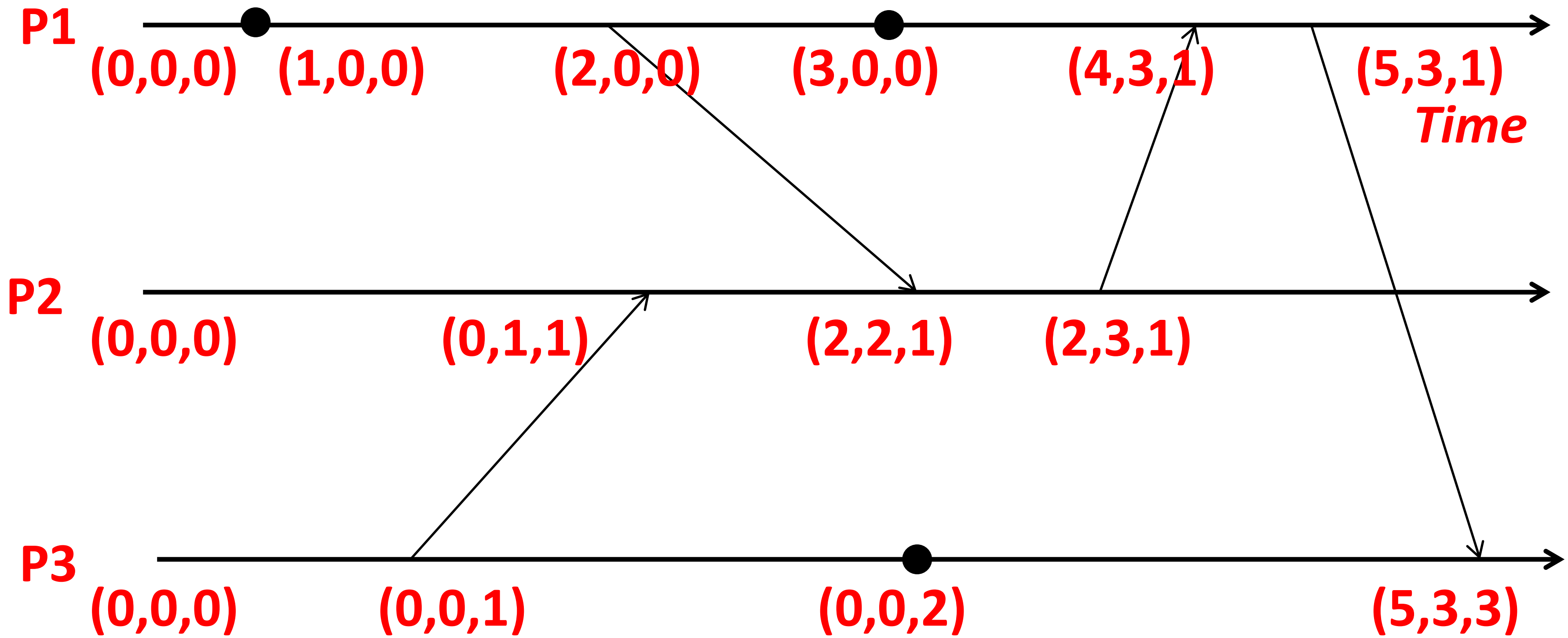
Vector Timestamps



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

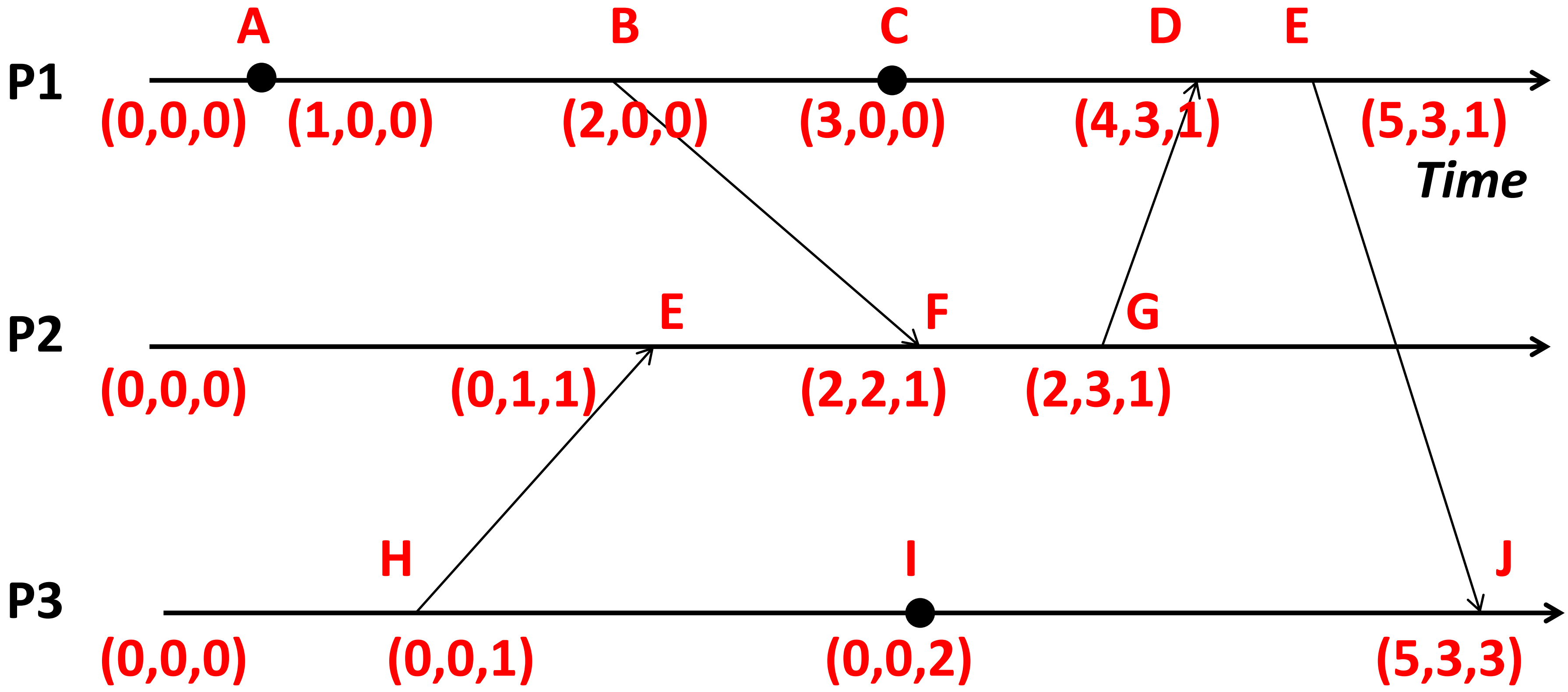
- Two events VT_1 and VT_2 are **concurrent**

  iff

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

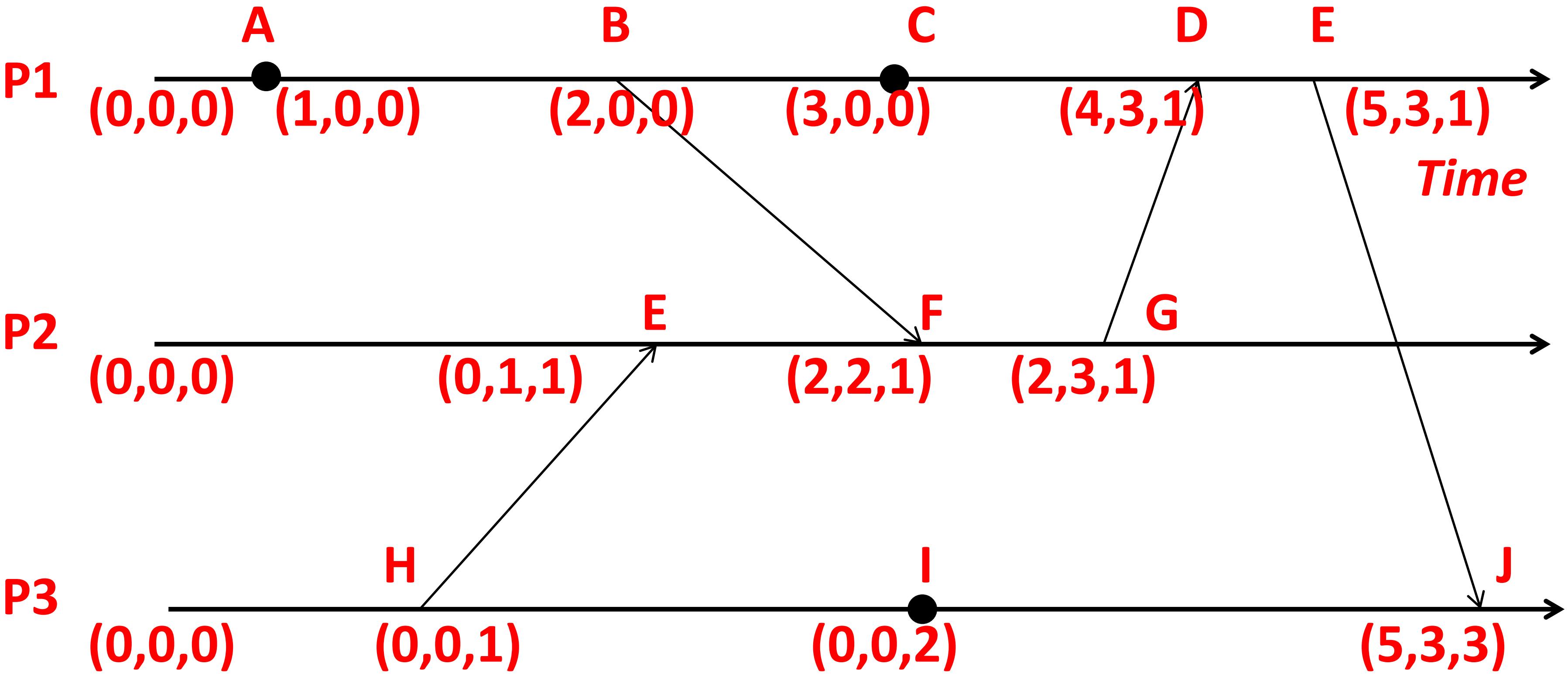
  We'll denote this as $VT_2 \; ||| \; VT_1$

Obeying Causality



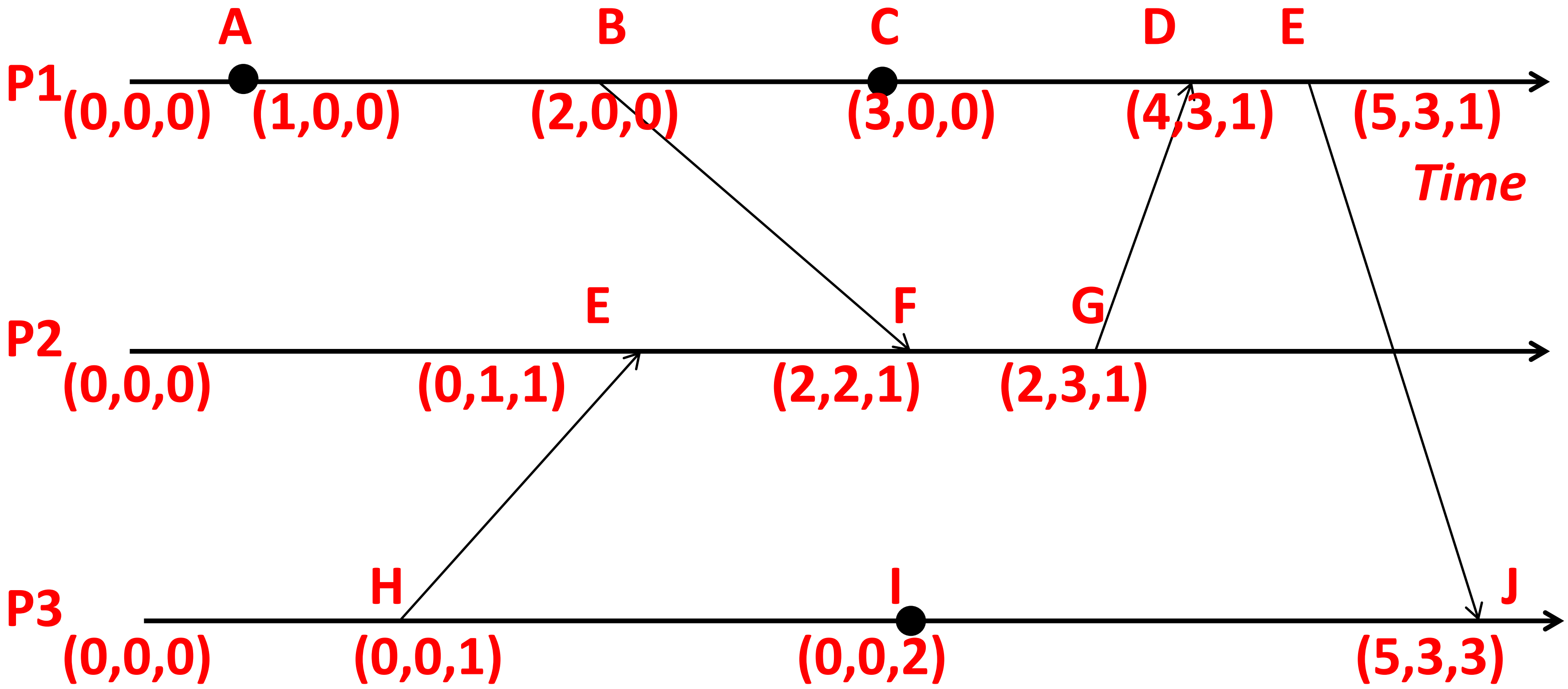
- A → B :: (1,0,0) < (2,0,0)
- B → F :: (2,0,0) < (2,2,1)
- A → F :: (1,0,0) < (2,2,1)

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

Summary : Logical Timestamps

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

Conclusion

- Clocks are unsynchronized in an asynchronous distributed system
- But need to order events across processes!

- **Time synchronization:**
  - Christian’s algorithm
  - Berkeley algorithm
  - NTP
  - DTP
  - But error a function of RTT

- **Can avoid time synchronization altogether by instead assigning logical timestamps to events**