# 416 Distributed Systems

Time in distributed systems

Oct 8, 2018

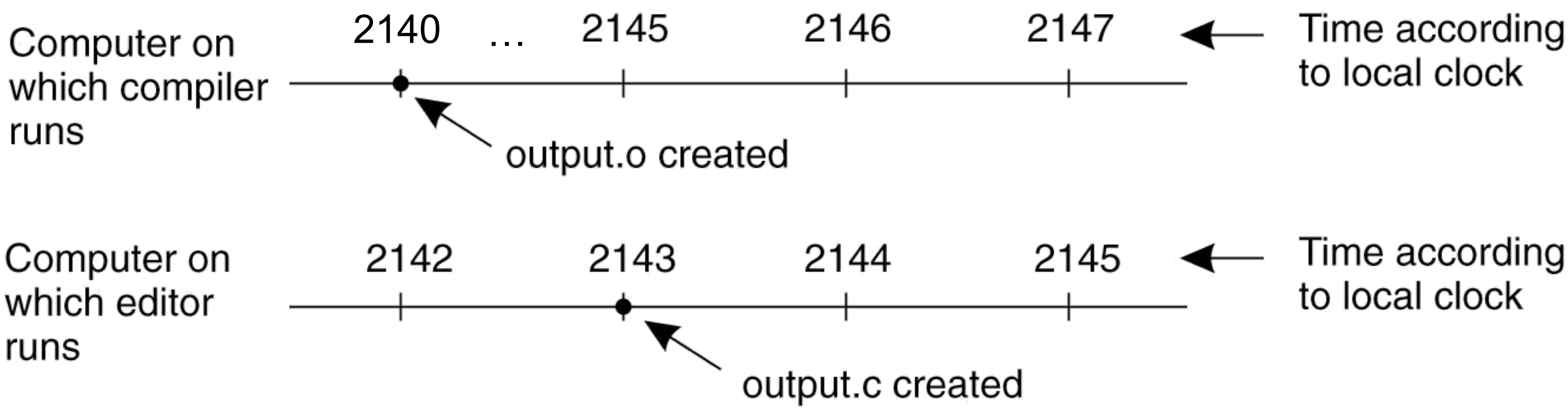
# Today's Lecture

- Need for time synchronization
- Time synchronization techniques
- Logical clocks
  - Lamport Clocks
  - Vector Clocks

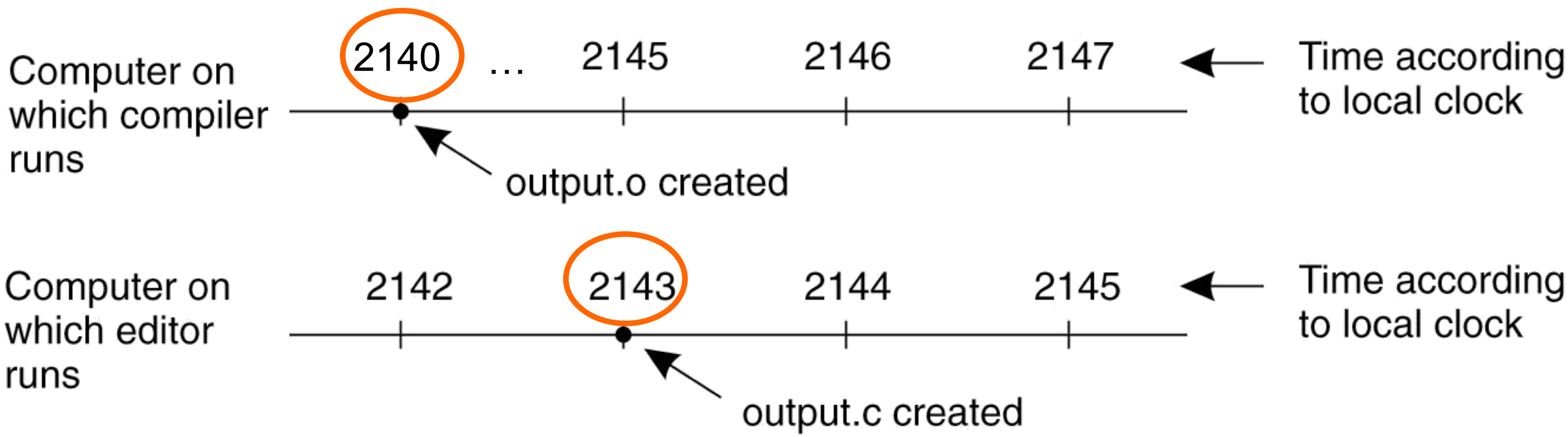
# Why Global Timing?

- Suppose there were a globally consistent time standard
- Would be handy
  - Who got last seat on airplane?
  - Who submitted final auction bid before deadline?
  - Did defense move before snap? (football reference)
  - RFS:
    - Did Append(rec0) happen before Append(rec1)?

# Impact of Clock Synchronization

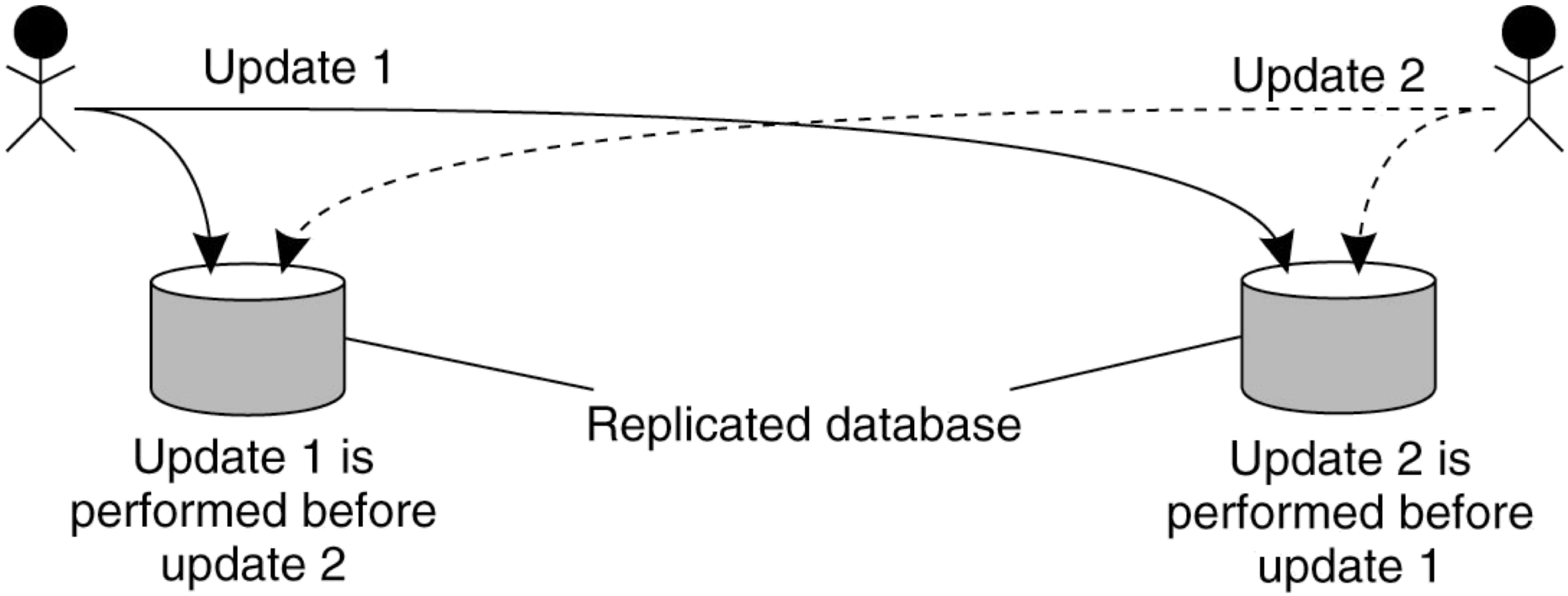


# Impact of Clock Synchronization




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

# Replicated Database Update



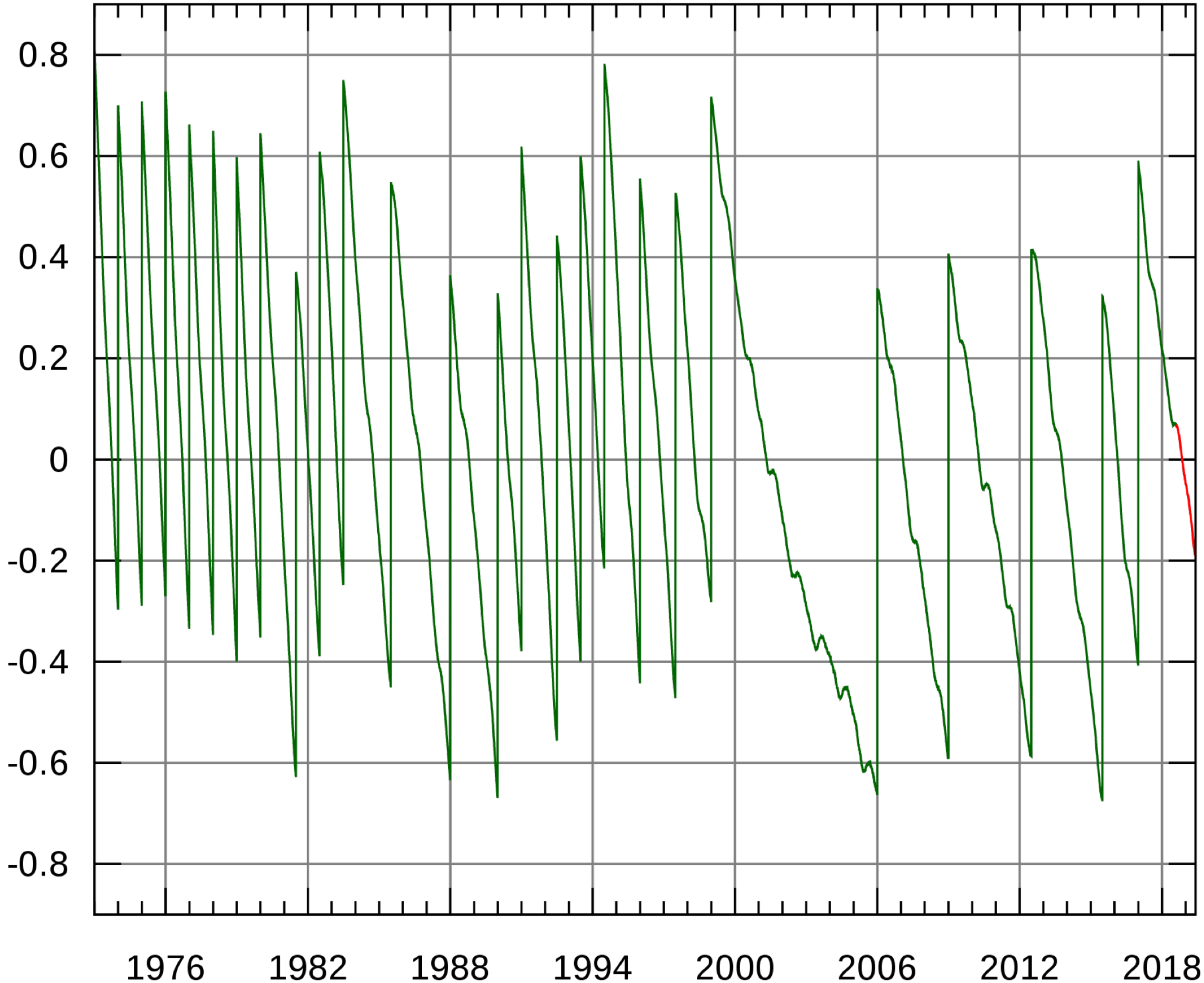

- Updating a replicated database and leaving it in an inconsistent state

# Time Standards

- UT1 (universal time)
  - Based on astronomical observations
  - ~ "Greenwich Mean Time" (GMT)
- TAI (international atomic time)
  - Started Jan 1, 1958
  - Each second is 9,192,631,770 cycles of radiation emitted by Cesium atom
  - Has diverged from UT1 due to slowing of earth's rotation
- UTC (coordinated universal time)
  - TAI + leap seconds to be within 0.9s of UT1
  - Currently ~37s

# Comparing Time Standards

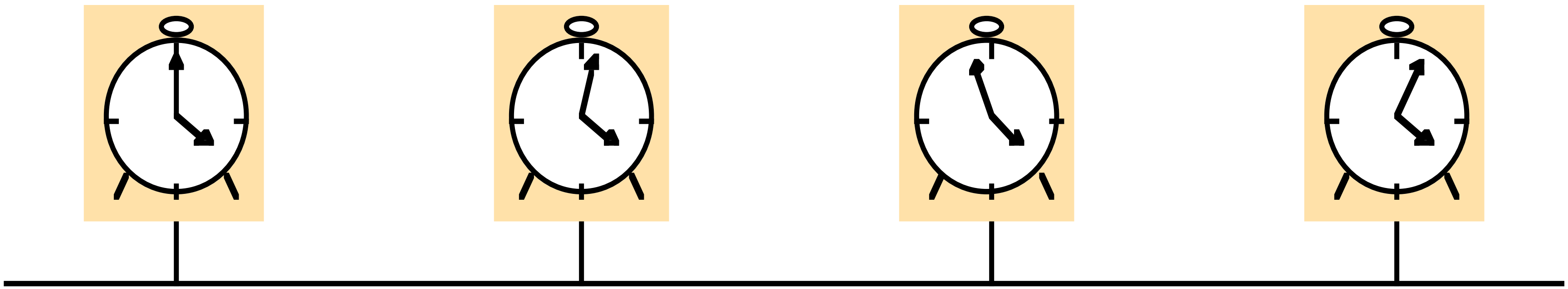
UT1 – UTC



# Coordinated Universal Time (UTC)

- Is broadcast from radio stations on land and satellite (e.g., GPS)
- Computers with receivers can synchronize their clocks with these timing signals
- Signals from land-based stations are accurate to about 0.1-10 millisecond
- Signals from GPS are accurate to about 1 microsecond
  - Why can't we use GPS receivers on all our computers?

# Clocks in a Distributed System

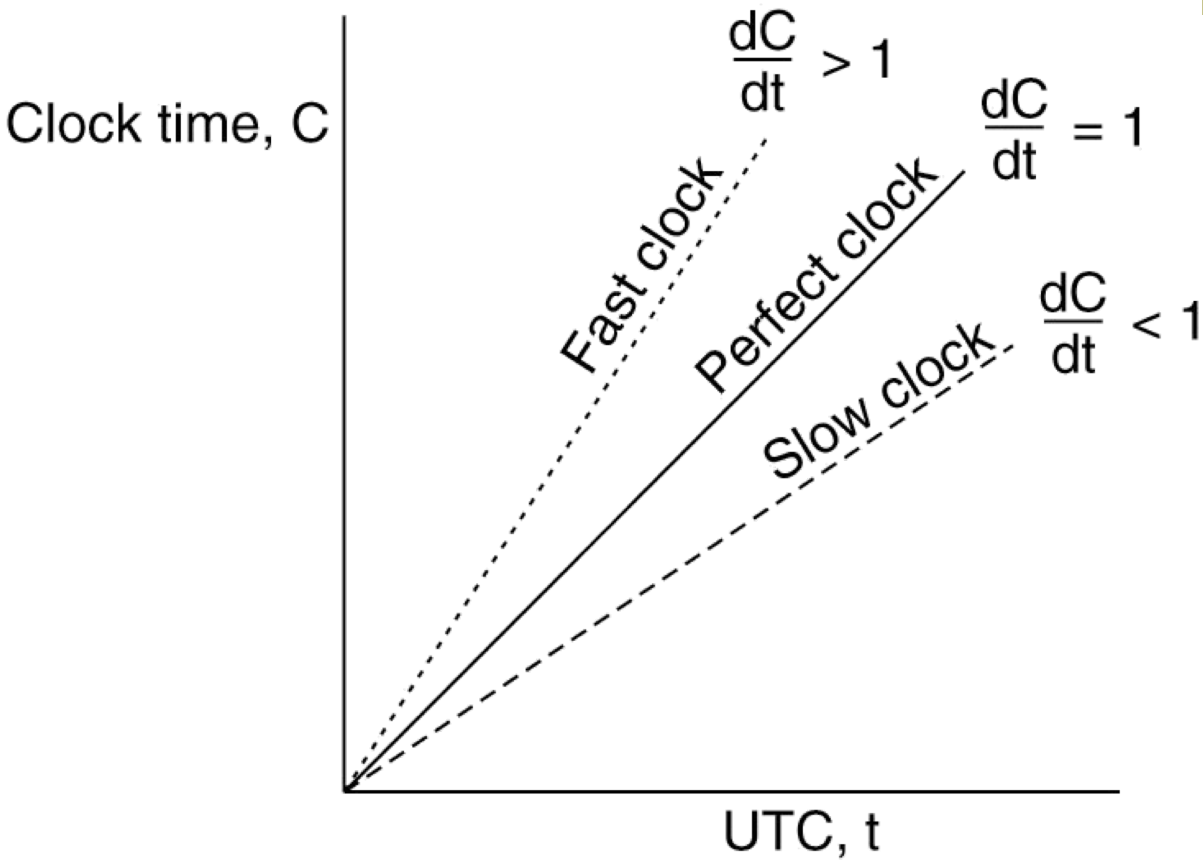


Network

- Computer clocks are not generally in perfect agreement
  - **Skew**: the difference between the times on two clocks (at any instant)

- Computer clocks are subject to clock drift (they count time at different rates; consider batteries)
  - **Clock drift rate**: the difference per unit of time from some ideal reference clock
  - Ordinary quartz clocks drift by about 1 sec in 11-12 days. ($10^{-6}$ secs/sec).
  - High precision quartz clocks drift rate is about $10^{-7}$ or $10^{-8}$ secs/sec

# Clock Synchronization Algorithms



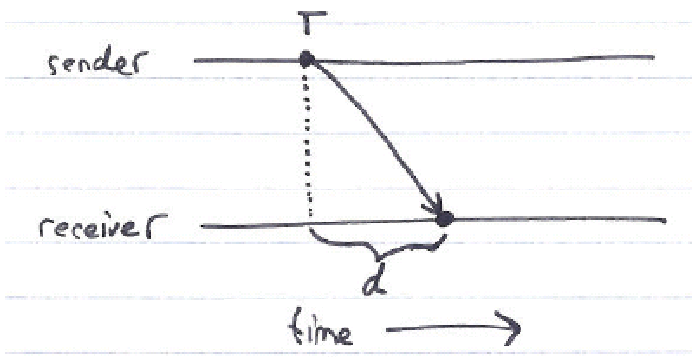

- The relation between clock time and UTC when clocks tick at different rates.

# Today's Lecture

- Need for time synchronization
- Time synchronization techniques
- Lamport Clocks
- Vector Clocks

# Perfect networks

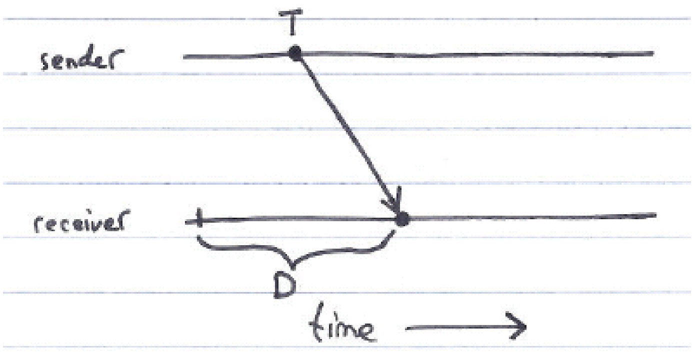
- Messages always arrive, with propagation delay exactly *d*




- Sender sends time *T* in a message
- Receiver sets clock to *T+d*
  - Synchronization is exact

# Synchronous networks

- Messages always arrive, with propagation delay *at most D*



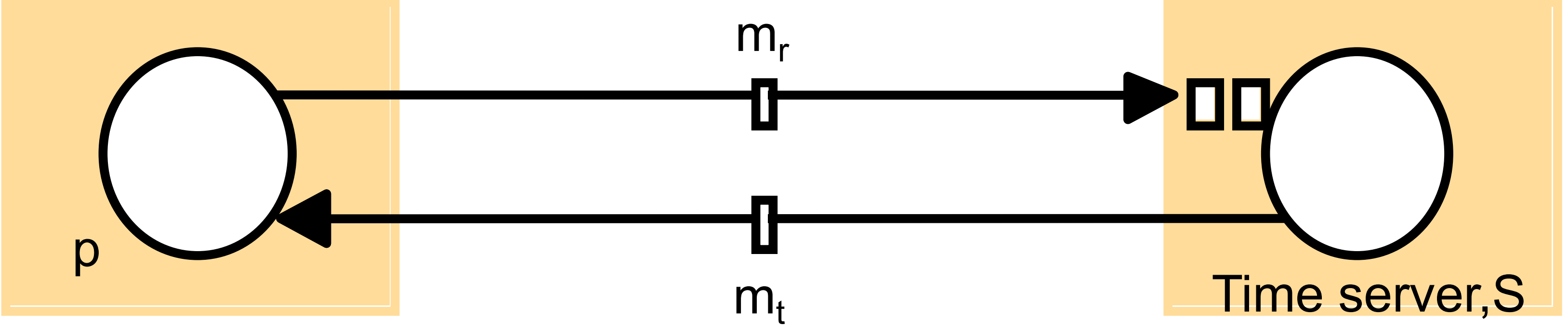

- Sender sends time *T* in a message
- Receiver sets clock to *T + D/2*
  - Synchronization error is at most *D/2*

# Synchronization in the real world

- Real networks are asynchronous
  - Message delays are arbitrary
- Real networks are unreliable
  - Messages don't always arrive

# Cristian's Time Sync ('89)

- A time server $S$ receives signals from a UTC source
  - Process $p$ requests time in $m_r$ and receives $t$ in $m_t$ from $S$
  - $p$ sets its clock to $t + T_{\text{round-trip}}/2$
  - Accuracy $\pm$ ($T_{\text{round-trip}}/2$ - *min*) :
    - Where *min* is minimum one-way transmission delay
    - because the earliest time $S$ puts $t$ in message $m_t$ is *min* after $p$ sent $m_r$.
    - the latest time was *min* before $m_t$ arrived at $p$
    - the time by $S$'s clock when $m_t$ arrives is in the range [$t$+*min*, $t + T_{\text{round-trip}}$ - *min*]



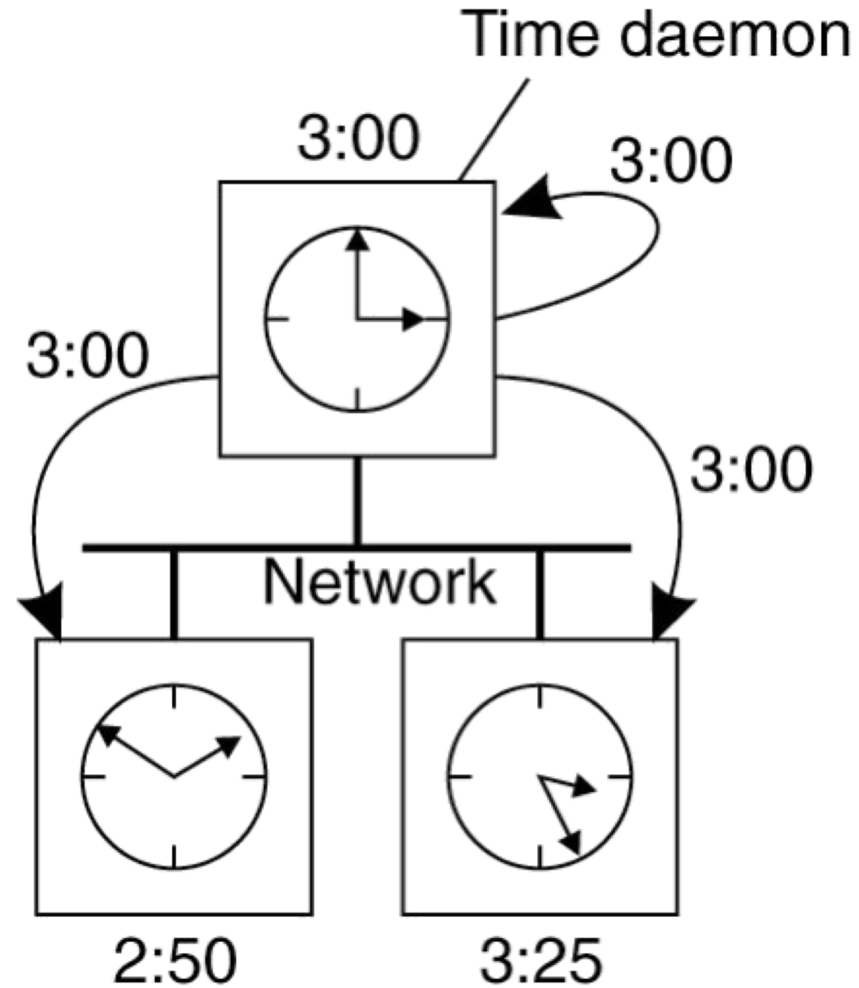

$T_{round}$ is the round trip time recorded by $p$
*min* is an estimated minimum one way delay

# Berkeley algorithm

- Cristian's algorithm -
  - a single time server might fail, so they suggest the use of a group of synchronized servers
  - it does not deal with faulty servers
- Berkeley algorithm (also 1989)
  - An algorithm for *internal* synchronization of a group of computers
  - A *master* polls to collect clock values from the others (*slaves*)
  - The master uses round trip times to estimate the slaves' clock values (only master computes RTT)
  - It takes an average (eliminating any above average round trip time or with faulty clocks)
  - It sends the required adjustment to the slaves (better than sending the time which depends on the round trip time)
  - Failures
    - If master fails, can elect a new master to take over (not in bounded time)

# The Berkeley Algorithm (1)

- The time daemon asks all the other machines for their clock values.

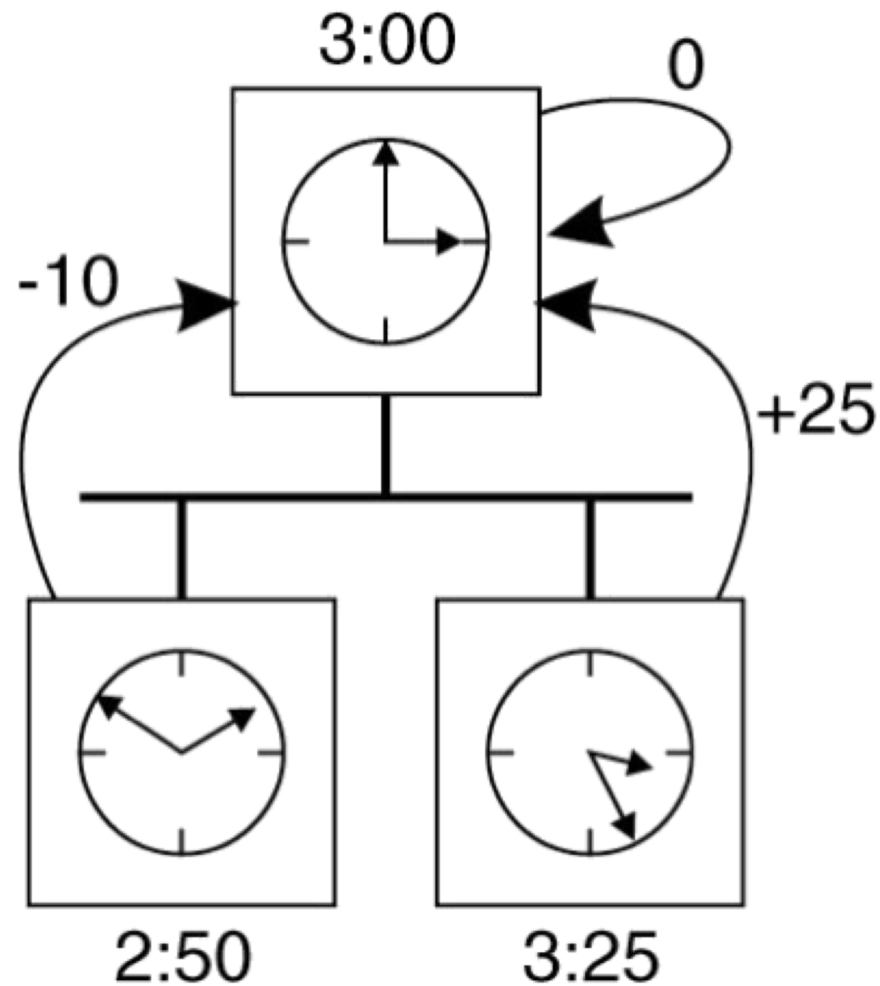


# The Berkeley Algorithm (2)

- The machines answer.

Compute avg:
+15 / 3 = +5

Adjustment:
0 → +5 = +5
-10 → +5 = +15
+25 → +5 = -20

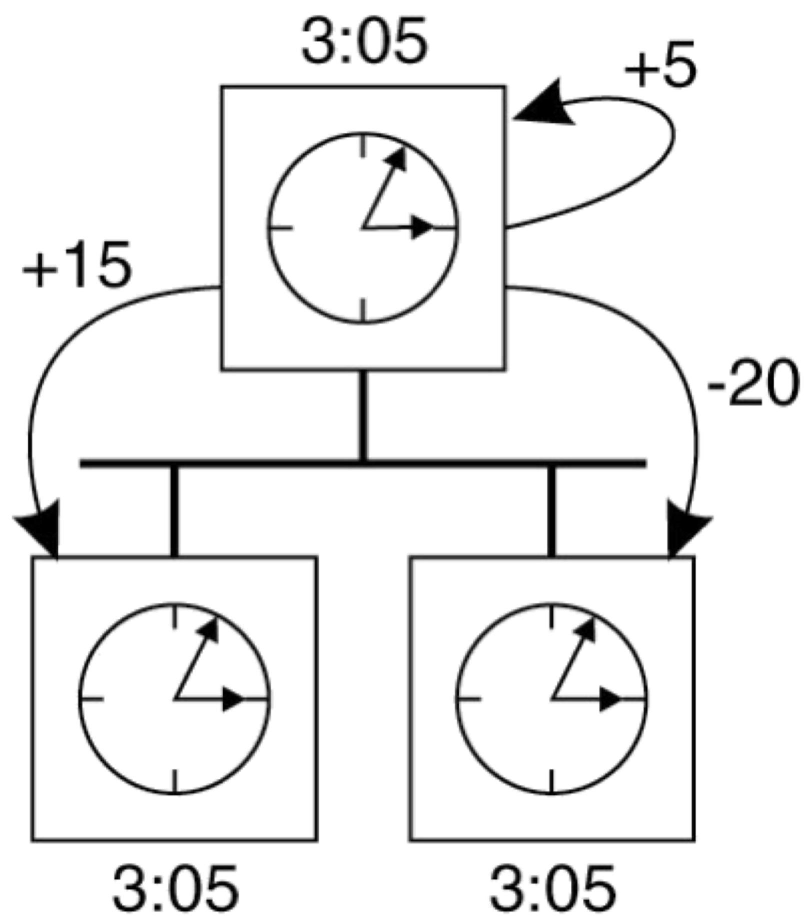


# The Berkeley Algorithm (3)

- The time daemon tells everyone how to adjust their clock.

Compute avg:
+15 / 3 = +5

Adjustment:
0 → +5 = +5
-10 → +5 = +15
+25 → +5 = -20



# Network Time Protocol (NTP)

(invented by David Mills, 1981)

- A time service for the Internet - synchronizes clients to

Reli… …enticates
time sou…

Primary servers are connected to UTC

Secondary servers are synchronized to primary servers

Synchronization subnet - lowest level servers in users’ computers

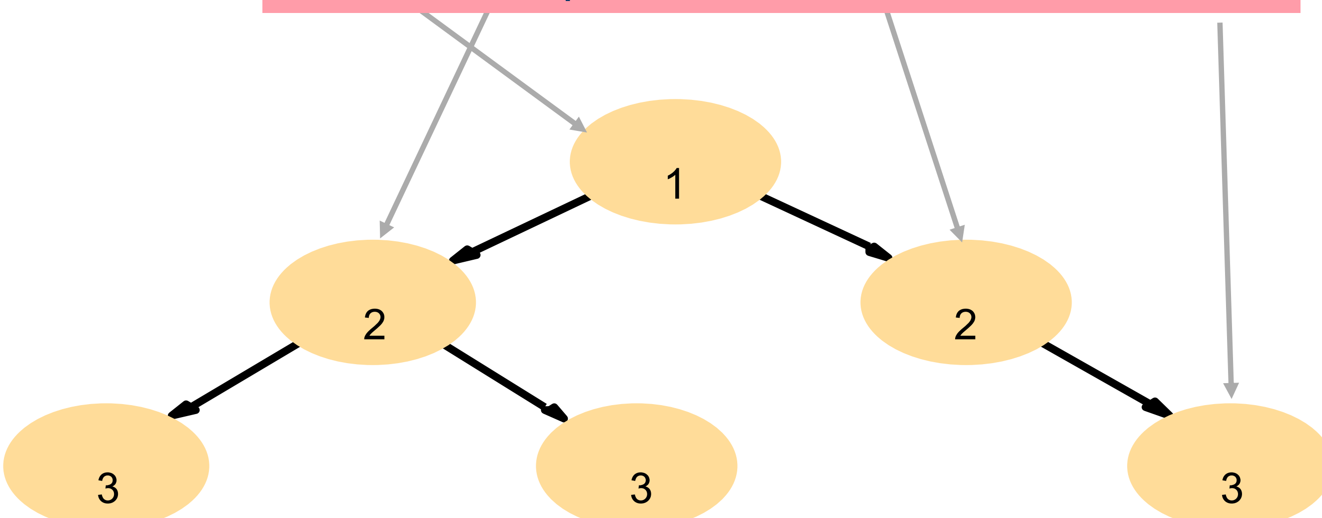



Figure 10.3

# The Network Time Protocol (NTP)

- Uses UDP (minimal overhead/OS stack latency)
- Uses a hierarchy of time servers
  - Class 1 servers have highly-accurate clocks
    - connected directly to atomic clocks, etc.
  - Class 2 servers get time from only Class 1 and Class 2 servers
  - Class 3 servers get time from any server (usually 3)
- Synchronization similar to Cristian's alg.
  - Modified to use multiple one-way messages instead of immediate round-trip
- Accuracy: Local ~1ms, Global ~10ms

# How To Change Time

- Can’t just change time
  - Why not?

# How To Change Time

- Can’t just change time
  - Why not?
- Change the update rate for the clock
  - Changes time in a more gradual fashion
  - Prevents inconsistent local timestamps

# Important Lessons

- Clocks on different systems will always behave differently
  - Skew and drift between clocks

- Time disagreement between machines can result in undesirable behavior

- Clock synchronization
  - Rely on a time-stamped network messages
  - Estimate delay for message transmission
  - Can synchronize to UTC or to local source
  - Clocks never exactly synchronized

- Often inadequate for distributed systems
  - might need totally-ordered events
  - might need millionth-of-a-second precision

# Today's Lecture

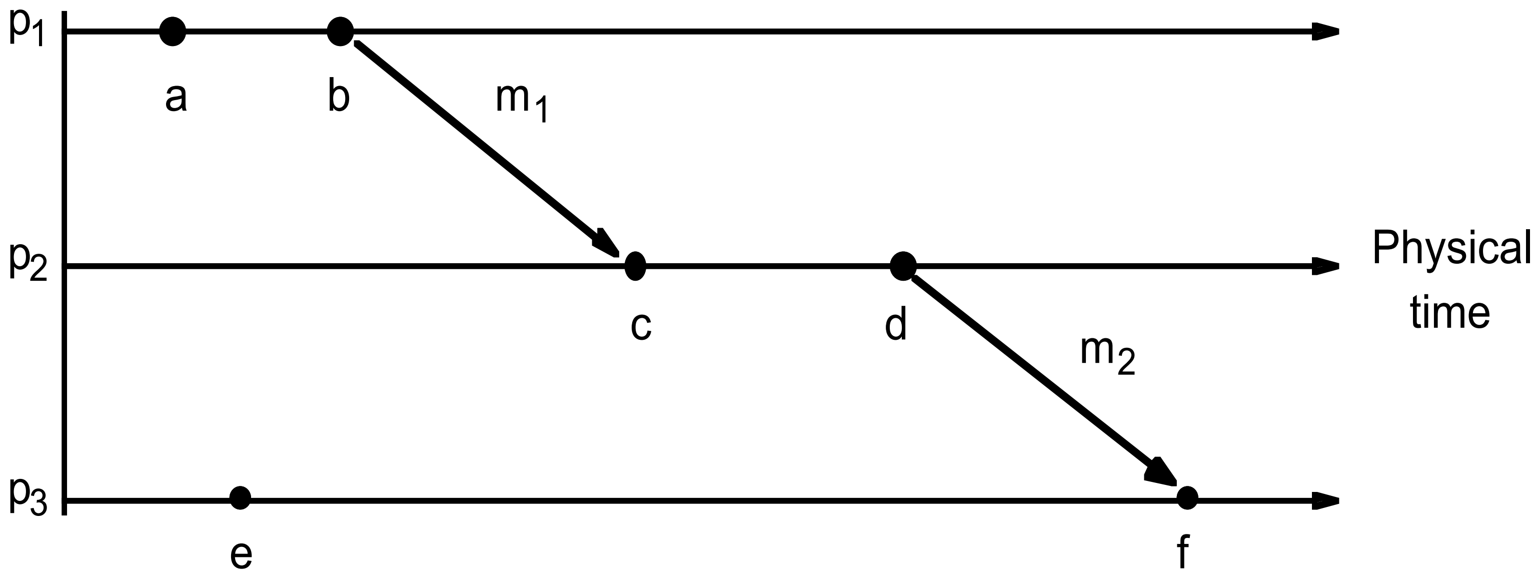
- Need for time synchronization
- Time synchronization techniques
- Lamport Clocks
- Vector Clocks

# Logical time

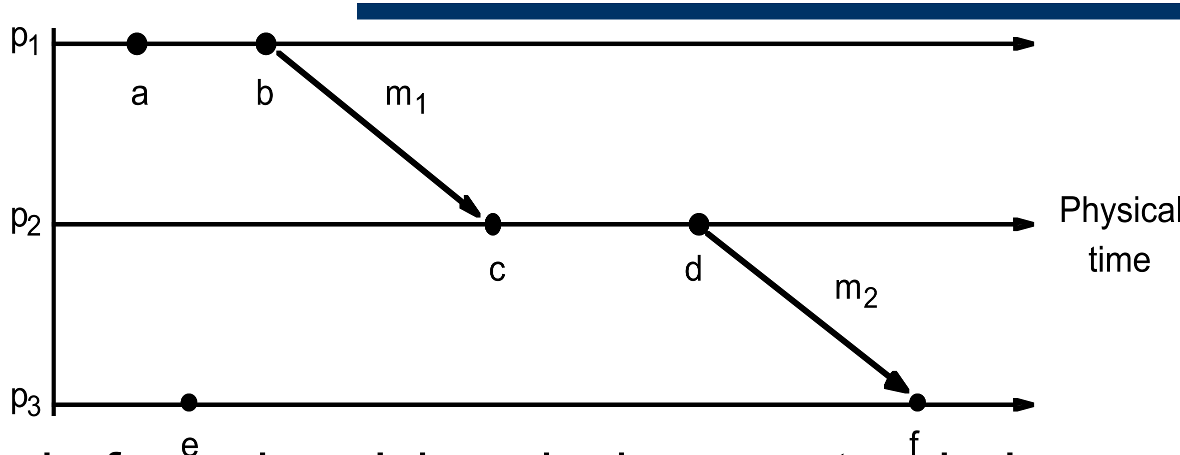
- Capture just the “happens before” relationship between events
  - Discard the infinitesimal granularity of time
  - Corresponds roughly to causality

# Logical time and logical clocks (Lamport 1978)

- Events at three processes



# Logical time and logical clocks (Lamport 1978)

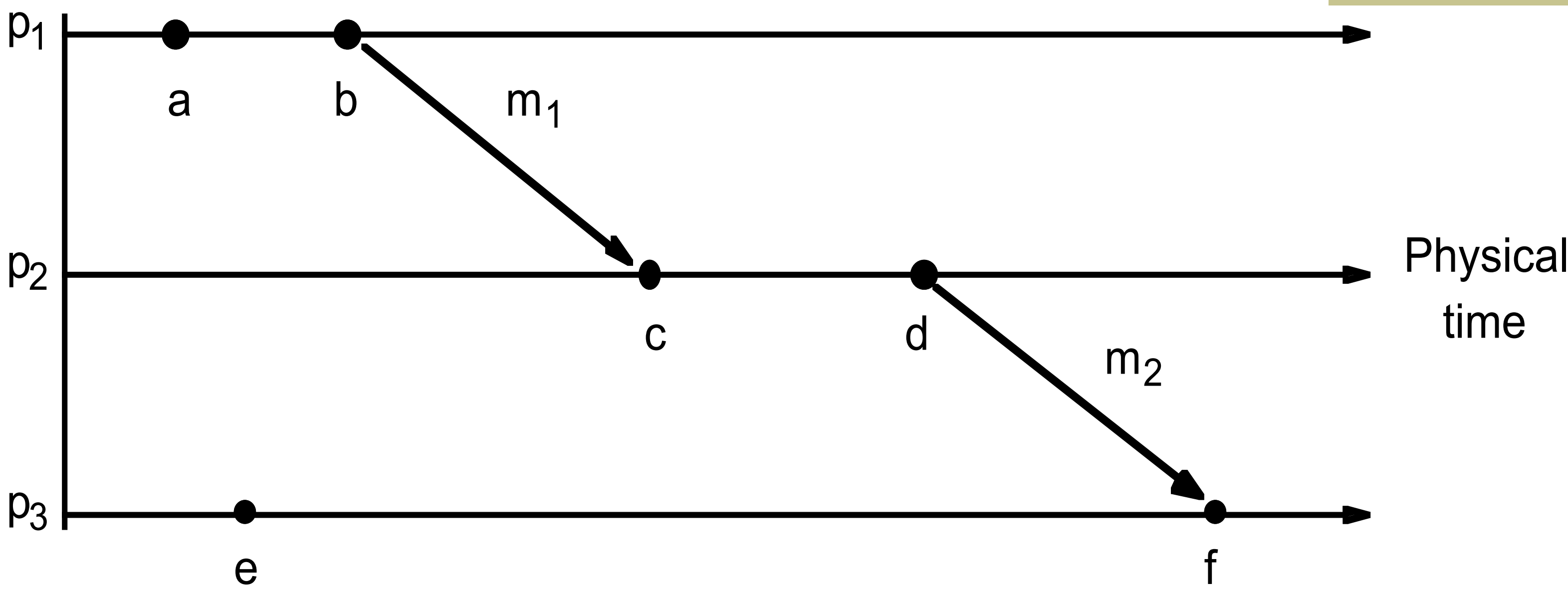



- Instead of synchronizing clocks, event ordering can be used

1. If two events occurred at the same process $p_i$ ($i = 1, 2, \ldots N$) then they occurred in the order observed by $p_i$, that is the definition of: $\rightarrow_i$
2. When a message, m is sent between two processes, send(m) 'happens before' receive(m)
3. The 'happened before' relation is transitive

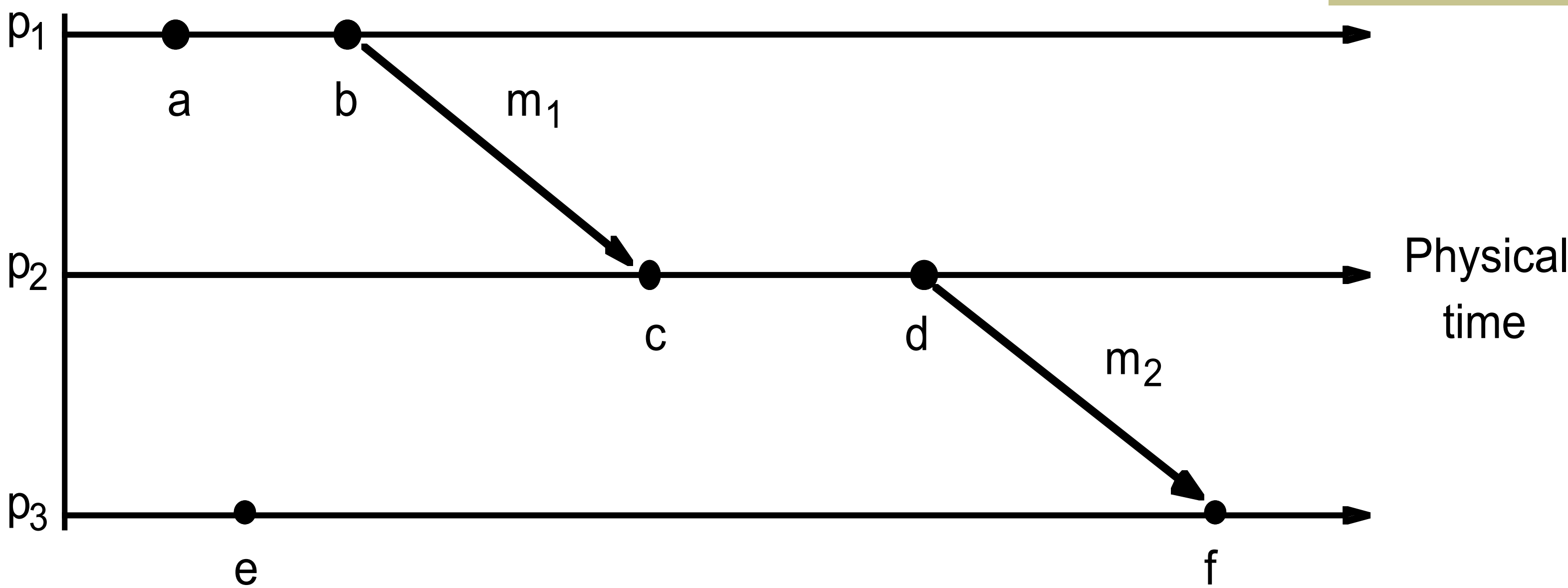
- The happened before relation ($\rightarrow$) is necessary for causal ordering

# Logical time and logical clocks (Lamport 1978)



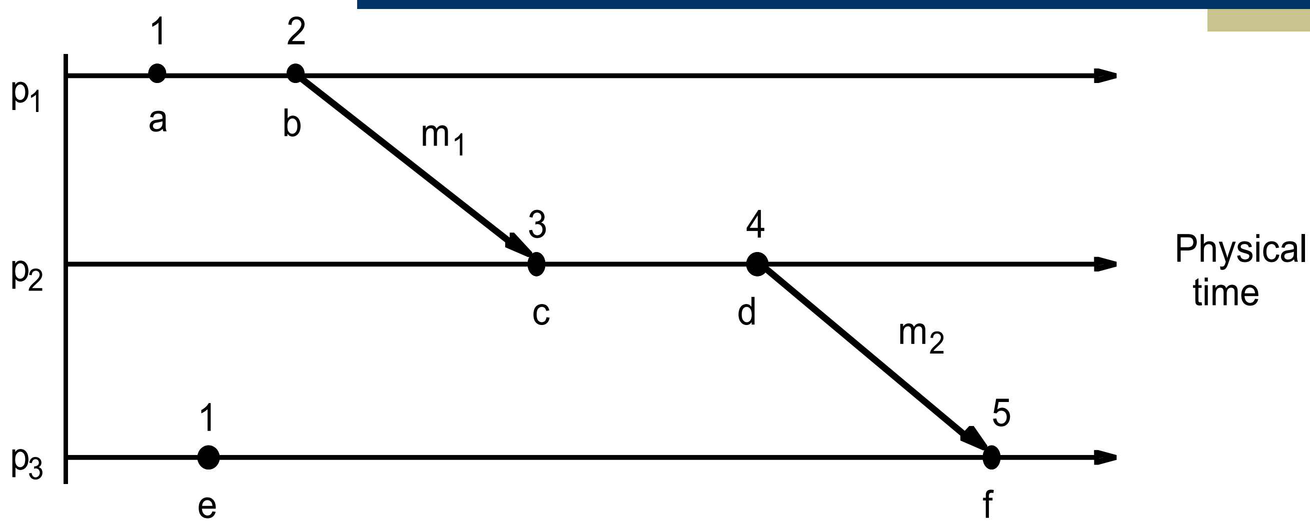

- $a \rightarrow b$ (at $p_1$) $c \rightarrow d$ (at $p_2$)
- $b \rightarrow c$ because of $m_1$
- also $d \rightarrow f$ because of $m_2$

# Logical time and logical clocks (Lamport 1978)



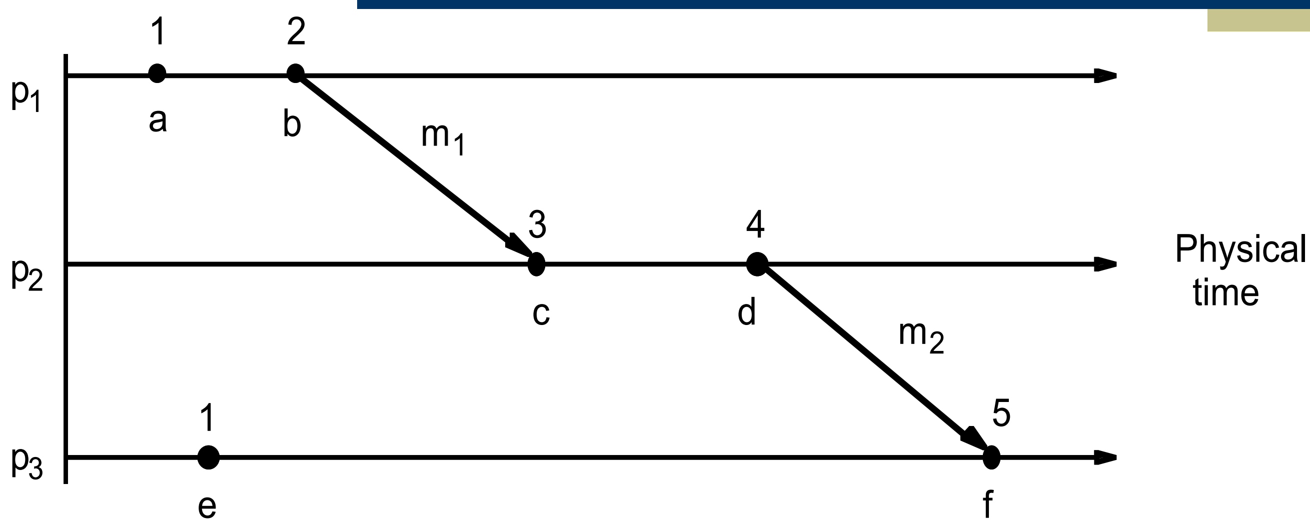

- Not all events are related by $\rightarrow$
- Consider *a* and *e* (different processes and no chain of messages to relate them)
  - they are not related by $\rightarrow$ ; they are said to be concurrent
  - written as *a* || *e*

# Lamport Clock (1)




- A logical clock is a monotonically increasing software counter
  - It need *not* relate to a physical clock.
- Each process $p_i$ has a logical clock, $L_i$ which can be used to apply logical timestamps to events
  - Rule 0: initially all clocks are set to 0
  - Rule 1: $L_i$ is incremented by 1 before each event at process $p_i$
  - Rule 2:
    - (a) when process $p_i$ sends message $m$, it piggybacks $t = L_i$
    - (b) when $p_j$ receives $(m,t)$ it sets $L_j := max(L_j, t)$ and applies rule 1 before timestamping the event *receive* $(m)$

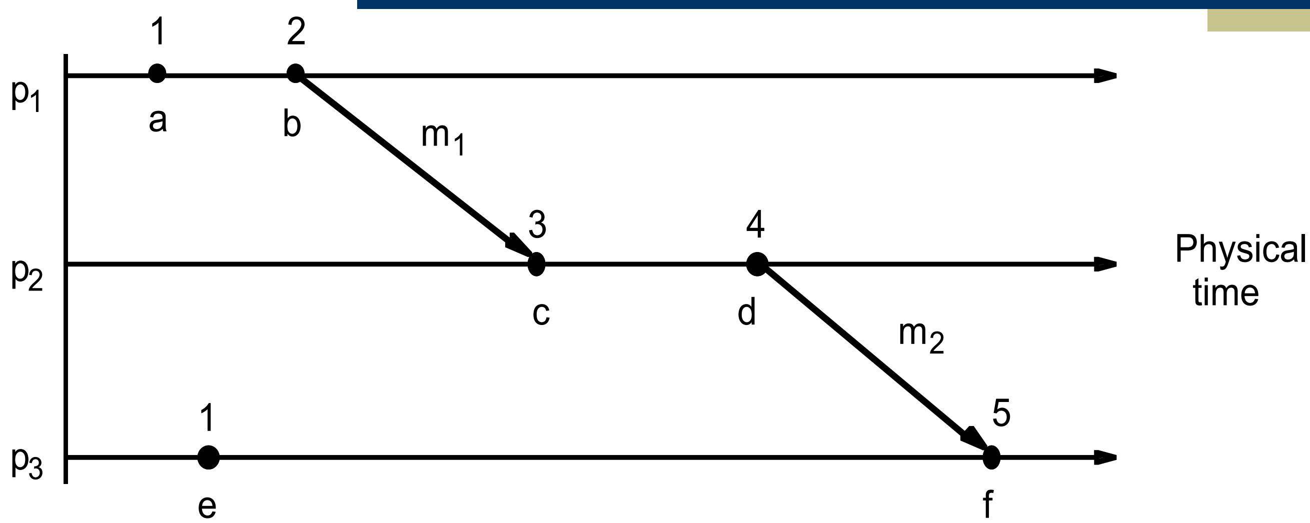
# Lamport Clock (1)




- each of $p_1$, $p_2$, $p_3$ has its logical clock initialised to zero,
- the clock values are those immediately after the event.
- e.g. 1 for a, 2 for b.

- for $m_1$, 2 is piggybacked and c gets max(0,2)+1 = 3

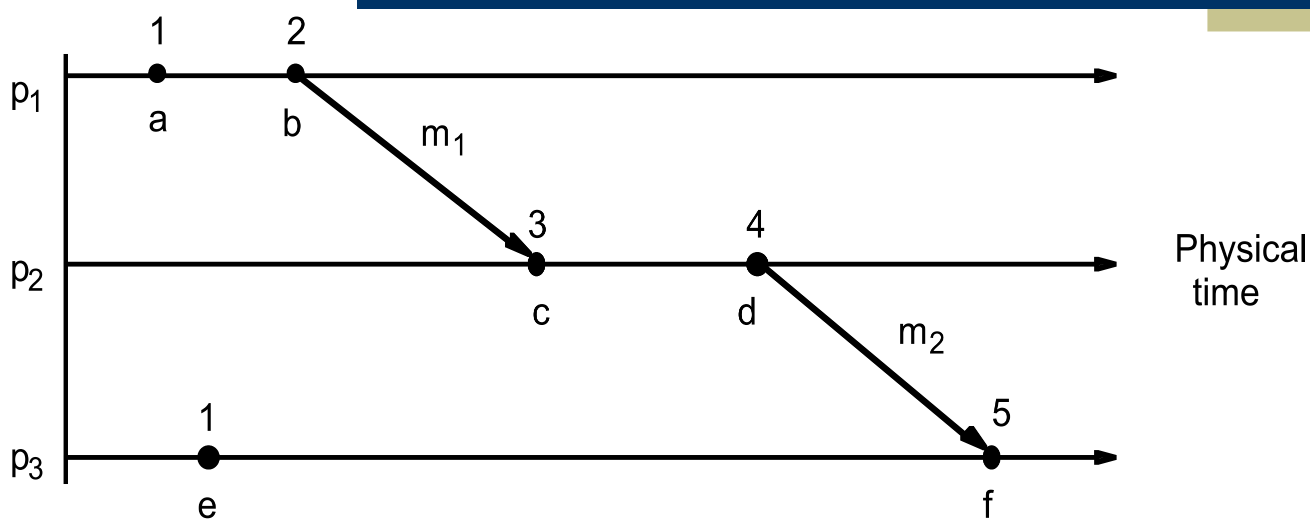
# Lamport Clock (1)




- $e \rightarrow e'$ (e happened before e') implies $L(e)<L(e')$
  (where L(e) is Lamport clock value of event e)

- **The converse is not true**, that is $L(e)<L(e')$ does not imply $e \rightarrow e'$. What's an example of this above?

# Lamport Clock (1)




- $e \rightarrow e'$ (e happened before e') implies $L(e)<L(e')$
- The converse is not true, that is $L(e)<L(e')$ does not imply $e \rightarrow e'$
  - e.g. $L(b) > L(e)$ but $b \parallel e$

# Lamport logical clocks

- Lamport clock *L* orders events consistent with logical "happens before" ordering
  - If e → e', then *L(e)* < *L(e')*
- But not the converse
  - *L(e)* < *L(e')* does not imply *e* → *e'*

- Similar rules for concurrency
  - *L(e)* = *L(e')* implies *e* ‖ *e'* (for distinct *e*,*e'*)
  - *e* ‖ *e'* does not imply *L(e)* = *L(e')*
  - i.e., Lamport clocks arbitrarily order some concurrent events

# Total-order Lamport clocks

- Many systems require a total-ordering of events, not a partial-ordering
- Use Lamport's algorithm, but break ties using the process ID; one example scheme:
  - $L(e) = M * L_i(e) + i$
    - $M$ = maximum number of processes
    - i = process ID

# Today's Lecture

- Need for time synchronization
- Time synchronization techniques
- Lamport Clocks
- Vector Clocks

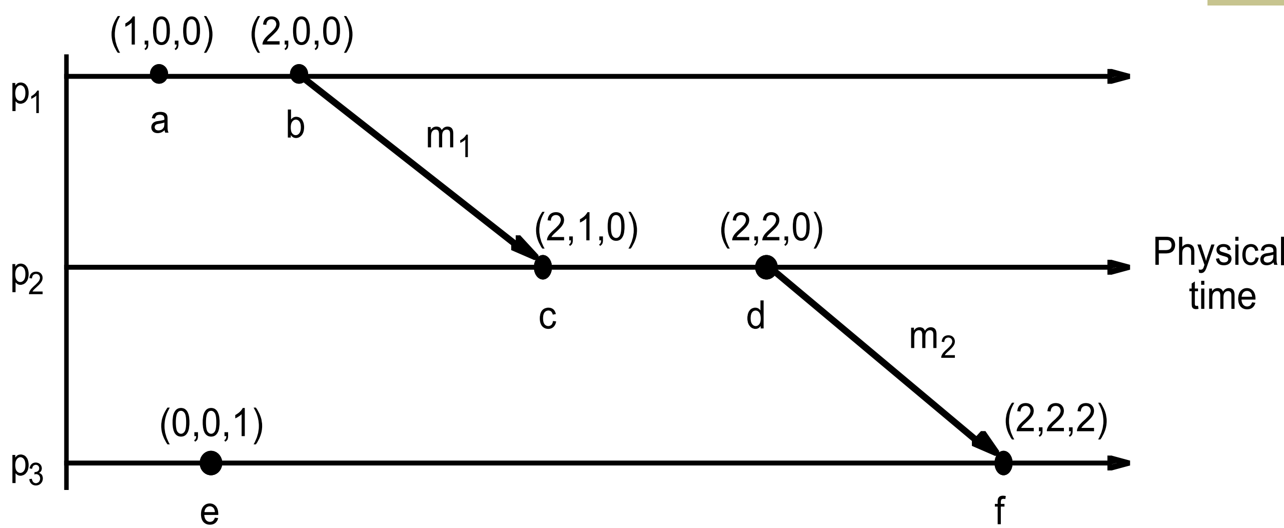
# Vector Clocks

- Vector clocks overcome the shortcoming of Lamport logical clocks
  - *L*(e) < *L*(*e*') does not imply *e* happened before *e*'
- Goal
  - Want ordering that matches happened before
  - V(e) < V(e' ) if and only if e → e'
- Method
  - Label each event by vector V(e) [$c_1$, $c_2$ …, $c_n$]
    - $c_i$ = # events in process i that precede e

# Vector Clock Algorithm

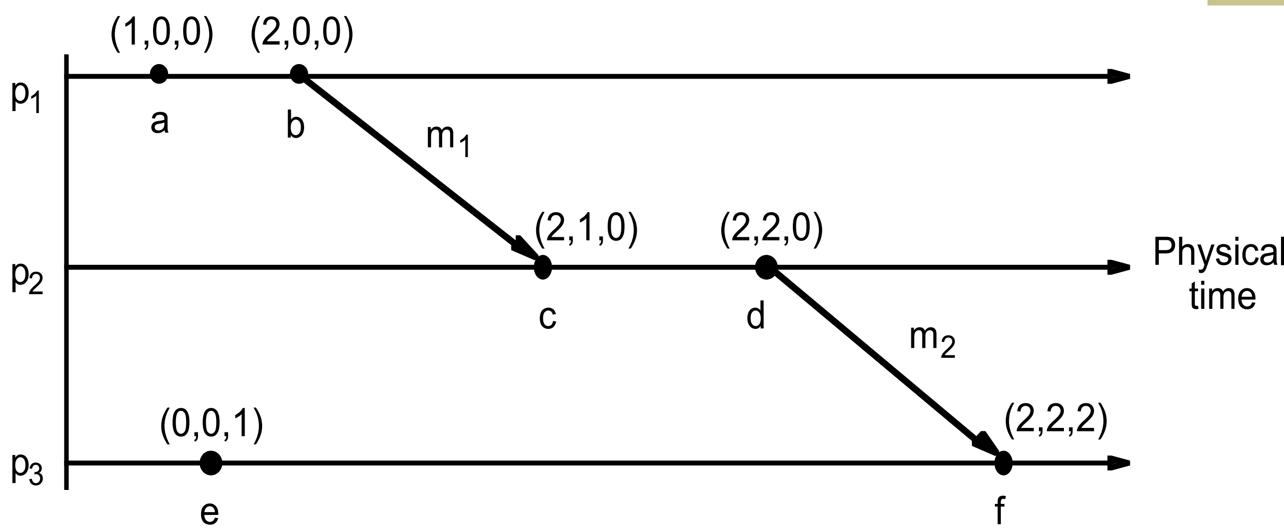
- Initially, all vectors [0,0,…,0]
- For event on process i, increment own $c_i$
- Label message sent with local vector
- When process j receives message with vector $[d_1, d_2, \ldots, d_n]$:
  - Set each local vector entry k to $\max(c_k, d_k)$
  - Increment value of $c_j$

# Vector Clocks



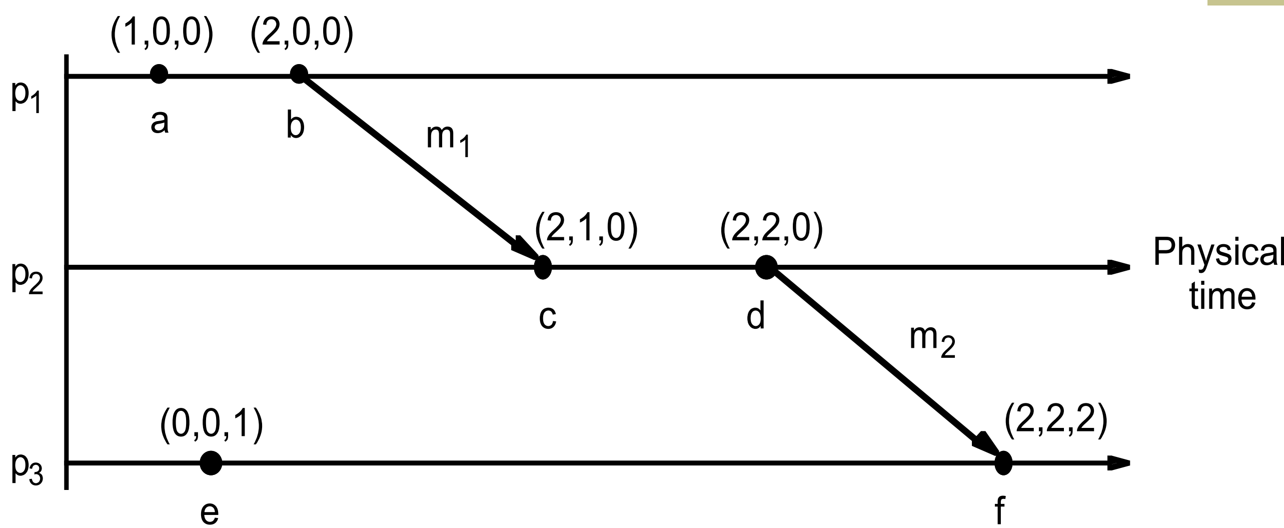

- At $p_1$
  - *a occurs at* (1,0,0); *b* occurs at (2,0,0)
  - piggyback (2,0,0) on $m_1$
- At $p_2$ on receipt of $m_1$ use *max* ((0,0,0), (2,0,0)) = (2, 0, 0) and add 1 to own element = (2,1,0)
- Meaning of =, <=, max etc for vector timestamps
  - compare elements pairwise

# Vector Clocks




- Note that e → e' implies V(e)<V(e'). The converse is also true
- Can you see a pair of concurrent events; Can you infer they are concurrent from their vectors clocks?

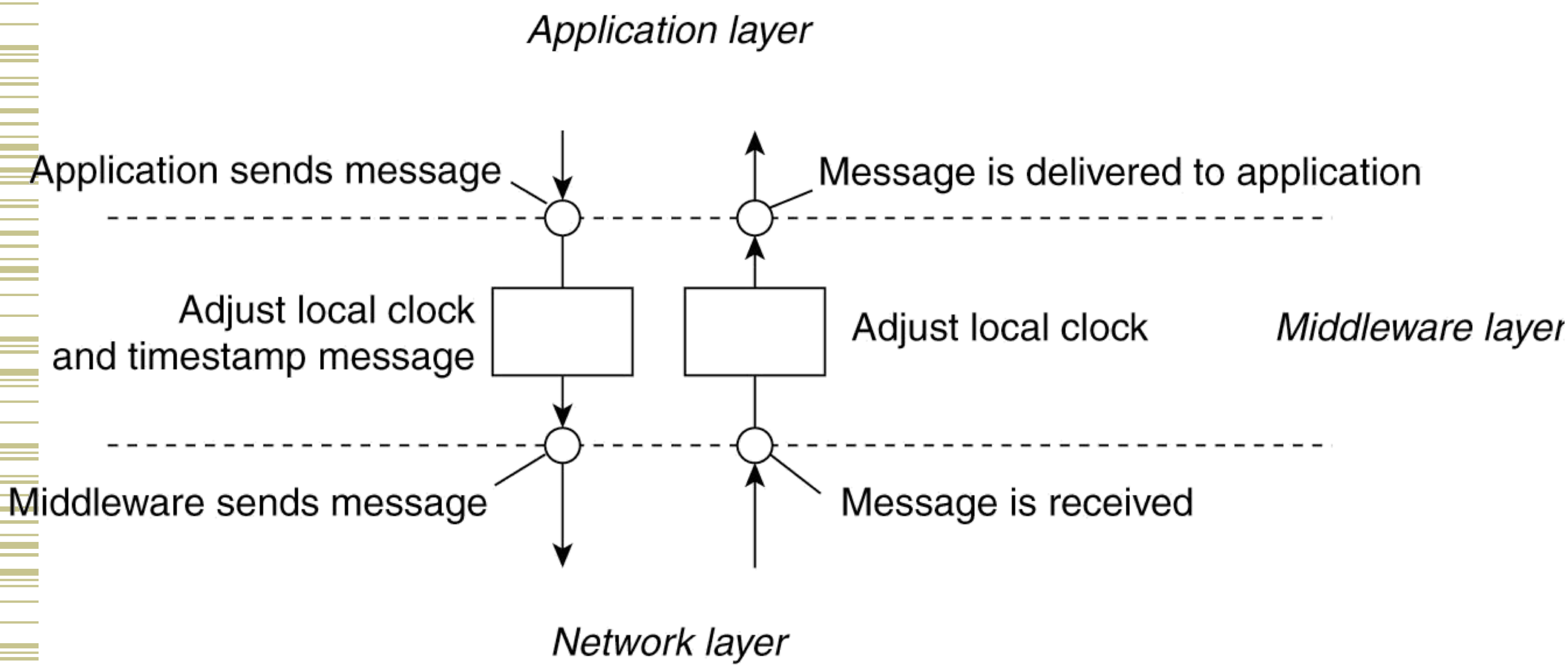
# Vector Clocks




- Note that e $\rightarrow$ e' implies V(e)<V(e'). The converse is also true
- Can you see a pair of concurrent events?
  - *c* || *e* (concurrent) because neither *V*(c) <= *V*(*e*) nor *V*(*e*) <= *V*(*c*)

# Implementing logical clocks

- Positioning of logical timestamping in distributed systems.



# Distributed time

- Premise
  - The notion of time is well-defined (and measurable) at each single location
  - But the relationship between time at different locations is unclear
    - Can minimize discrepancies, but never eliminate them
- Reality
  - Stationary GPS receivers can get global time with < 1μs error
  - Few systems designed to use this; logical clocks key mechanism for ordering
    - Recent exception: (Spanner system from Google)

# Important Points

- Physical Clocks
  - Can keep closely synchronized, but never perfect
- Logical Clocks
  - Encode happens before relationship (necessary for causality)
  - Lamport clocks provide only one-way encoding
  - Vector clocks precedence necessary for causality (but *not sufficient*: could have been caused by some event along the path, not all events)