Time in Distributed Systems

Prof. Nalini Venkatasubramanian

Distributed Computing Systems - Week 2

-includes slides/examples from

Indy Gupta (UIUC) and Kshemkalyani&Singhal (book slides)

The Concept of Time

- The Concept of Time
  - A standard time is a set of instants with a temporal precedence order $<$ satisfying certain conditions [Van Benthem 83]:
    - Transitivity
    - Irreflexivity
    - Linearity
    - Eternity ($\forall x \exists y: x<y$)
    - Density ($\forall x,y: x<y \rightarrow \exists z: x<z<y$)

  - Transitivity and Irreflexivity imply asymmetry
  - A linearly ordered structure may be insufficient to represent time in distributed systems..

Time and clocks (A real world example)

Cloud airline reservation system (with multiple servers A, B, C,...)

- 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 at C.

cf: Indy Gupta, CS 425, UIUC

Time

Global Time & Global States of Distributed Systems

- Asynchronous distributed systems consist of several *processes*
  - no common shared memory or global clock,
  - unpredictable processing delays
  - communicate (solely) via *messages* with unpredictable transmission delays
- Global time & global state are hard to realize
  - Rate of event occurrence may be very high
  - Event execution times may be very small
- We can only *approximate* the global view
  - *Simulate* a *synchronous* distributed system on an asynchronous system
  - *Simulate* a *global time* – Clocks (Physical and Logical)
  - *Simulate* a *global state* – Global Snapshots

Simulate Synchronous Distributed Systems

- *Synchronizers* [Awerbuch 85]
  - Simulate clock pulses in such a way that a message is only generated at a clock pulse and will be received before the next pulse
  - Drawback
    - Very high message overhead

Global time in distributed systems

- An accurate notion of global time is difficult to achieve in distributed systems.
  - Uniform notion of time is necessary for correct operation
  - Apps: Mission critical distributed control, online games/ entertainment, financial apps, smart environments
  - We often derive "causality" from loosely synchronized clocks

- Class Activity!!
  - Check wall clock, laptop clock (with sec setting), and mobile device clock (use timer)
  - Check half way ;
  - Repeat with network/GPS turned off...

Simulating global time

- Clocks in a distributed system drift
  - Relative to each other
  - Relative to a real world clock
    - Determination of this real world clock itself may be an issue

- Clock ***Skew*** versus ***Drift***
  - Clock Skew = Relative Difference in clock values of two processes
    - Like distance between two vehicles on a road
  - Clock Drift = Relative Difference in clock frequencies (rates) of two processes
    - Like difference in speeds between 2 vehicles on a road

.

Clock Synchronization

- Needed to simulate global time.

- A non-zero clock drift will cause skew to continuously increase
  - If faster device is ahead, it will drift away
  - If faster device is behind, it will catch up and then drift away
- ***Maximum Drift Rate (MDR)*** of a clock
  - Absolute MDR is defined relative to a Coordinated Universal Time (UTC)
  - MDR of a process depends on the environment.
  - Max 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

Clock Synchronization

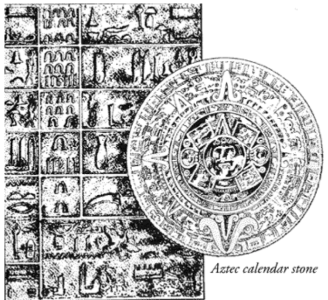
- Physical Clocks vs. Logical clocks

Physical Clock Synchronization

Physical Clocks

How do we measure real time?

- Early – Stonehenge, sundials
- 13th -17th century
    - Mechanical clocks based on astronomical measurements
- Solar Day - Transit of the sun
- Solar Seconds - Solar Day/(3600*24)



Problem (1940): Rotation of earth varies!

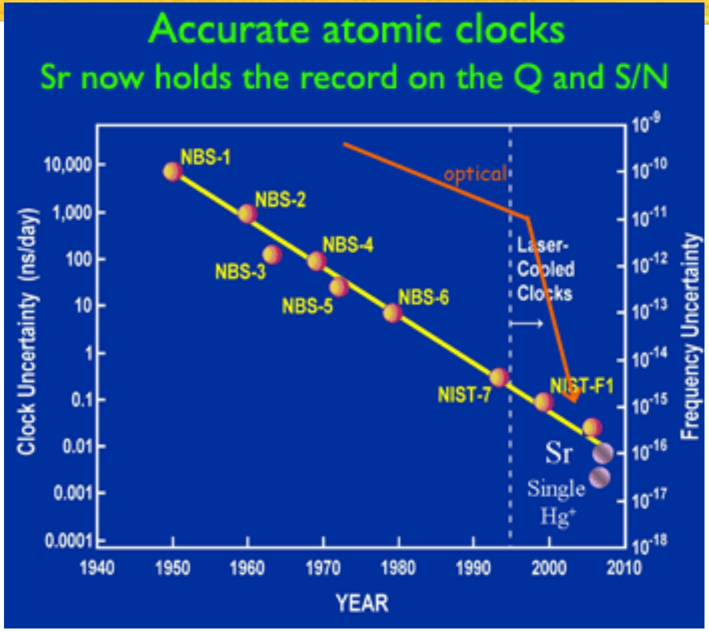
Mean solar second = average over many days

Date	Duration in mean solar time
February 11	24 hours
March 26	24 hours – 18.1 sec
May 14	24 hours
June 19	24 hours + 13.1 sec
July 26	24 hours
September 16	24 hours – 21.3 sec
November 3	24 hours
December 22	24 hours + 29.9 sec

Length of apparent solar day (1998)
– (*cf: wikipedia*)

Atomic Clocks

- 1948 - Counting transitions of a crystal (Cesium 133, quartz) used as atomic clock
  - crystal oscillates at a well known frequency
- 2014 – NIST-F2 Atomic clock
  - Accuracy: ± 1 sec in 300 mil years
  - NIST-F2 measures particular transitions in Cesium atom (9,192,631,770 vibrations per second), in much colder environment, minus 316F, than NIST-F1
- TAI - International Atomic Time
  - 9,192,631,779 transitions = 1 mean solar second in 1948

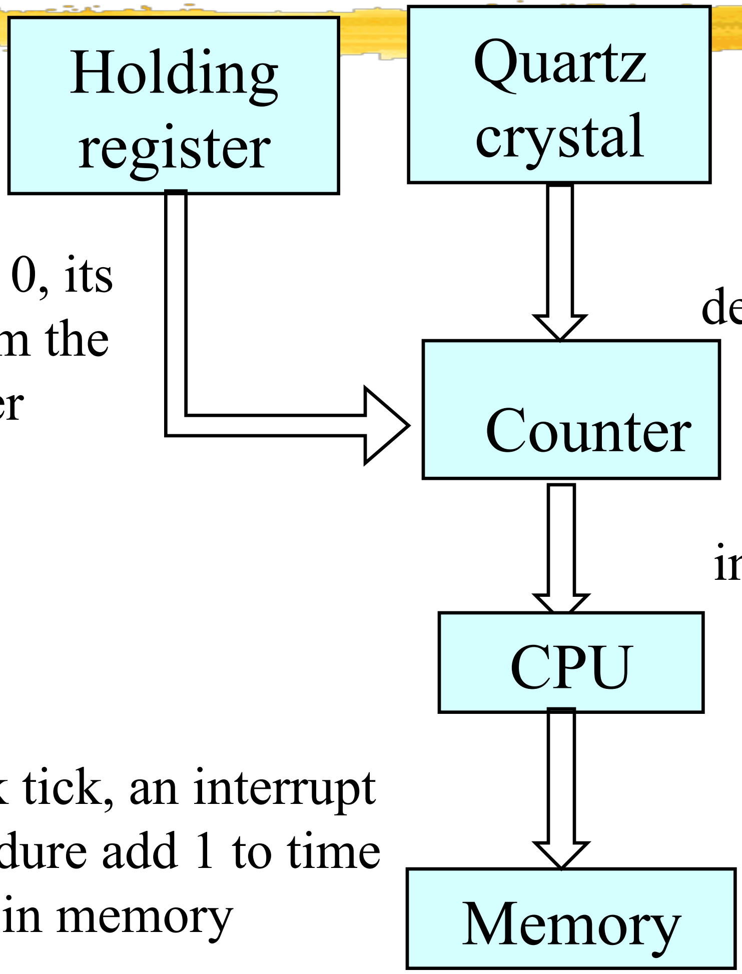


UTC (Universal Coordinated Time)
From time to time, UTC skips a solar second to stay in phase with the sun (30+ times since 1958)

UTC is broadcast by several sources (satellites…)

Next Generation Atomic Clocks -- NIST F2

How Clocks Work in Computers



Oscillation at a well-defined frequency

Each crystal oscillation decrements the counter by 1

When counter gets 0, its value reloaded from the holding register

When counter is 0, an interrupt is generated, which is call a **clock tick**

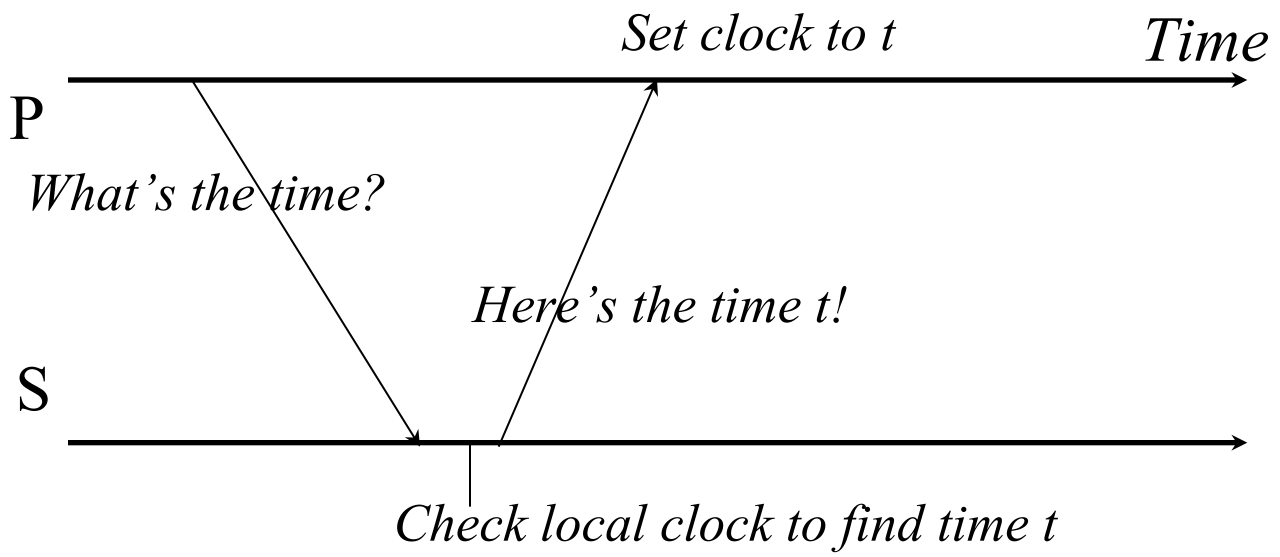
At each clock tick, an interrupt service procedure add 1 to time stored in memory

Accuracy of Computer Clocks

- Modern timer chips (RTCs) have a relative error of 1/100,000 – (~1 - 8 sec a day)
- To maintain synchronized clocks
  - External Synchronization
    - Can use UTC source (time server) to obtain current notion of time
  - Internal Synchronization
    - Use solutions without UTC.

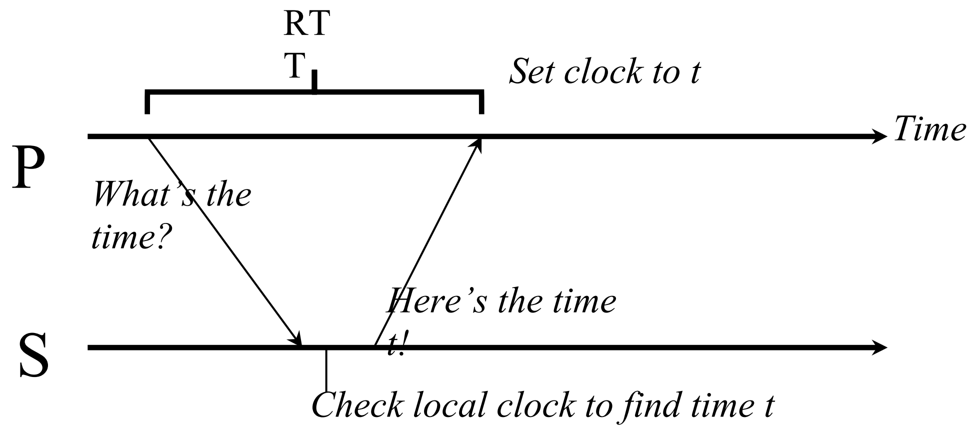
Cristian's (Time Server) Algorithm (external synchronization)

- Uses a *time server* (S) to synchronize clocks
  - Time server keeps the reference time (say UTC)
  - A client asks the time server for time, the server responds with its current time, and the client uses the received value to set its clock.



Cristian's Algorithm (cont.)

- But network round-trip time introduces errors…
  - By the time response message is received at P, time has moved on
  - Let **RTT = response-received-time – request-sent-time** (measurable at client),
  - If we know (a) min = minimum client-server one-way transmission time and (b) that the server timestamped the message at the last possible instant before sending it back
  - Then, the actual time could be between **[T+min,T+RTT— min]**



Cristian's Algorithm (cont.)

- Client sets its clock to halfway between T+min and T +RTT— min i.e., at T+RTT/2
  - Expected (i.e., average) skew in client clock time = (RTT/2 – *min*)
- Can increase clock value, should never decrease it.
- Can adjust speed of clock too (either up or down is ok)
- Multiple requests to increase accuracy
  - For unusually long RTTs, repeat the time request
  - For non-uniform RTTs
    - Drop values beyond threshold; Use averages (or weighted average)

Berkeley UNIX algorithm (Internal Synchronization)

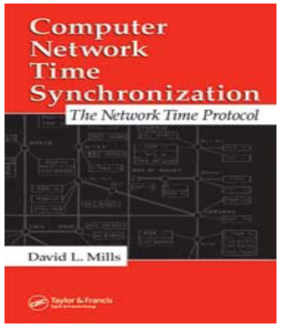
- One Version
  - One daemon without UTC
  - Periodically, this daemon polls and asks all the machines for their time
  - The machines respond.
  - The daemon computes an average time and then broadcasts this average time.
- Another Version
  - Master/daemon uses Cristian's algorithm to calculate time from multiple sources, removes outliers, computes average and broadcasts

Decentralized Averaging Algorithm (Internal Synchronization)

- Each machine has a daemon without UTC
- Periodically, at fixed agreed-upon times, each machine broadcasts its local time.
- Each of them calculates the average time by averaging all the received local times.

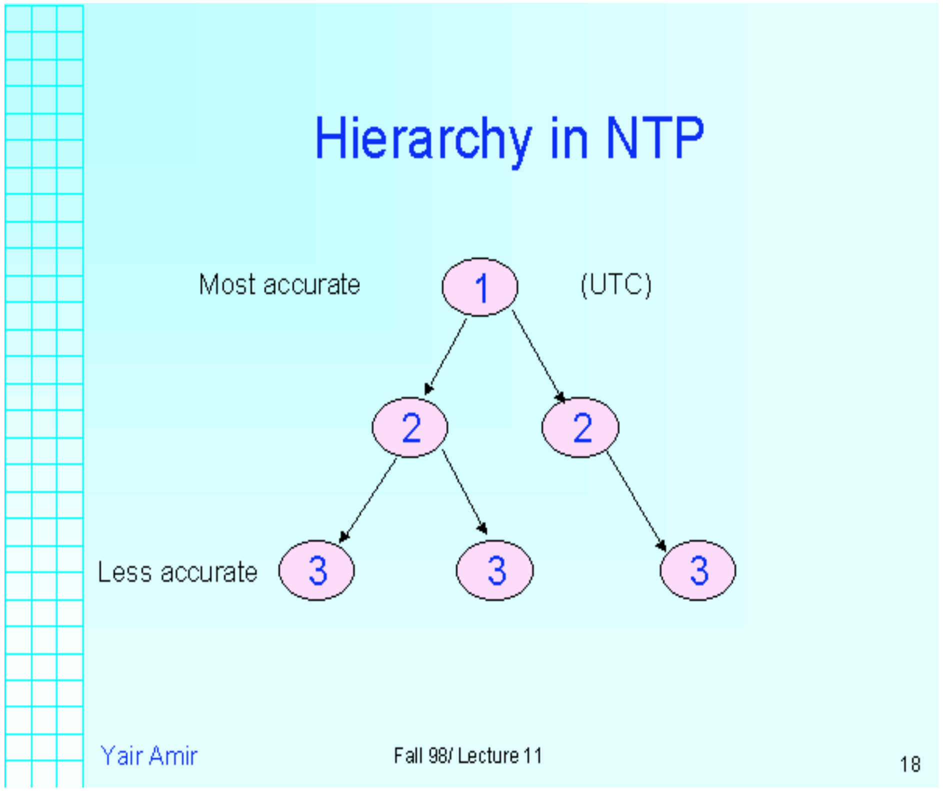
Network Time Protocol (NTP)

- Most widely used physical clock synchronization protocol on the Internet (http://www.ntp.org)
  - Currently used: NTP V3 and V4
- 10-20 million NTP servers and clients in the Internet
- Claimed Accuracy (Varies)
  - milliseconds on WANs, submilliseconds on LANs, submicroseconds using a precision timesource
  - Nanosecond NTP in progress

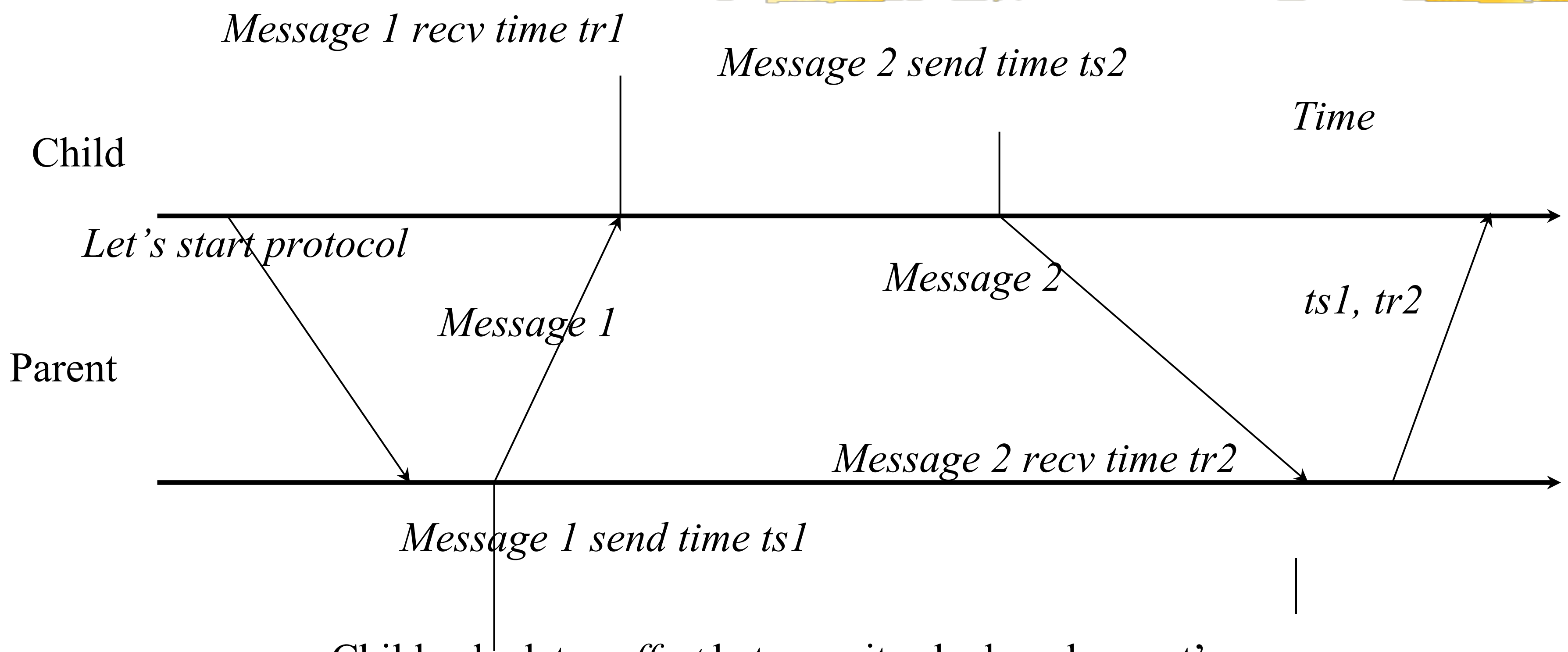

Computer Network Time Synchronization
The Network Time Protocol
David L. Mills
Taylor & Francis

NTP Design

- Hierarchical tree of time servers.
  - The primary server at the root synchronizes with the UTC.
  - The next level contains secondary servers, which act as a backup to the primary server.
  - At the lowest level is the synchronization subnet which has the clients.
  - Variant of Cristian's algorithm that does not use RTT's, but multiple 1-way messages



NTP Protocol - Determining Error



- 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

NTP Protocol - Determining 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**

  $tr1 = ts1 + L1 + oreal$

  $tr2 = ts2 + L2 - oreal$

- **Subtracting second equation from the first**

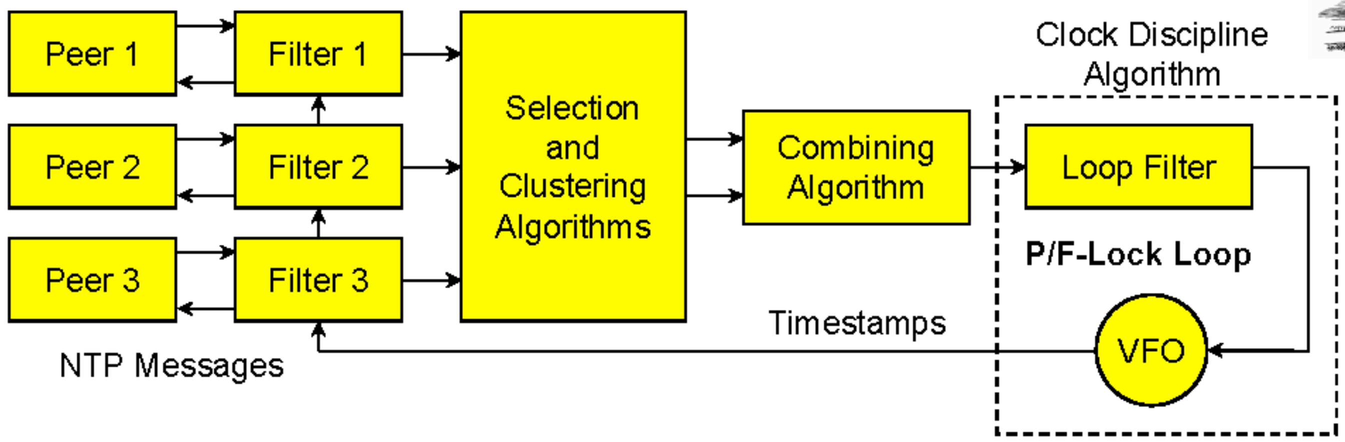
  $oreal = (tr1 - tr2 + ts2 - ts1)/2 + (L2 - L1)/2 \Rightarrow oreal = o + (L2 - L1)/2$

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

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

We still have a non-zero error! Will exist as long as message latency exists!

NTP architecture overview

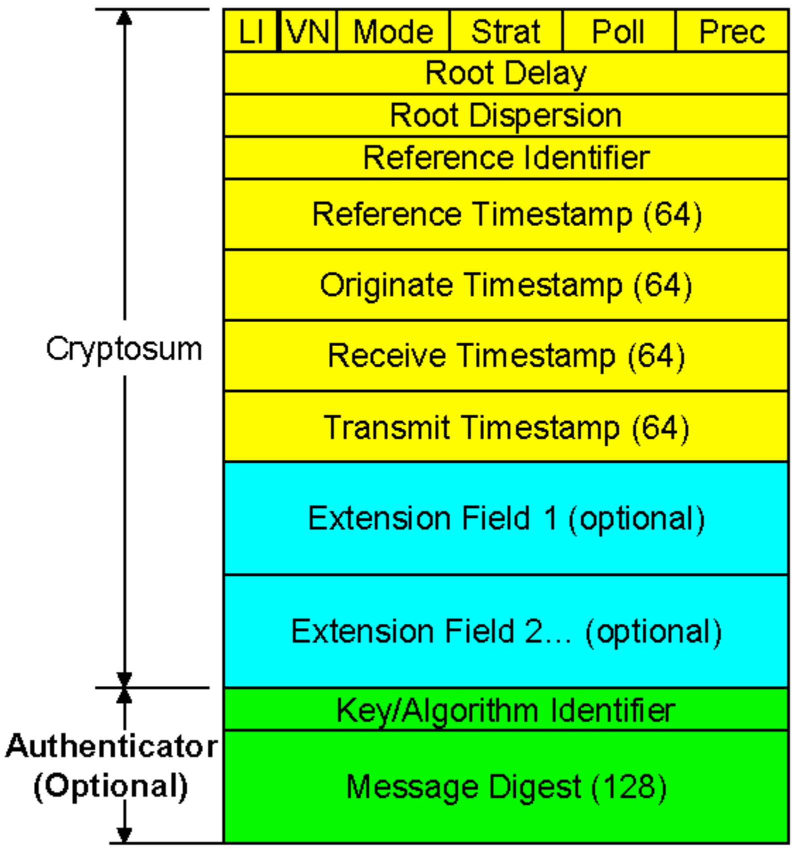


- Multiple servers/peers provide redundancy and diversity.
- Clock filters select best from a window of eight time offset samples.
- Intersection and clustering algorithms pick best *truechimers* and discard *falsetickers.*
- Combining algorithm computes weighted average of time offsets.
- Loop filter and variable frequency oscillator (VFO) implement hybrid phase/frequency-lock (P/F) feedback loop to minimize jitter and wander.

From (http://www.ece.udel.edu/~mills/database/brief/seminar/ntp.pdf)

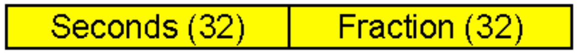
NTP protocol header and timestamp formats

NTP Protocol Header Format (32 bits)



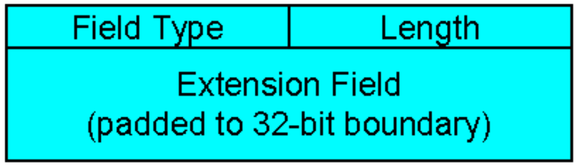
LI	leap warning indicator
VN	version number (4)
Strat	stratum (0-15)
Poll	poll interval (log2)
Prec	precision (log2)

NTP Timestamp Format (64 bits)

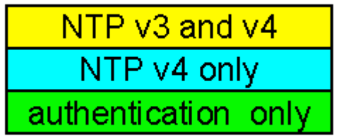


Value is in seconds and fraction since 0h 1 January 1900

NTP v4 Extension Field



Last field padded to 64-bit boundary



Authenticator uses MD5 cryptosum
of NTP header plus extension fields (NTPv4)

From (http://www.ece.udel.edu/~mills/database/brief/seminar/ntp.pdf)

Logical Clock Synchronization

Ordering Events in a Distributed System

- Trying to sync physical clocks is one approach.
- What if we instead assigned timestamps to events that were not *absolute* time?

- Timestamps must obey *causality* to preserve event ordering
  - 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 Clocks

- Used to determine causality in distributed systems
- Time is represented by non-negative integers
- Event structures represent distributed computation (in an abstract way)
  - A process can be viewed as consisting of a sequence of events, where an event is an atomic transition of the local state which happens in no time
  - Process Actions can be modeled using the 3 types of events
    - Send Message
    - Receive Message
    - Internal (change of state)

Causal Relations

- Distributed application results in a set of distributed events
  - Induces a partial order [illegible] causal precedence relation
- Knowledge of this causal precedence relation is useful in reasoning about and analyzing the properties of distributed computations
  - Liveness and fairness in mutual exclusion
  - Consistency in replicated databases
  - Distributed debugging, checkpointing

Event Ordering

- Lamport defined the “happens before” (=>) relation
  - If a and b are events in the same process, and a occurs before b, then a => b.
  - 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.
  - If X =>Y and Y=>Z then X => Z.

If a => b then time (a) => time (b)

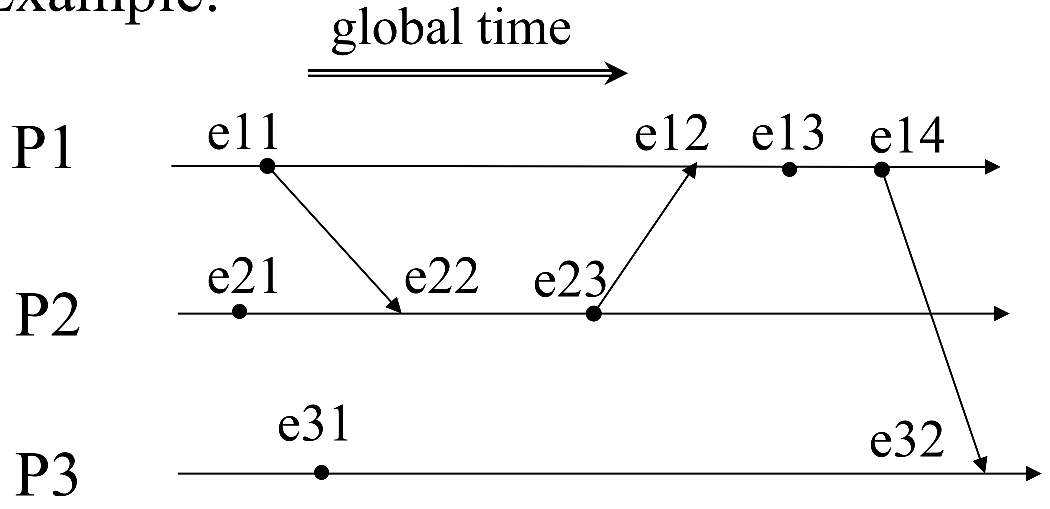
Event Ordering- an example

Processor Order: e precedes e′ in the same process

Send-Receive: e is a send and e′ is the corresponding receive

Transitivity: exists e″ s.t. e < e″ and e″< e′

Example:



Program order:	e13 < e14
Send-Receive:	e23 < e12
Transitivity:	e21 < e32

Causal Ordering

- “Happens Before” also called causal ordering
- Possible to draw a causality relation between 2 events if
  - They happen in the same process
  - There is a chain of messages between them
- “Happens Before” notion is not straightforward in distributed systems
  - No guarantees of synchronized clocks
  - Communication latency

Logical Clocks

- A logical Clock C is some abstract mechanism which assigns to any event $e \in E$ the value $C(e)$ of some time domain T such that certain conditions are met
  - $C: E \rightarrow T$ :: T is a partially ordered set : $e < e' \rightarrow C(e) < C(e')$ holds

- Consequences of the clock condition [Morgan 85]:
  - Events occurring at a particular process are totally ordered by their local sequence of occurrence
    - If an event e occurs before event e' at some single process, then event e is assigned a logical time earlier than the logical time assigned to event e'
  - For any message sent from one process to another, the logical time of the send event is always earlier than the logical time of the receive event
    - Each receive event has a corresponding send event
    - Future can not influence the past (causality relation)

Implementation of Logical Clocks

- Requires
  - Data structures local to every process to represent logical time and
  - A protocol to update the data structures to ensure the consistency condition.
- Each process Pi maintains data structures that allow it the following two capabilities:
  - A local logical clock, denoted by LC_i , that helps process Pi measure its own progress.
  - A logical global clock, denoted by GCi , that is a representation of process Pi 's local view of the logical global time. Typically, lci is a part of gci
- The protocol ensures that a process's logical clock, and thus its view of the global time, is managed consistently.
  - The protocol consists of the following two rules:
    - R1: This rule governs how the local logical clock is updated by a process when it executes an event.
    - R2: This rule governs how a process updates its global logical clock to update its view of the global time and global progress.

Types of Logical Clocks

- Systems of logical clocks differ in their representation of logical time and also in the protocol to update the logical clocks.
- 3 kinds of logical clocks
  - Scalar
  - Vector
  - Matrix

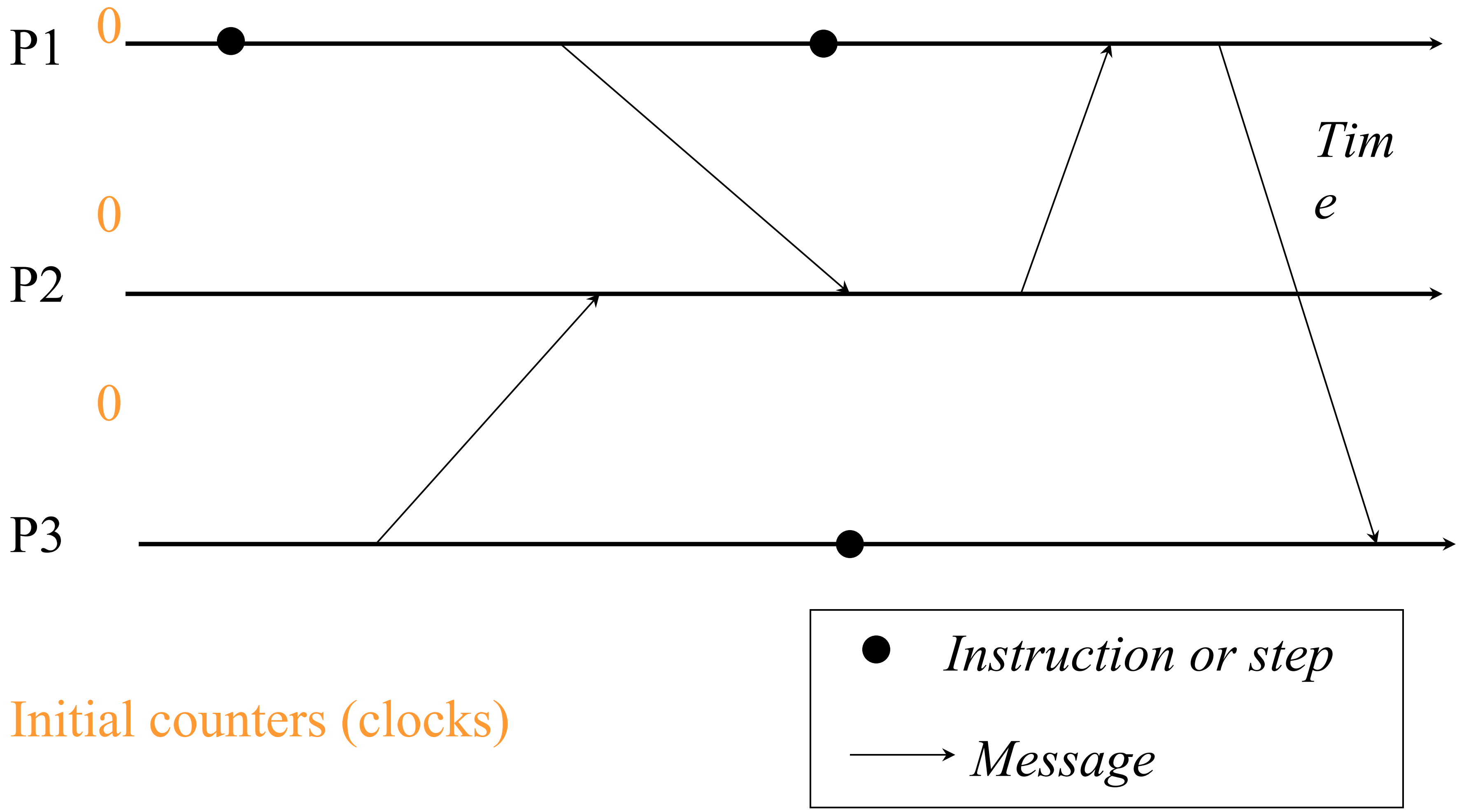
Scalar Logical Clocks - Lamport

- Proposed by Lamport in 1978 as an attempt to totally order events in a distributed system.
- Time domain is the set of non-negative integers.
- The logical local clock of a process pi and its local view of the global time are squashed into one integer variable Ci .
- Monotonically increasing counter
  - No relation with real clock
- Each process keeps its own logical clock used to timestamp events

Consistency with Scalar Clocks

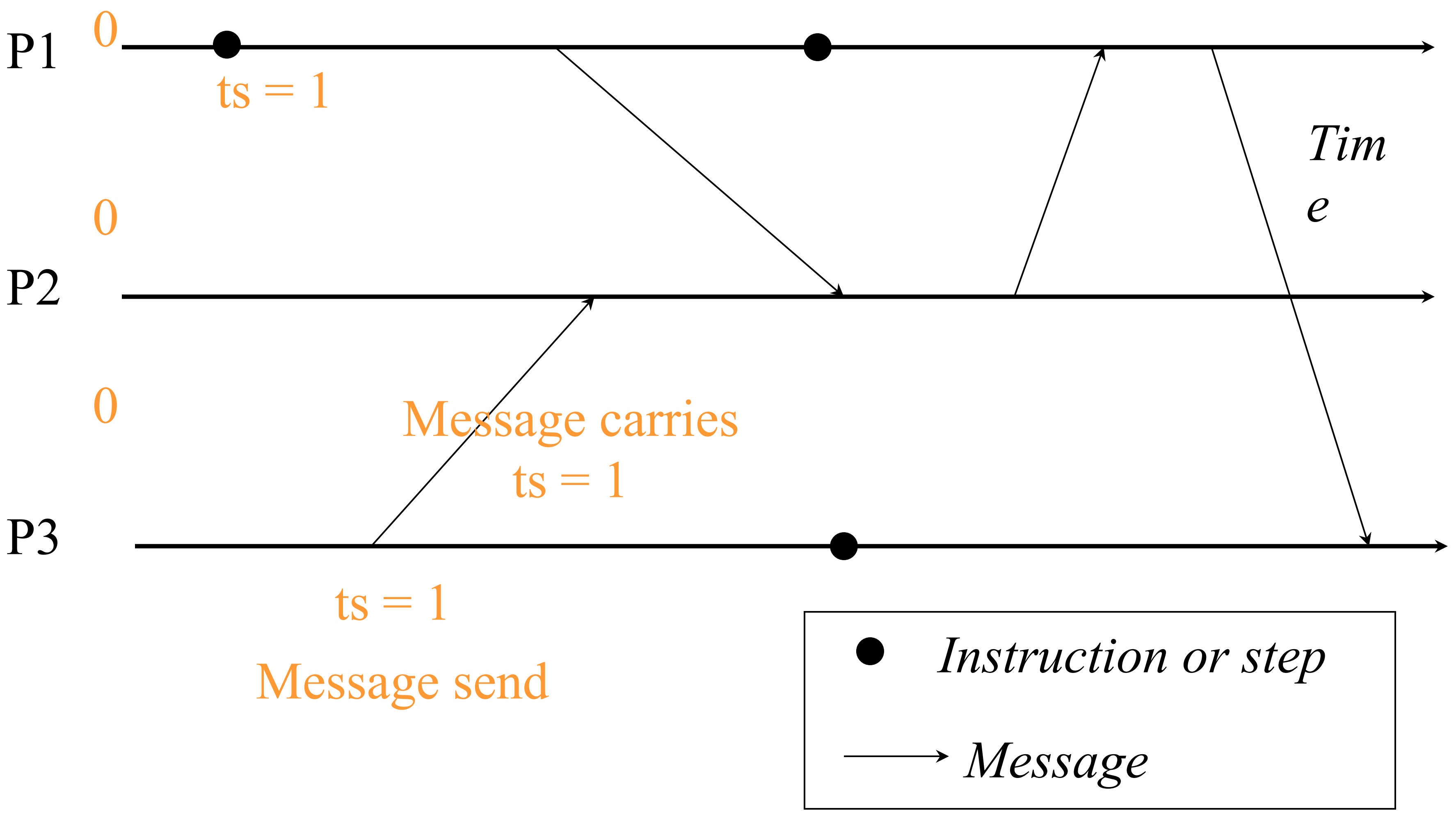
- To guarantee the clock condition, local clocks must obey a simple protocol:
  - When executing an internal event or a send event at process P_i the clock C_i ticks
    - C_i += d   (d>0)
  - When P_i sends a message m, it piggybacks a logical timestamp t which equals the time of the send event
  - When executing a receive event at P_i where a message with timestamp *t* is received, the clock is advanced
    - $C_i = \max(C_i, t) + d$ (d>0)
- Results in a partial ordering of events.

Lamport Timestamps

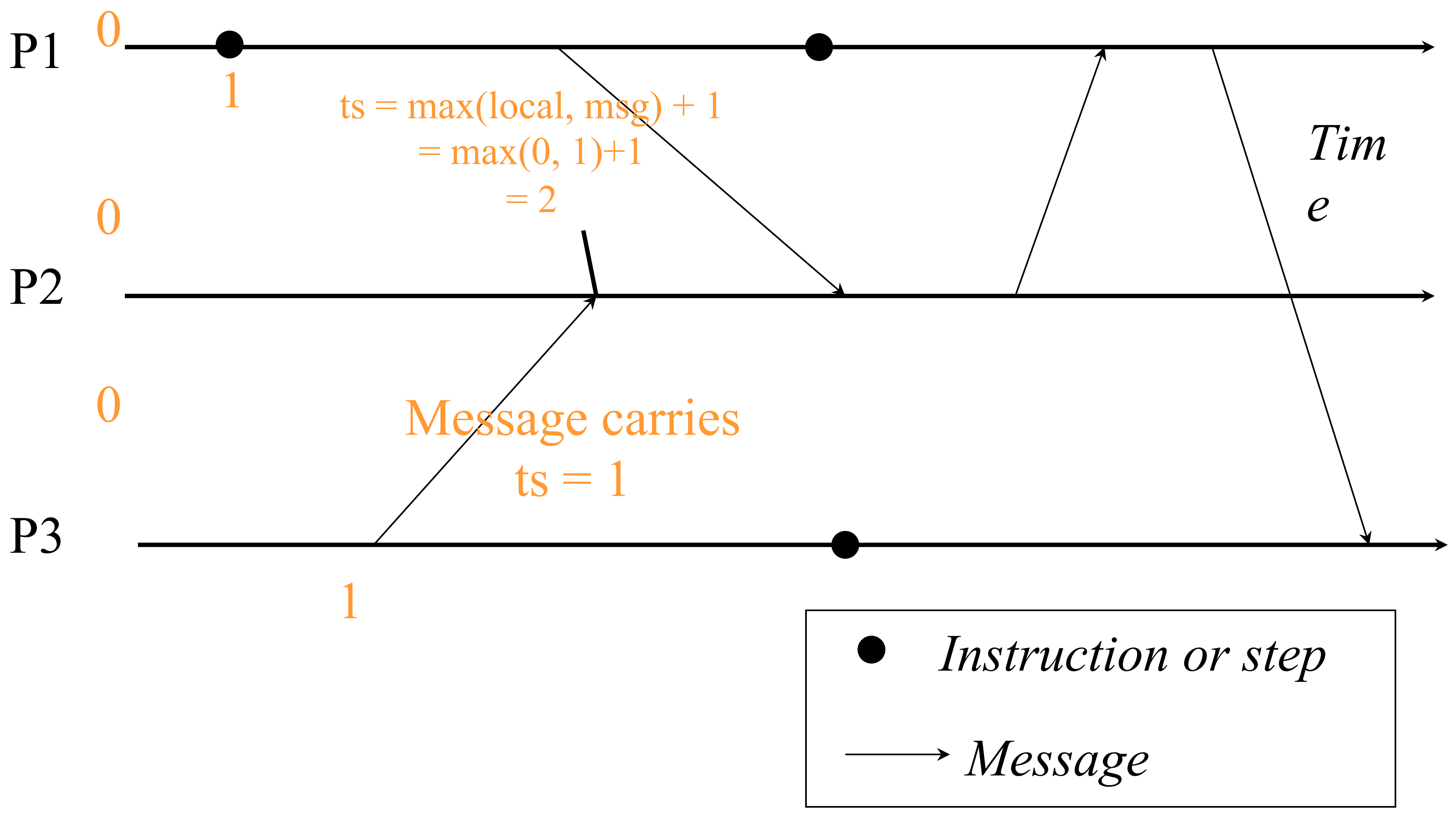


Initial counters (clocks)

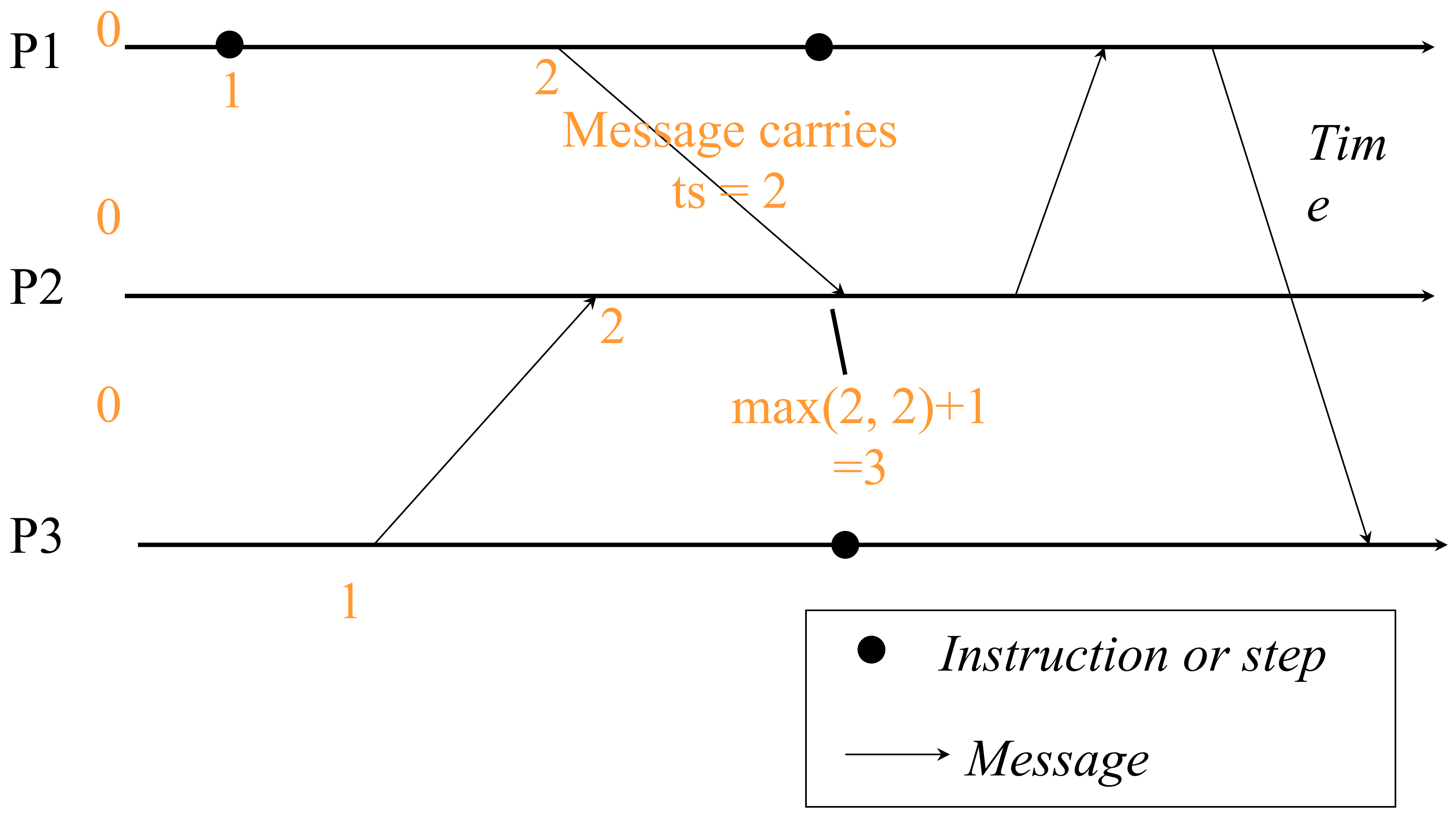
Lamport Timestamps



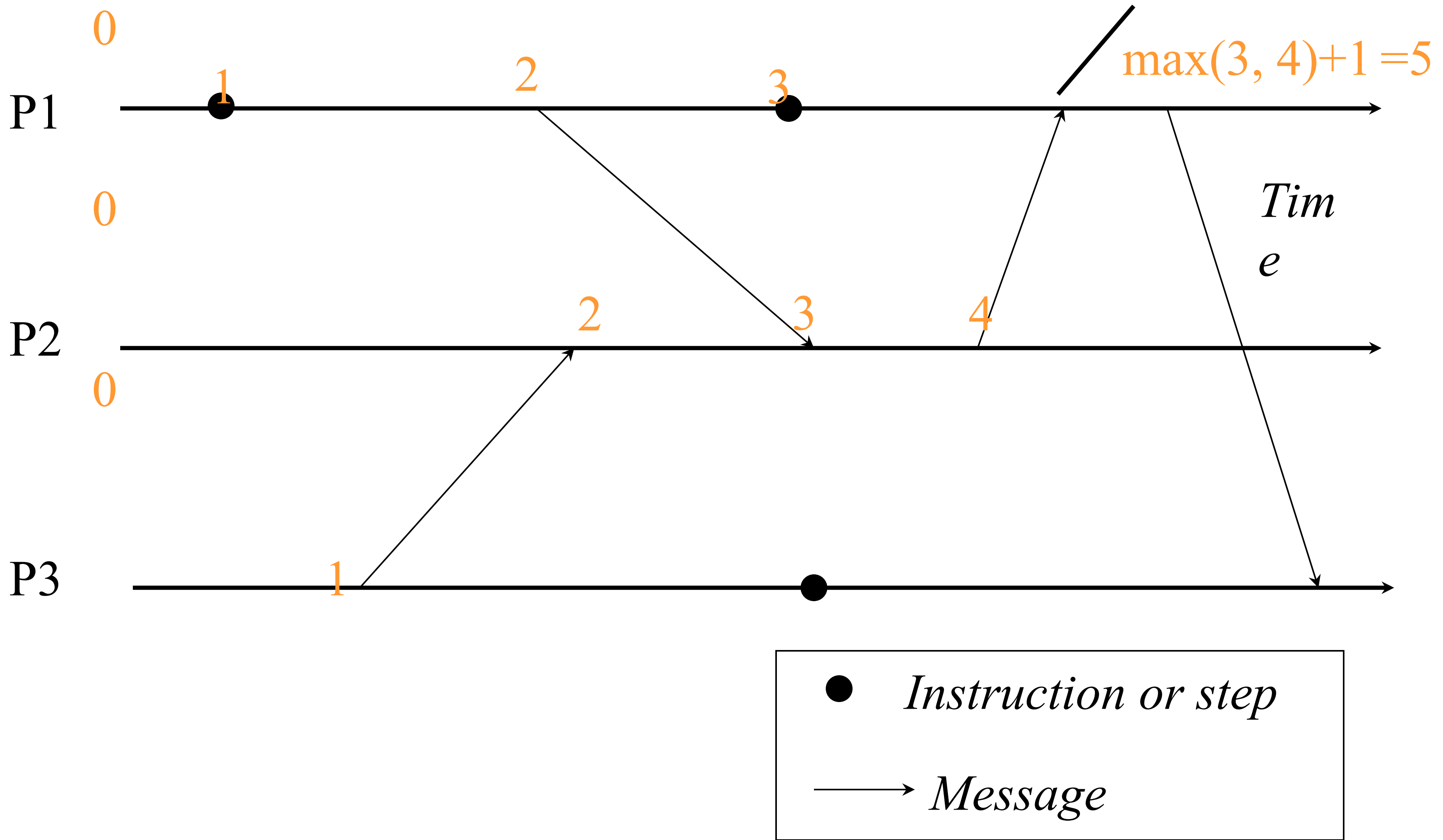
Lamport Timestamps



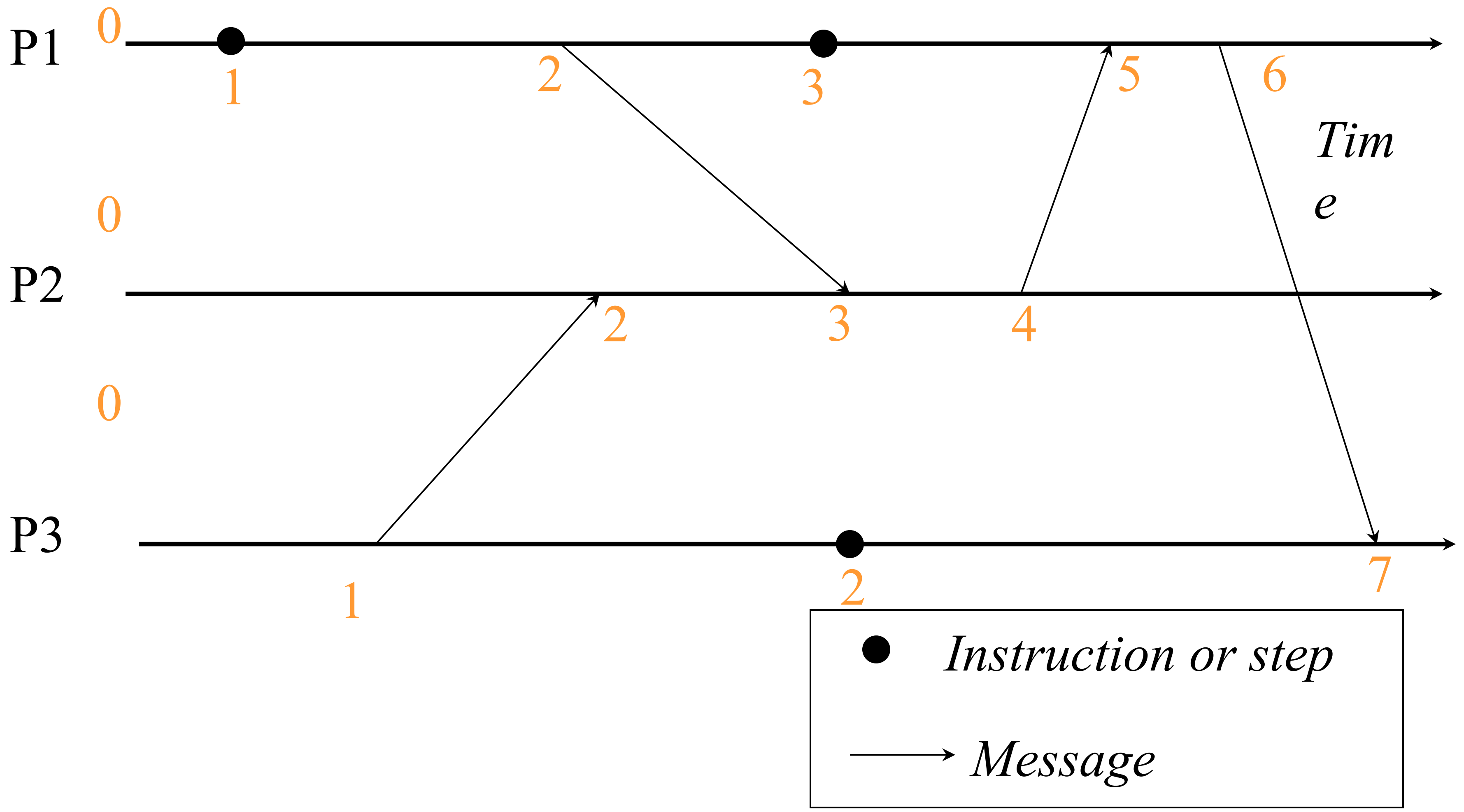
Lamport Timestamps



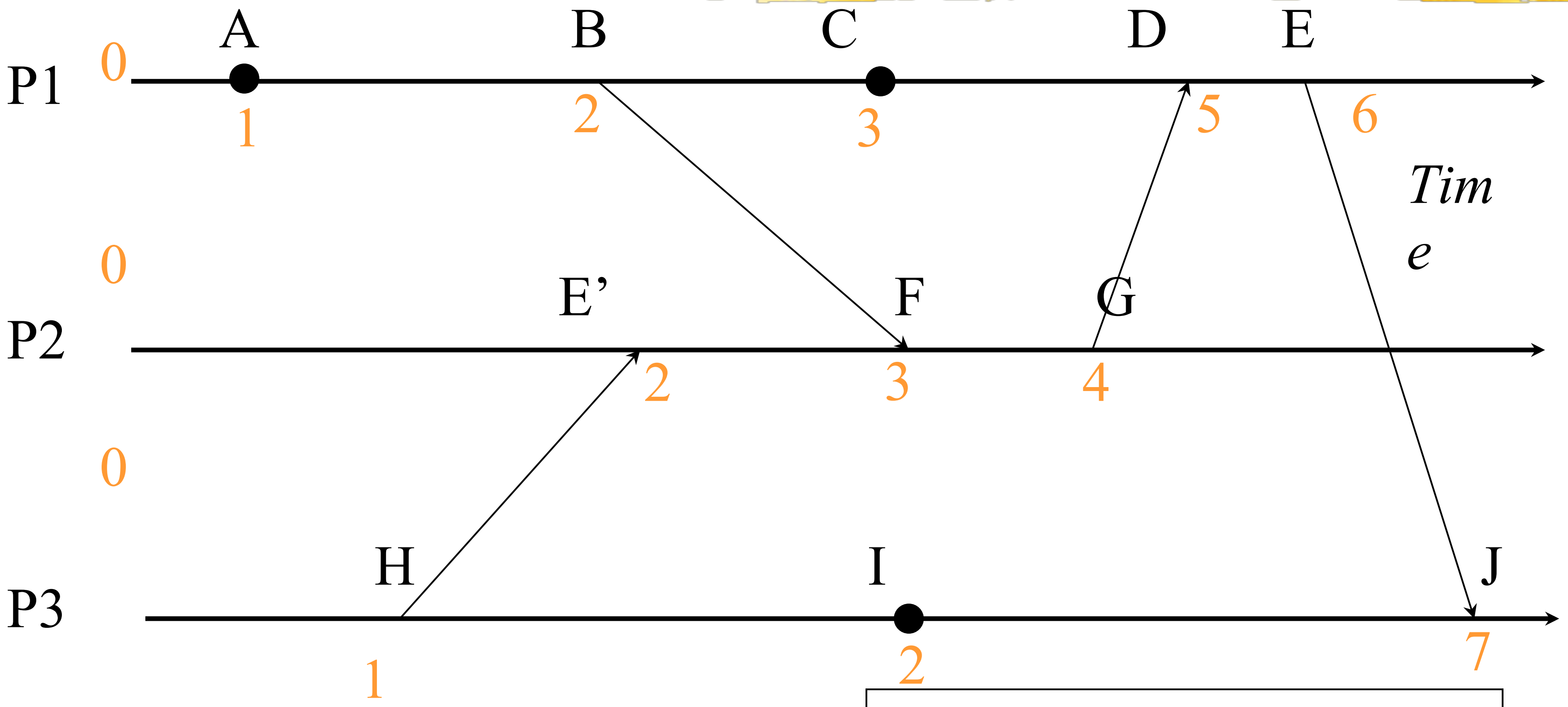
Lamport Timestamps



Lamport Timestamps



Obeying Causality

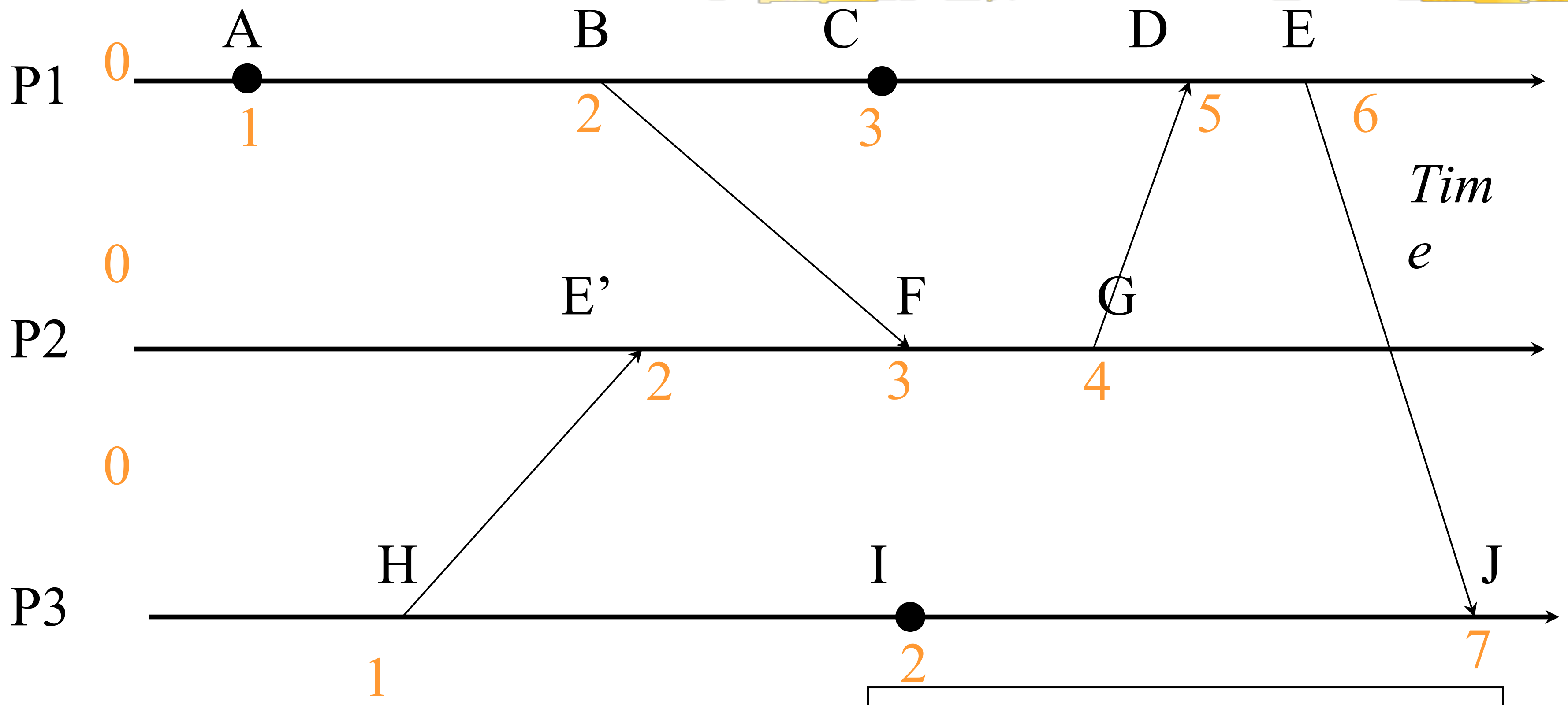


- A ① B :: 1 < 2
- B ① F :: 2 < 3
- A ① F :: 1 < 3

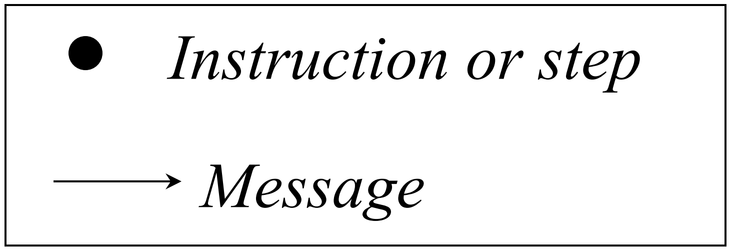
● *Instruction or step*

→ *Message*

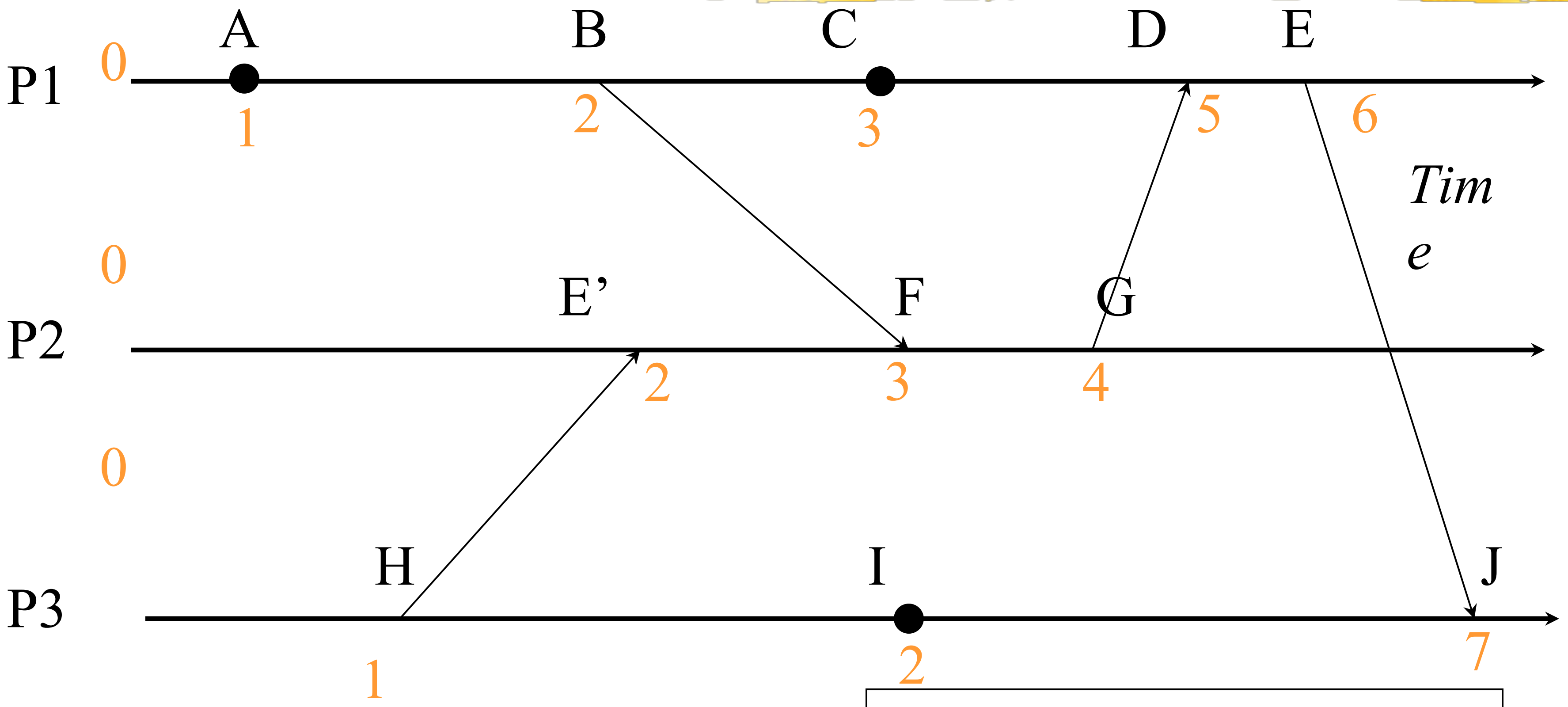
Obeying Causality (2)



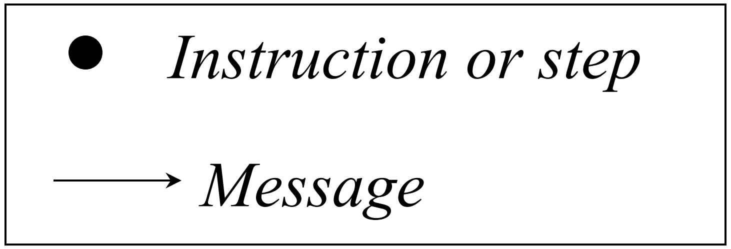
- H ① G :: 1 < 4
- F ① J :: 3 < 7
- H ① J :: 1 < 7
- C ① J :: 3 < 7



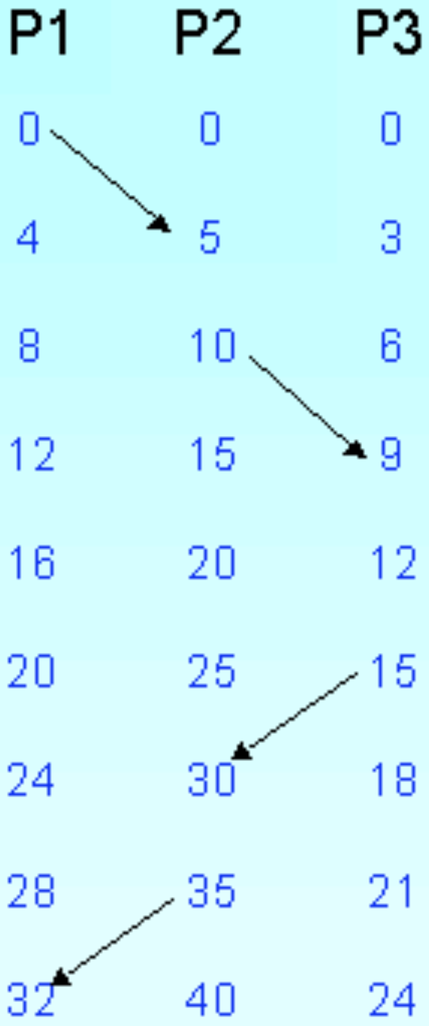
Not always *implying* Causality



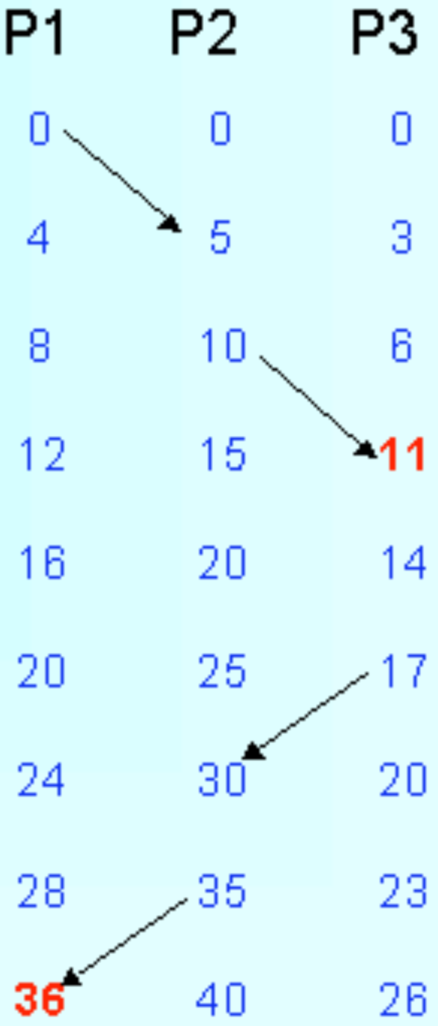
- ? C ① F ? :: 3 = 3
- ? H ① C ? :: 1 < 3
- (C, F) and (H, C) are pairs of *concurrent* events



Lamport Logical Clock



P1	P2	P3
0	0	0
4	5	3
8	10	6
12	15	9
16	20	12
20	25	15
24	30	18
28	35	21
32	40	24



P1	P2	P3
0	0	0
4	5	3
8	10	6
12	15	**11**
16	20	14
20	25	17
24	30	20
28	35	23
36	40	26

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}

Total Ordering

- Extending partial order to total order

time	Proc_id

- Global timestamps:
  - (Ta, Pa) where Ta is the local timestamp and Pa is the process id.
  - (Ta,Pa) < (Tb,Pb) iff
    - (Ta < Tb) or ((Ta = Tb) and (Pa < Pb))
  - Total order is consistent with partial order.

Properties of Scalar Clocks

- Event counting
  - If the increment value d is always 1, the scalar time has the following interesting property: if event e has a timestamp h, then h-1 represents the minimum logical duration, counted in units of events, required before producing the event e;
  - We call it the height of the event e.
  - In other words, h-1 events have been produced sequentially before the event e regardless of the processes that produced these events.

Properties of Scalar Clocks

- No Strong Consistency
- The system of scalar clocks is not strongly consistent; that is, for two events ei and ej , C(ei) < C(ej) does not imply ei → ej .
- Reason: In scalar clocks, logical local clock and logical global clock of a process are squashed into one, resulting in the loss of causal dependency information among events at different processes.

Independence

- Two events e,e′ are mutually independent (i.e. e||e′) if ~(e<e′)∧~(e′<e)
  - Two events are independent if they have the same timestamp
  - Events which are causally independent may get the same or different timestamps
- By looking at the timestamps of events it is not possible to assert that some event *could not* influence some other event
  - If C(e)<C(e′) then ~(e′<e) *however*, it *is not possible* to decide whether e<e′ or e||e′
  - C is an order *homomorphism* which preserves < but it does not preserves negations (i.e. obliterates a lot of structure by mapping E into a linear order)

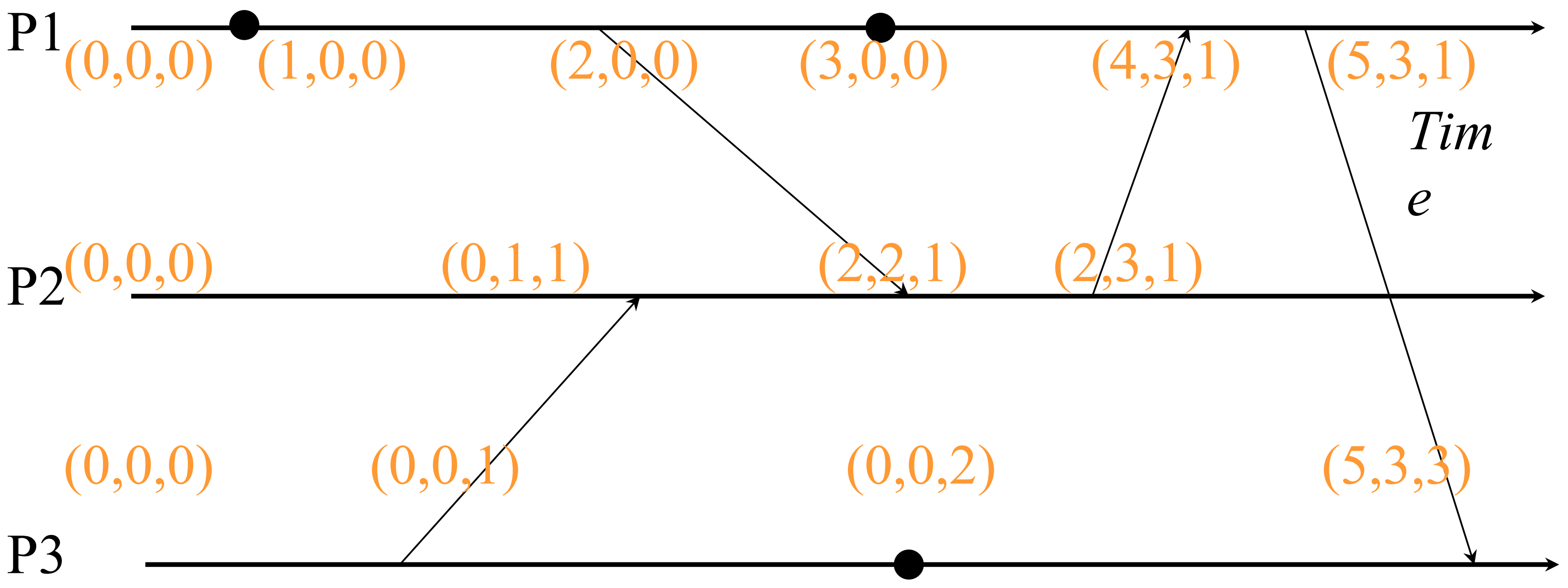
Problems with Total Ordering

- A linearly ordered structure of time is not always adequate for distributed systems
  - captures dependence of events
  - loses independence of events - artificially enforces an ordering for events that need not be ordered – loses information
- Mapping partial ordered events onto a linearly ordered set of integers is *losing information*
    - Events which may happen simultaneously may get different timestamps as if they happen in some definite order.
- A partially ordered system of *vectors* forming a *lattice* structure is a natural representation of time in a distributed system

Vector Clocks

- Independently developed by Fidge, Mattern and Schmuck.
- Aim: To construct a mechanism by which each process gets an optimal approximation of global time
- Time representation
  - Set of n-dimensional non-negative integer vectors.
  - Each process has a clock C_i consisting of a vector of length *n*, where *n* is the total number of processes vt[1..n], where vt[j] is the local logical clock of Pj and describes the logical time progress at process Pj .
  - A process P_i ticks by incrementing its own component of its clock
    - $C_i[i] += 1$
  - The timestamp C(e) of an event e is the clock value after ticking
  - Each message gets a piggybacked timestamp consisting of the vector of the local clock
    - The process gets some knowledge about the other process' time approximation
    - $C_i = \sup(C_i, t) :: \sup(u,v) = w : w[i] = \max(u[i], v[i]), \forall i$

Vector Timestamps



- $VT_1 = VT_2$,

  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 $VT_1 \leq VT_2$ &

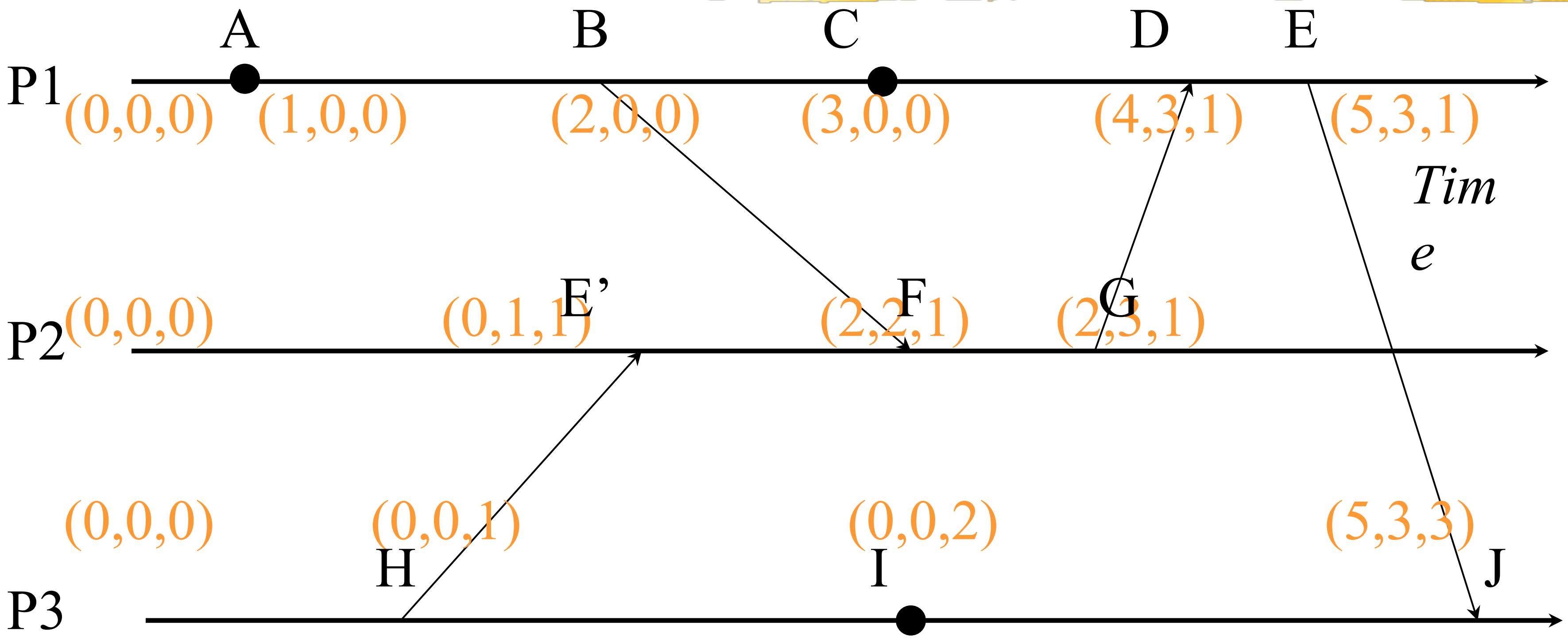
  there exists j such that

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

- Two events VT_1 and VT_2 are concurrent
  iff
  NOT $(VT_1 \leq VT_2)$ AND NOT $(VT_2 \leq VT_1)$

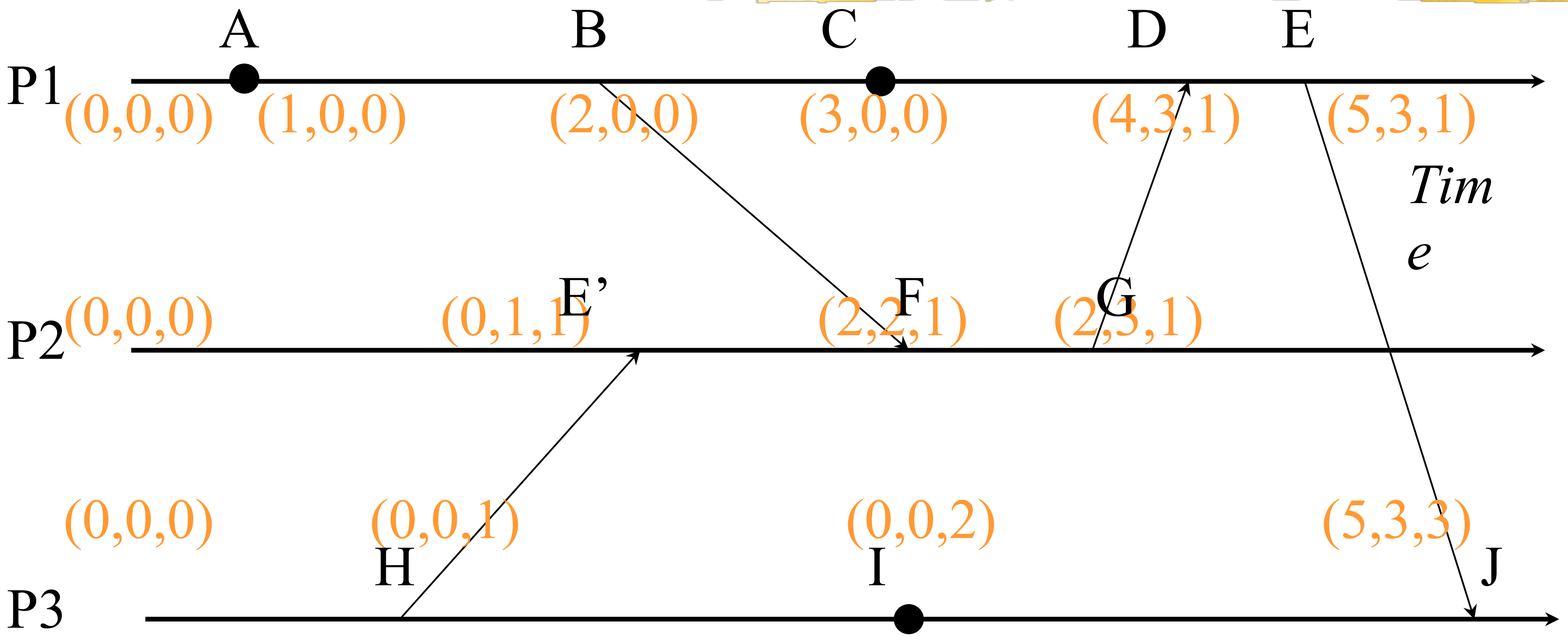
  We'll denote this as $VT_2 ||| VT_1$

Obeying Causality



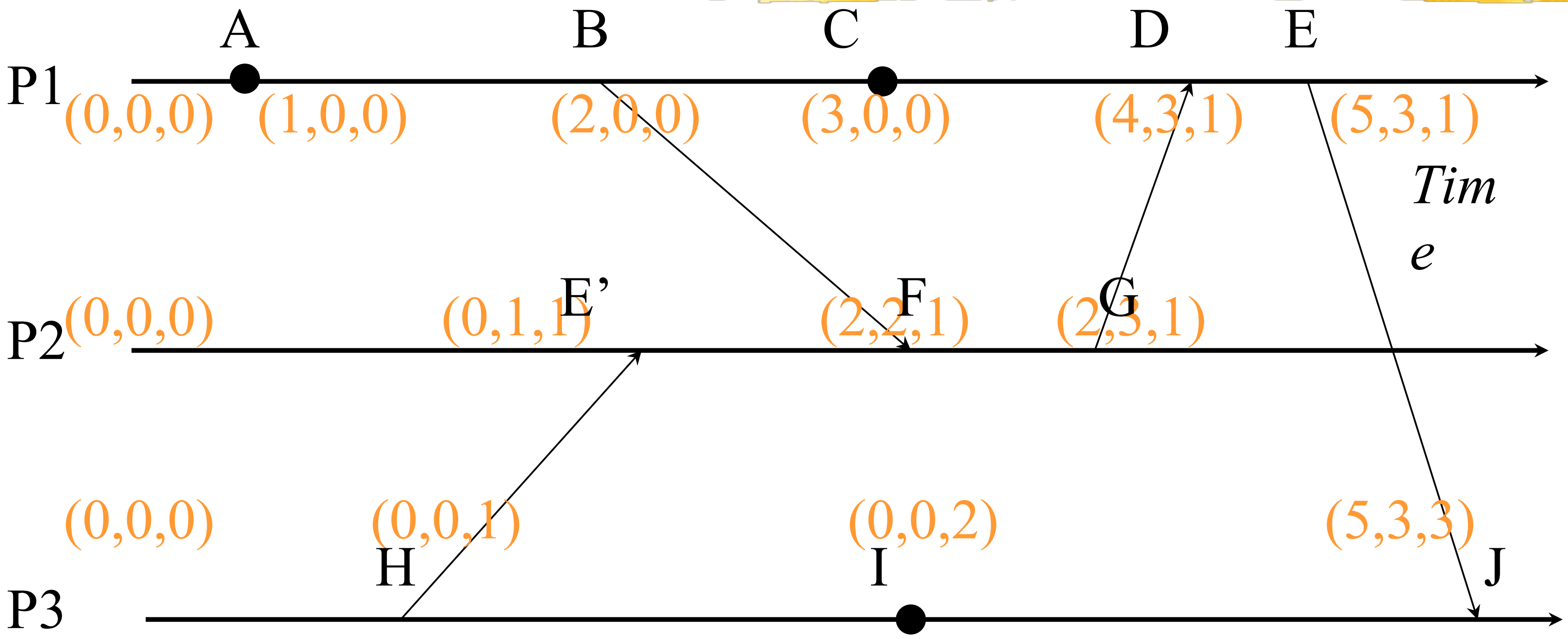
- A → B :: (1,0,0) < (2,0,0)
- B → F :: (2,0,0) < (2,2,1)
- A → F :: (1,0,0) < (2,2,1)

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

Vector Clocks example

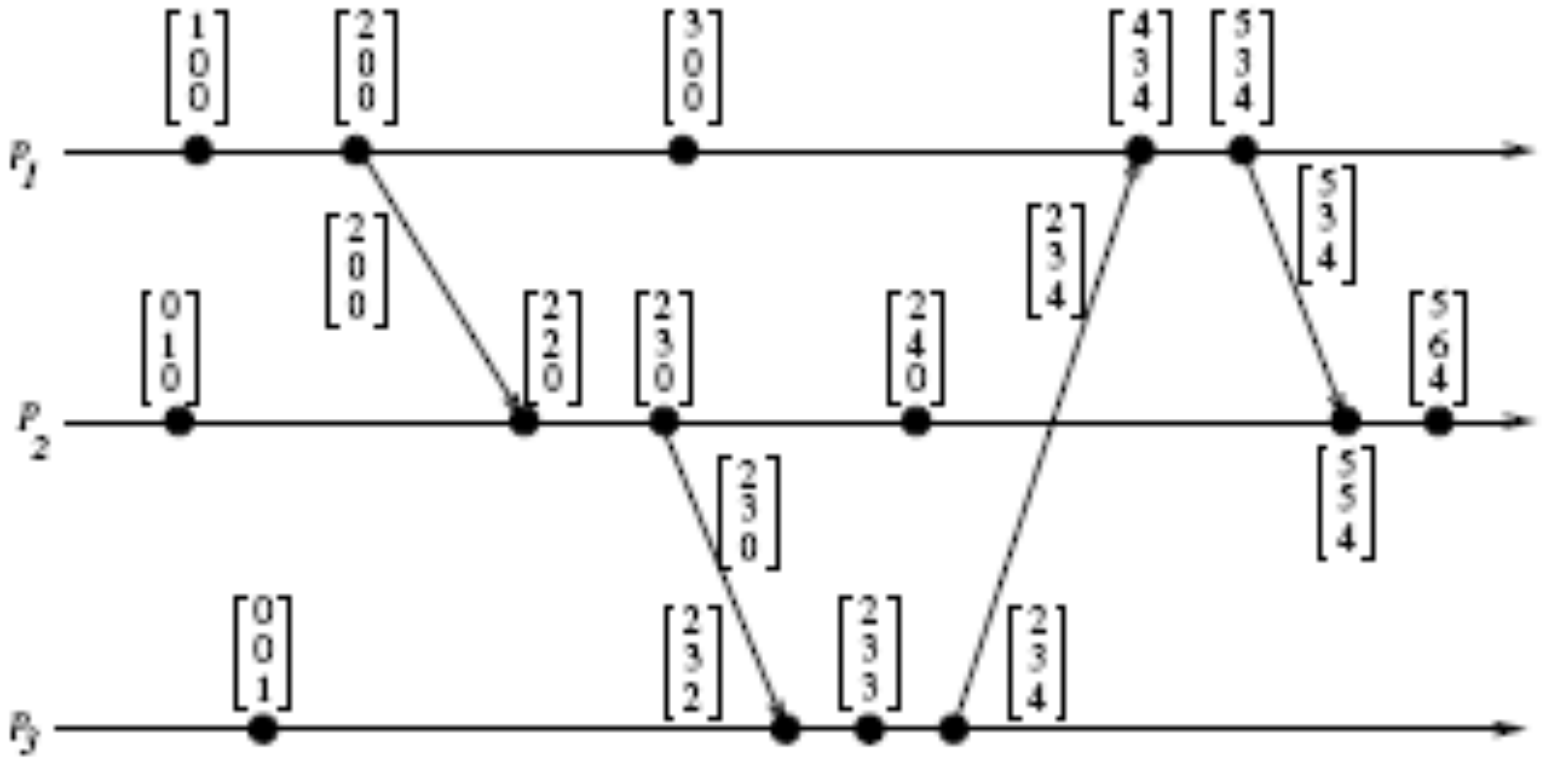


Figure 3.2: Evolution of vector time.

From A. Kshemkalyani and M. Singhal (Distributed Computing)

Vector Times (cont)

- Because of the transitive nature of the scheme, a process *may receive* time updates about clocks in non-neighboring process
- Since process P_i can advance the ith component of global time, it always has the most accurate knowledge of its local time
  - At any instant of real time $\forall i,j$: $C_i[i] \geq C_j[i]$

Structure of Vector Time

- For two time vectors u,v
  - $u \leq v$ iff $\forall i$: $u[i] \leq v[i]$
  - $u < v$ iff $u \leq v \wedge u \neq v$
  - $u || v$ iff $\sim(u<v) \wedge \sim(v<u)$ :: || is not transitive
- For an event set E,
  - $\forall e,e' \in E: e<e'$ iff $C(e)<C(e') \wedge e||e'$ iff iff $C(e)||C(e')$
- In order to determine if two events e,e' are causally related or not, just take their timestamps C(e) and C(e')
  - if $C(e)<C(e') \vee C(e')<C(e)$, then the events *are causally related*
  - Otherwise, they *are causally independent*

Matrix Time

- Vector time contains information about latest direct dependencies
  - What does Pi know about Pk
- Also contains info about latest direct dependencies of those dependencies
  - What does Pi know about what Pk knows about Pj
- Message and computation overheads are high
- Powerful and useful for applications like distributed garbage collection