Distributed Systems

15-440/640

Fall 2018

5 – Time Synchronization

Readings: Tanenbaum Book, Chapters 6.1 and 6.2.

Announcements

Who is taking legible notes and is willing to share them?

There is a student in our class who needs a copy of the class notes. We are looking for a volunteer peer notetaker. If you are interested in volunteering, please see me after class.

P0 due on Thursday.

HW1 will be released tomorrow (Wednesday).

Daniel's OH today will start at 1pm (instead of 12.30).

Today's Lecture

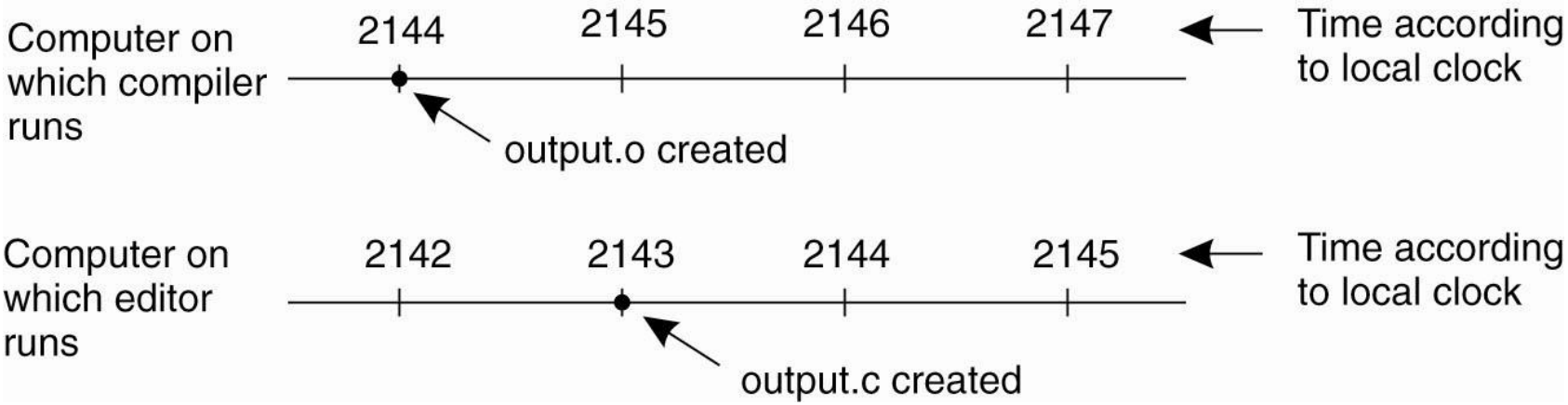
- Need for time Synchronization
- Basic Time Synchronization Techniques
- Lamport Clocks
- Vector Clocks
- Time Synchronization in 2018

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?
  - When exactly was a message sent (indoor positioning)?
  - Did defense move before snap?

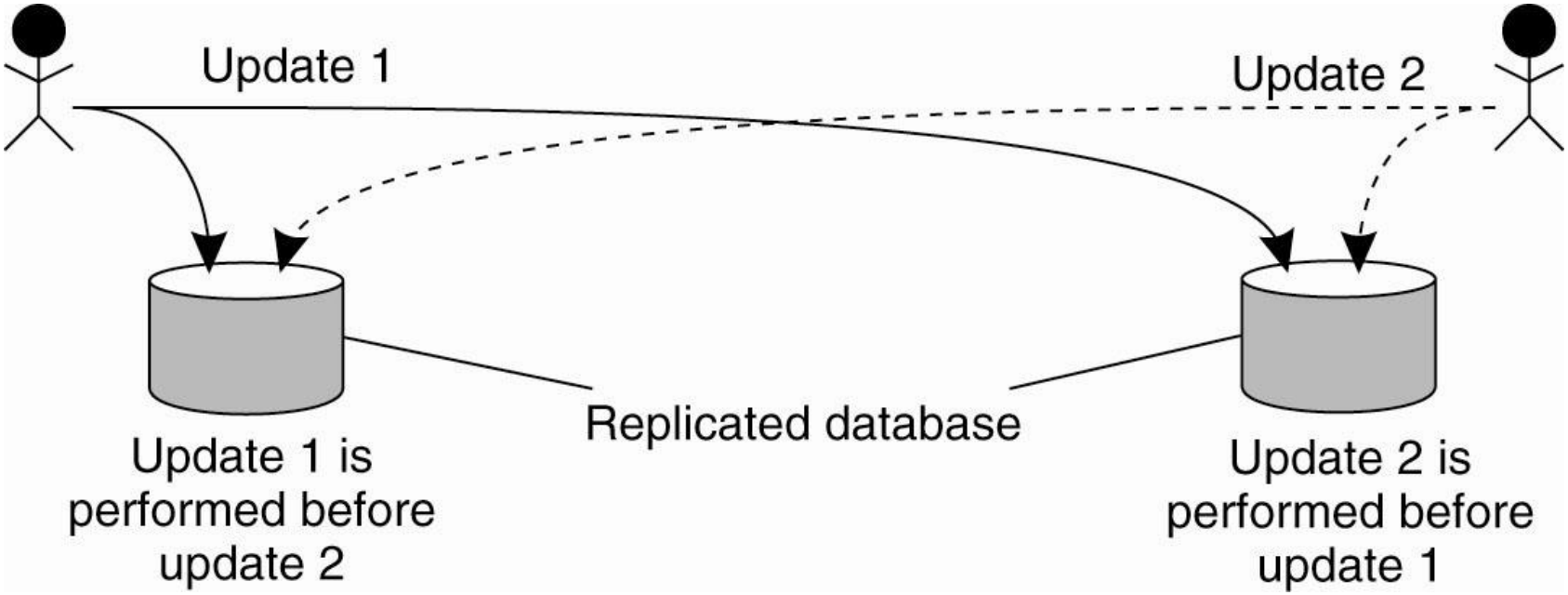
Impact of Clock Synchronization

How does make know which modules need recompiling?



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

Replicated Database Update



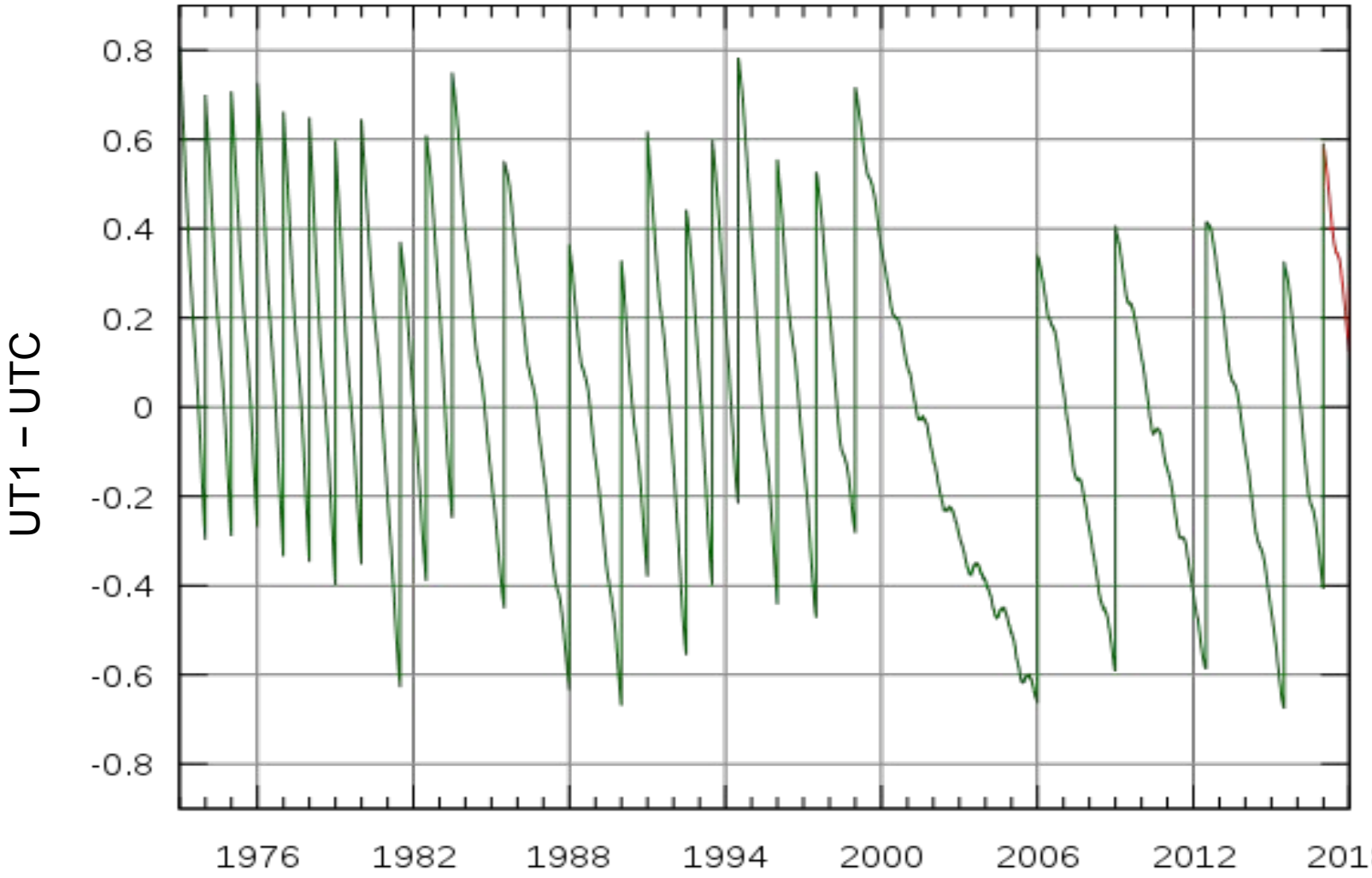
- Updating a replicated database and leaving it in an inconsistent state

Examples where update order matters?

Time Standards

- UT1 (***Universal Time)***
  - Based on astronomical observations
  - “Greenwich Mean Time”
- TAI (***Temps Atomique International)***
  - 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 (***Temps universel coordonné****)*
  - TAI + leap seconds to be within 0.9s of UT1
  - Currently 27 leap seconds
  - Most recent: Dec 31, 2016

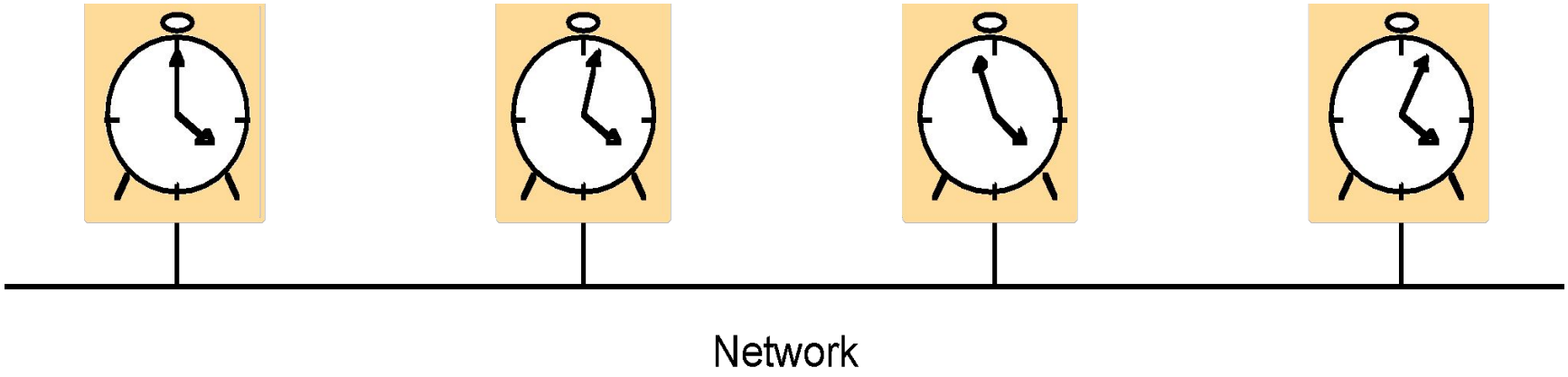
Comparing Time Standards



Coordinated Universal Time (UTC)

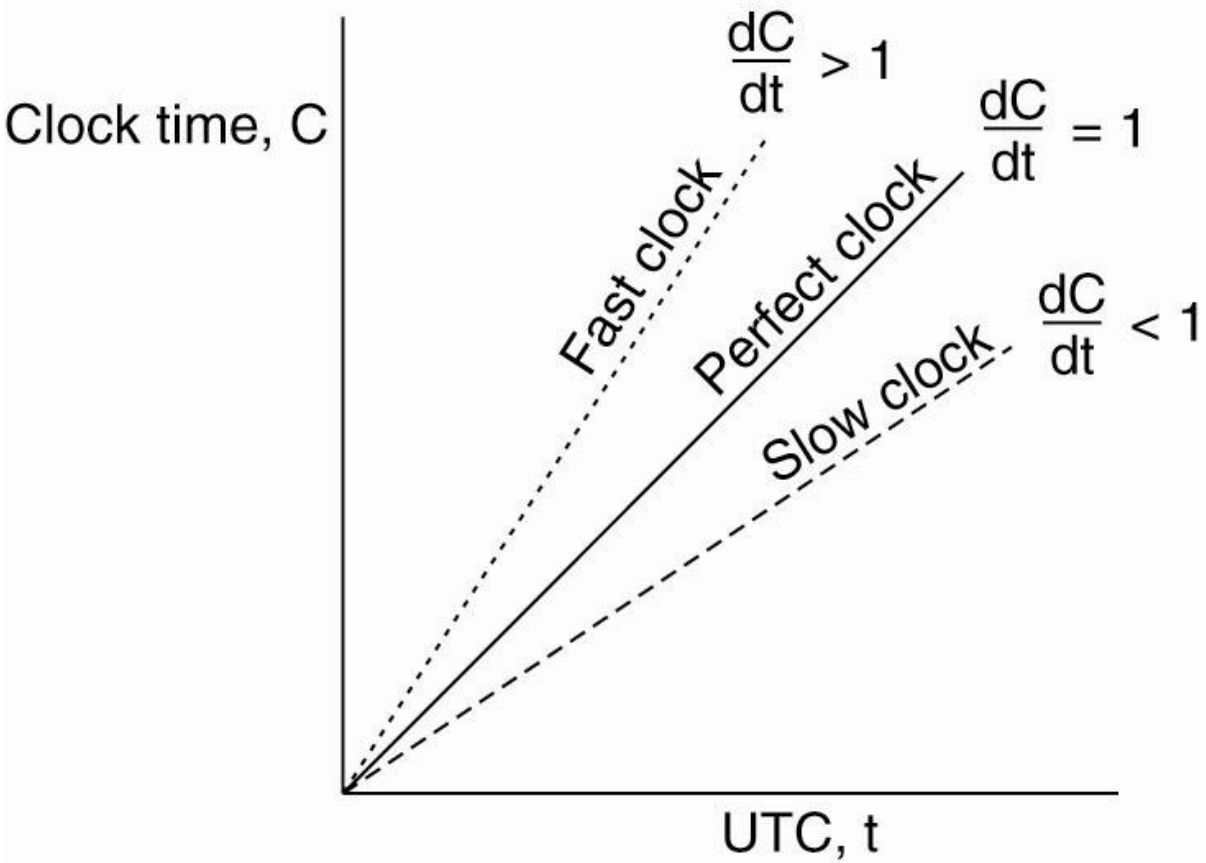
- Is broadcast from radio stations on land and satellite (e.g. GPS)
- Computers with receivers can synchronize their clocks with these timing signals
- Signals from land-based stations are accurate to about 0.1-10 millisecond
- Signals from GPS are accurate to about 1 microsecond
  - Why can't we put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

Clocks in a Distributed System



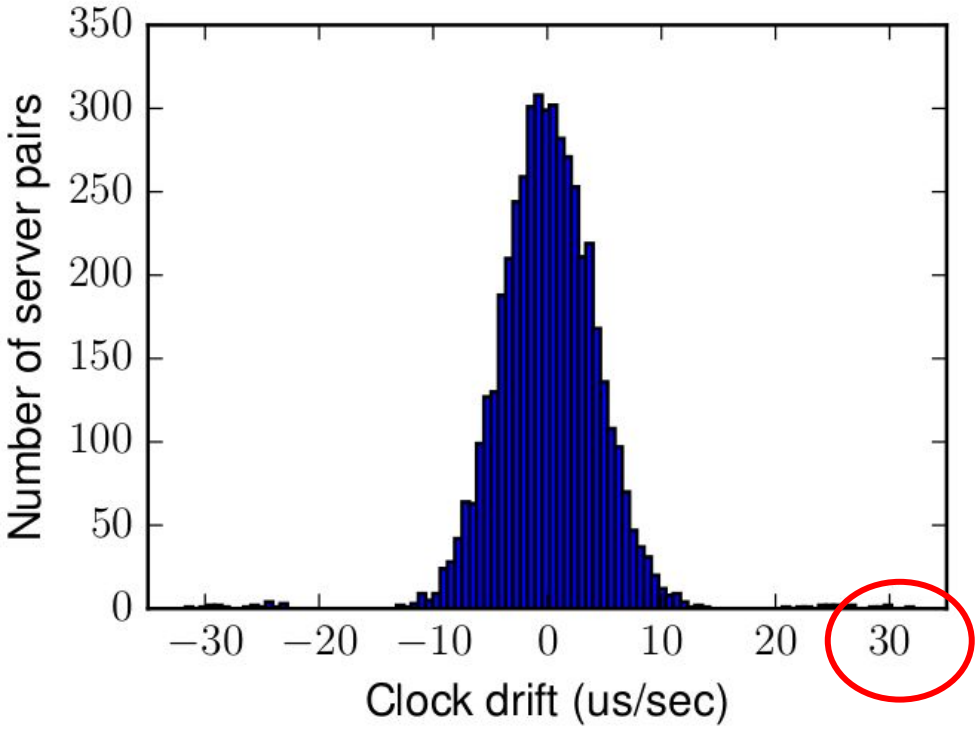
- Computer clocks are not generally in perfect agreement
  - **Skew**: the difference between the times on two clocks (at any instant)
- Computer clocks are subject to clock drift (they count time at different rates)
  - **Clock drift rate**: the difference per unit of time from some ideal reference 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

Fast and Slow Clocks



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

How Fast Do Clocks Drift in Real DS?



Geng, Yilong, et al. "Exploiting a natural network effect for scalable, fine-grained clock synchronization." *NSDI*, 2018.

- After 1min: errors almost 2 milliseconds
- Still assumes constant temperature

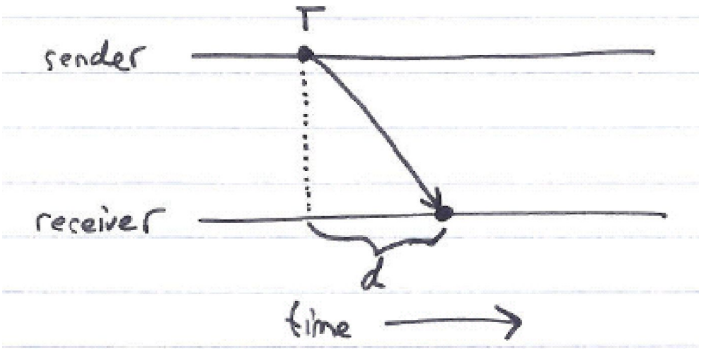
Timestamping datacenter network packets: need nanosecond accuracy!

Today's Lecture

- Need for time Synchronization
- Basic Time Synchronization Techniques
- Lamport Clocks
- Vector Clocks
- Time Synchronization in 2018

Perfect networks

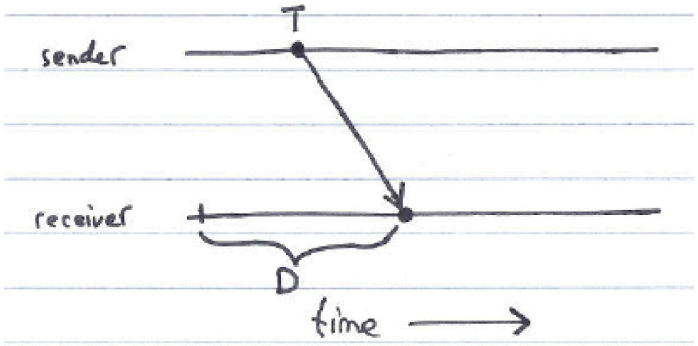
- Messages always arrive, with propagation delay exactly *d*



- 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

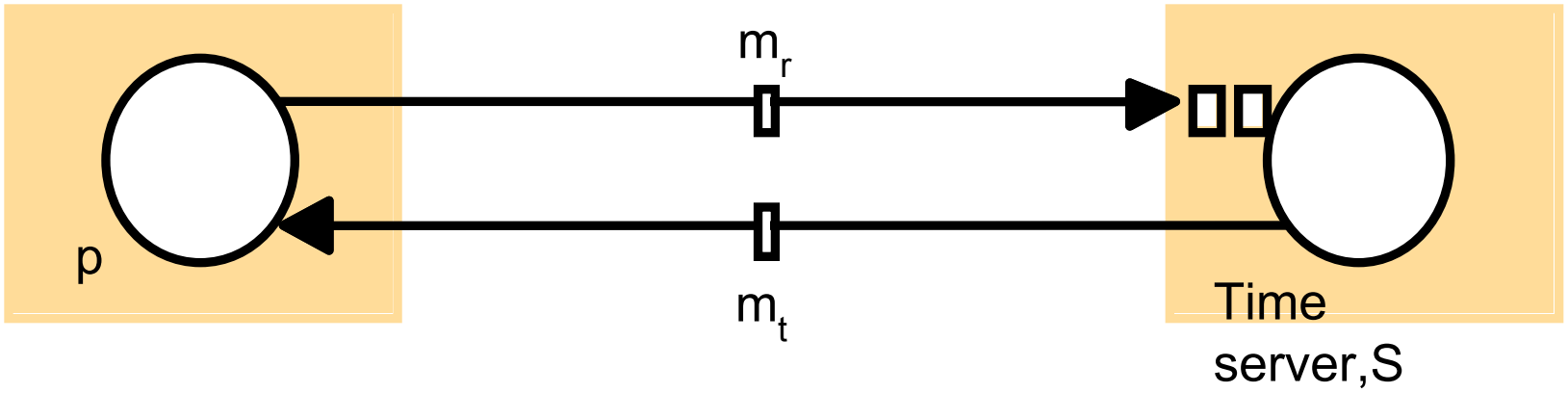
- Real networks are asynchronous
  - Message delays are arbitrary
- Real networks are unreliable
  - Messages don’t always arrive

Today's Lecture

- Need for time Synchronization
- Basic Time Synchronization Techniques
  - Christian's Time Sync
- Lamport Clocks
- Vector Clocks
- Time Synchronization in 2018

Cristian's Time Sync

- 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* + *RTT*/2
  - Accuracy ± (*RTT*/2 - *min*) :
    - 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* + *RTT* - *min*]



RTT is the round trip time recorded by *p*
min is an estimated minimum one way delay

Today's Lecture

- Need for time Synchronization
- Basic Time Synchronization Techniques
  - Christian's Time Sync
  - Berkeley Algorithm
- Lamport Clocks
- Vector Clocks
- Time Synchronization in 2018

Berkeley algorithm

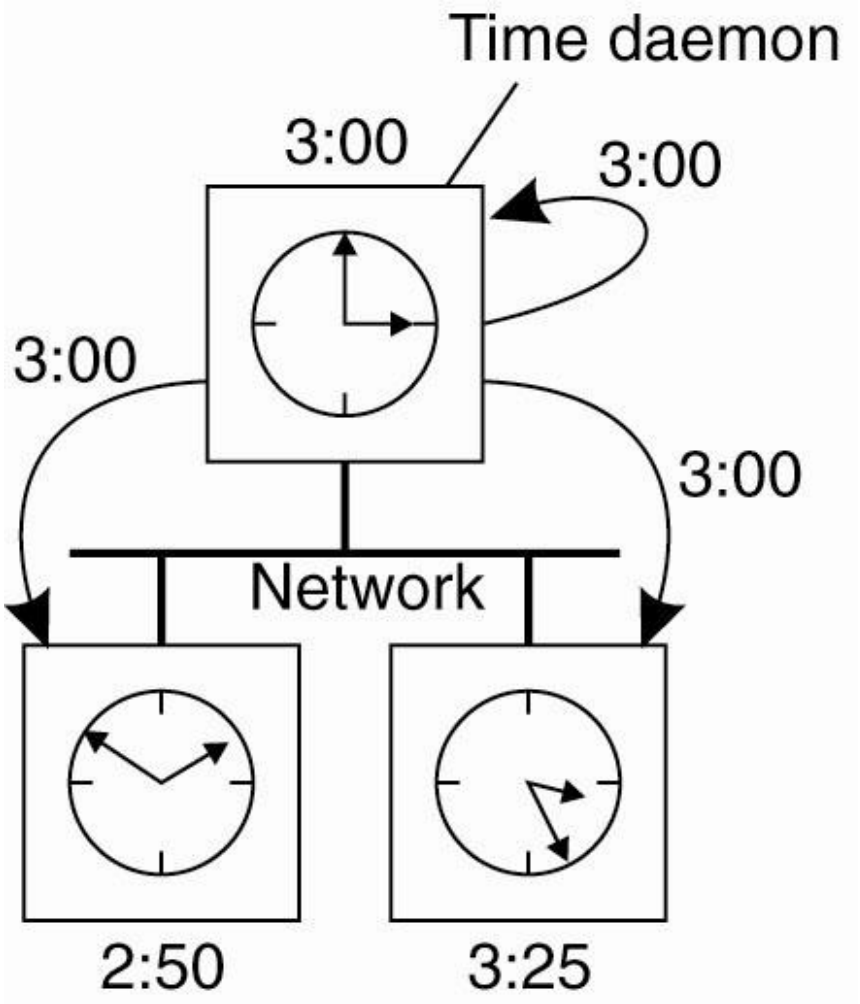
Problems with Christian's Algorithm?

- Cristian's algorithm -
  - a single time server might fail
  - ⇒ group of synchronized servers?
  - ⇒ how to 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
  - 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)
  - Measurements
    - 15 computers, clock synchronization 20-25 millisecs drift rate < 2×10^{-5}
    - If master fails, can elect a new master to take over (not in bounded time)

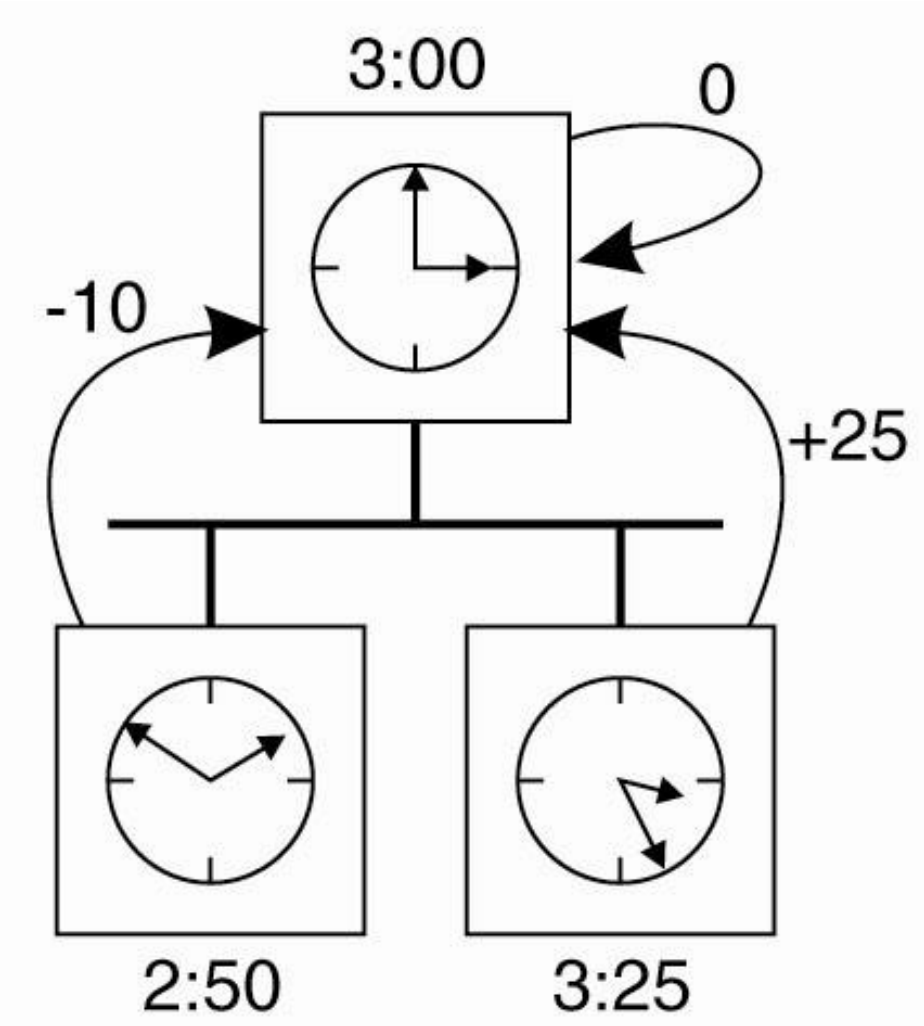
The Berkeley Algorithm (1)

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



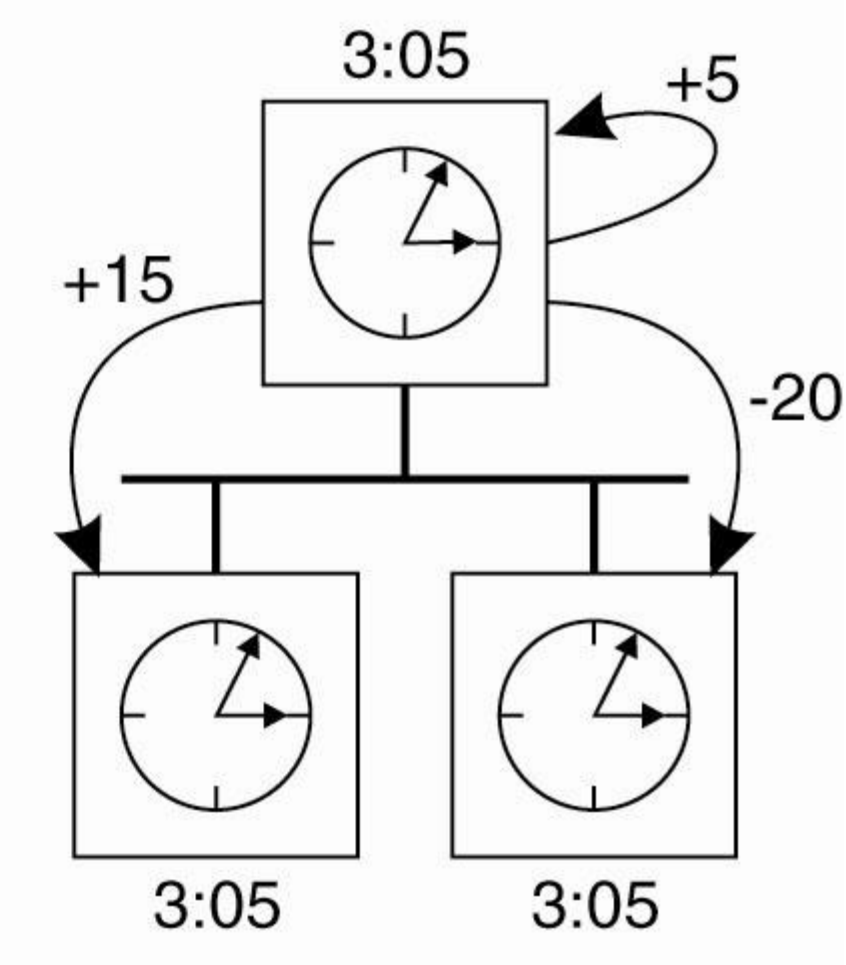
The Berkeley Algorithm (2)

- The machines answer.



The Berkeley Algorithm (3)

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



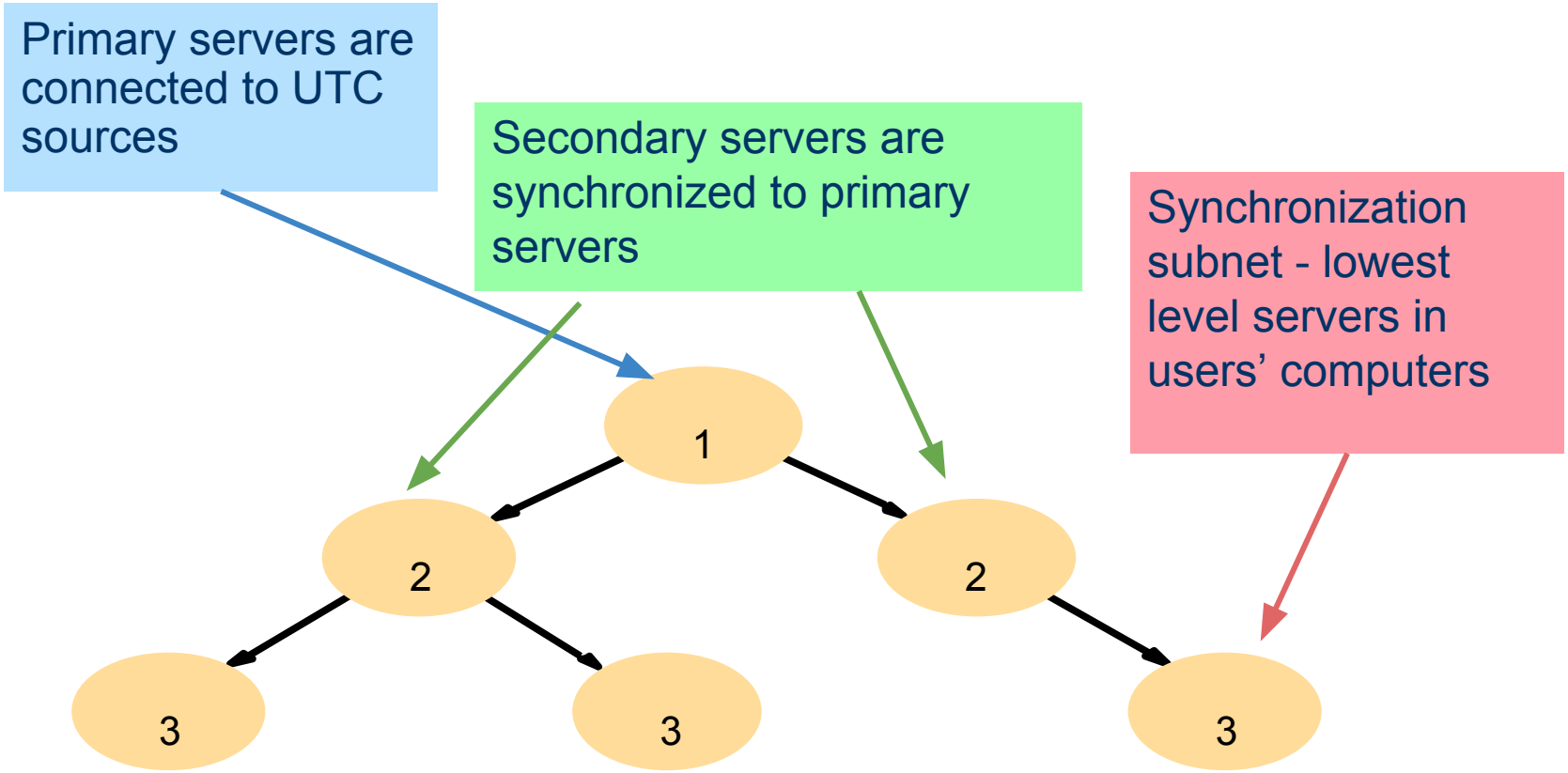
Today's Lecture

- Need for time Synchronization
- Basic Time Synchronization Techniques
  - Christian's Time Sync
  - Berkeley Algorithm
  - NTP
- Lamport Clocks
- Vector Clocks
- Time Synchronization in 2018

Network Time Protocol (NTP)

- A time service for the Internet - synchronizes clients to UTC

Reliability from redundant paths, scalable, authenticates time sources



The Network Time Protocol (NTP)

- 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
- Synchronization similar to Cristian's alg.
  - Modified to use multiple one-way messages instead of immediate round-trip
- Accuracy: Local ~1ms, Global ~10ms

Udel Master Time Facility (MTF)
(since January 2000)

Spectracom 8170 WWVB Receiver

Spectracom 8183 GPS Receiver

Spectracom 8170 WWVB Receiver

Spectracom 8183 GPS Receiver

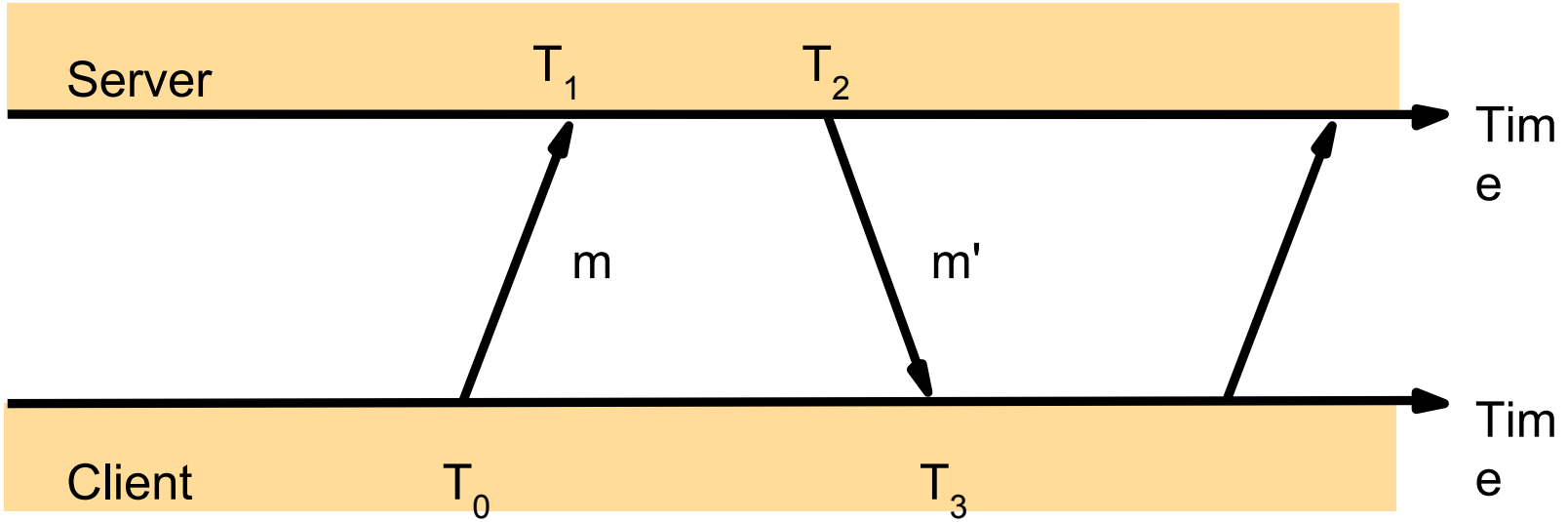
Hewlett Packard 105A Quartz Frequency Standard

Hewlett Packard 5061A Cesium Beam Frequency Standard

Inventor of NTPv0 (today v4): David Mills (http://www.eecis.udel.edu/~mills)

NTP Protocol

- Transport protocol? All messages use UDP
- Each message bears timestamps of recent events:
  - Local times of Send and Receive of previous message
  - Local times of Send of current message
- Recipient notes the time of receipt T_3
- (we have T_0, T_1, T_2, T_3)



Accuracy of NTP

- Timestamps
  - t_0 is the client's timestamp of the request packet transmission,
  - t_1 is the server's timestamp of the request packet reception,
  - t_2 is the server's timestamp of the response packet transmission and
  - t_3 is the client's timestamp of the response packet reception.
- RTT = wait_time_client – server_proc_time
  = $(t_3-t_0) – (t_2-t_1)$
- Offset = $t_2 - t_3 + RTT/2$
  = $((t_1-t_0) + (t_2-t_3))/2$
  = ((offset + delay) + (offset – delay))/2

NTP calls clock skew "offset"

- NTP servers filter pairs <rtt_i, $offset_i$>, estimating reliability from variation, allowing them to select peers
- 8 measurements ⇒ take min packet delay

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

Today's Lecture

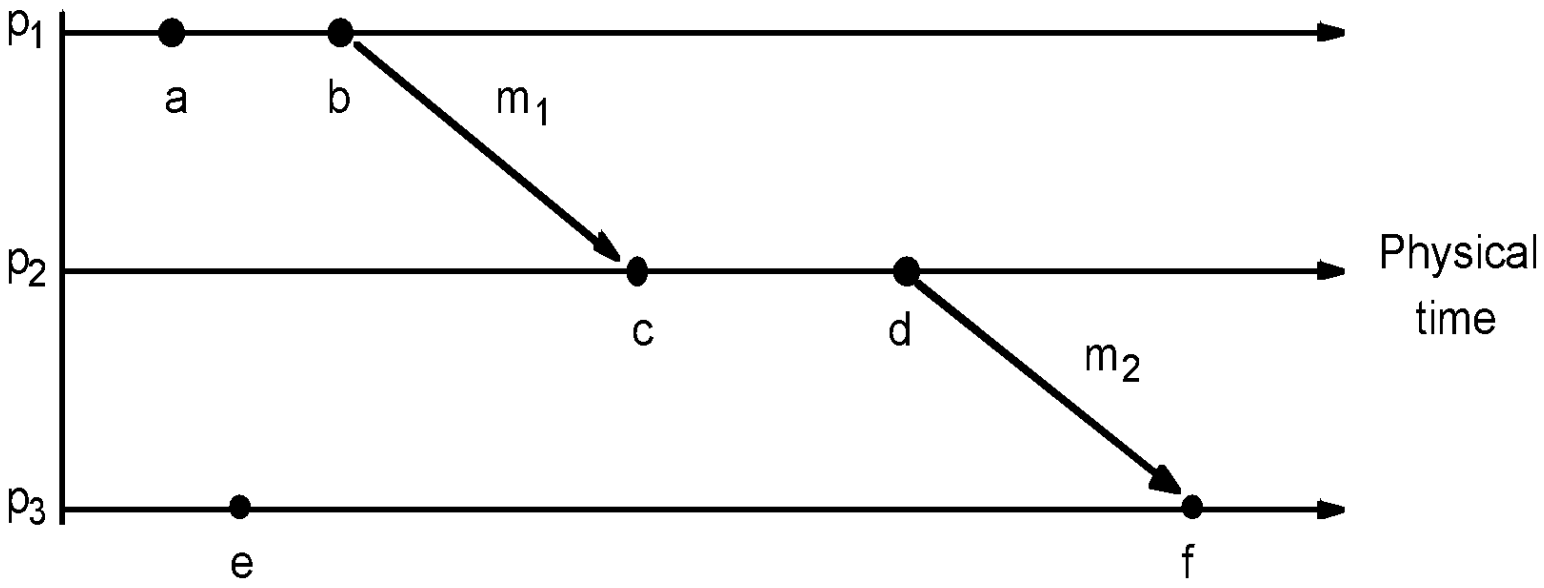
- Need for time Synchronization
- Basic Time Synchronization Techniques
- Lamport Clocks
- Vector Clocks
- Time Synchronization in 2018

Logical time

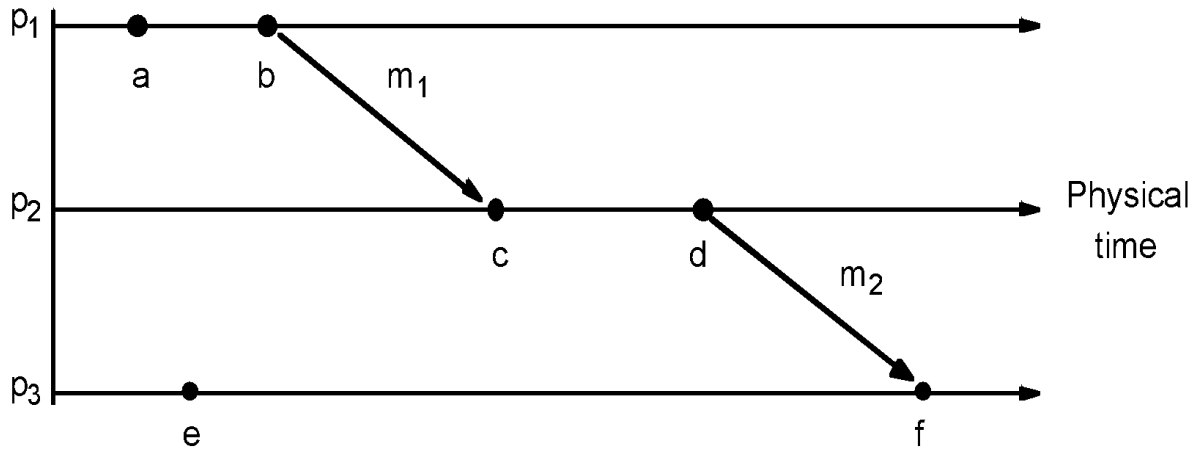
- Capture just the “happens before” relationship between events
  - Discard the infinitesimal granularity of time
  - Corresponds roughly to causality

Logical time and logical clocks (Lamport 1978)

- Events at three processes



Logical time and logical clocks [Lamport 1978]

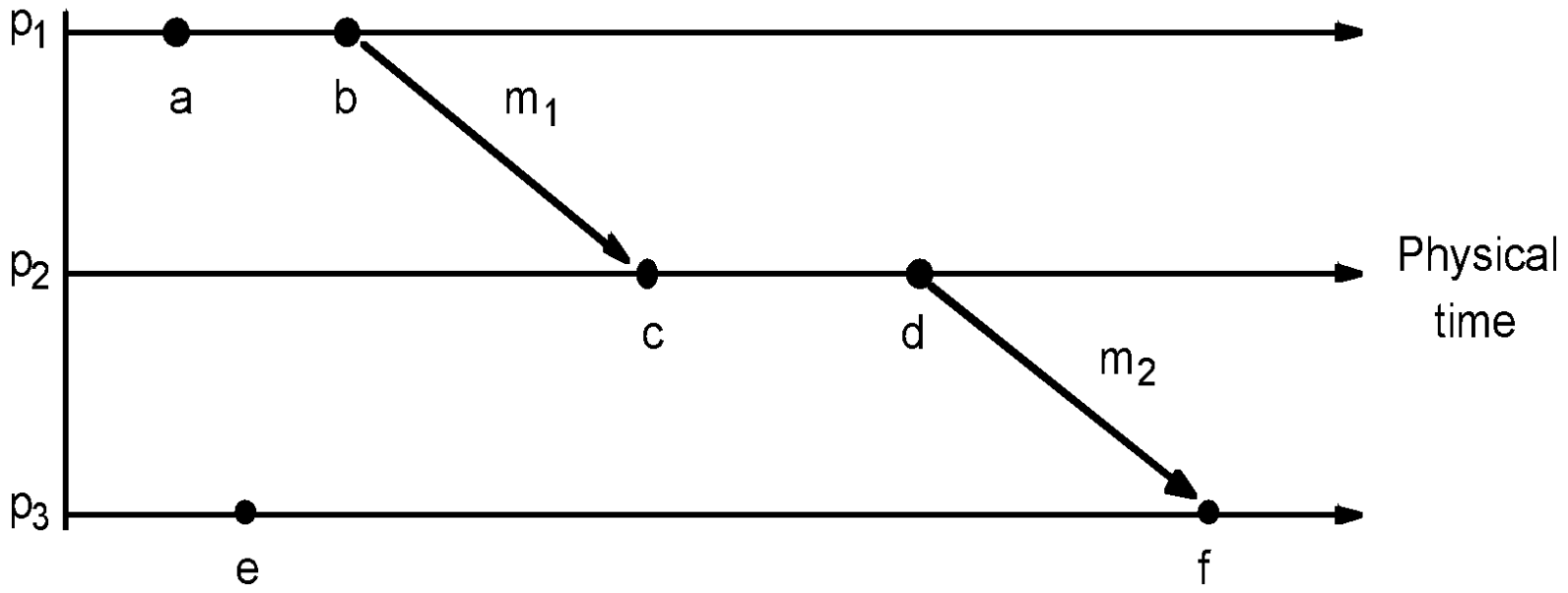


- Instead of synchronizing clocks, event ordering can be used:
  1. Two events occurred at same process p_i (i = 1, 2, … N): then they occurred in the order observed by p_i.

     Definition of: "→ i" ("happened before" i).
  2. When a message, m, is sent between two processes: send(m) happens before receive(m).
  3. The "happened before" relation is transitive.

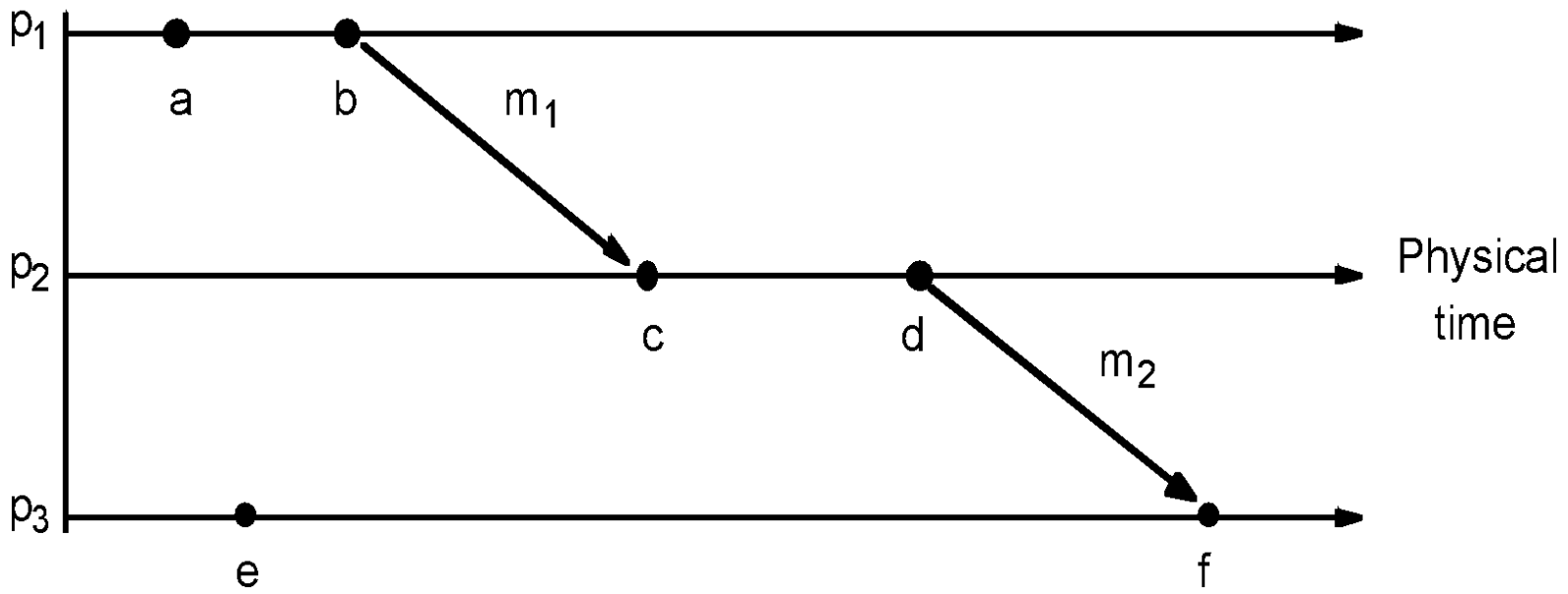
- The happened before relation is the relation of causal ordering

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
- so: $a \rightarrow f$

Logical time and logical clocks [Lamport 1978]

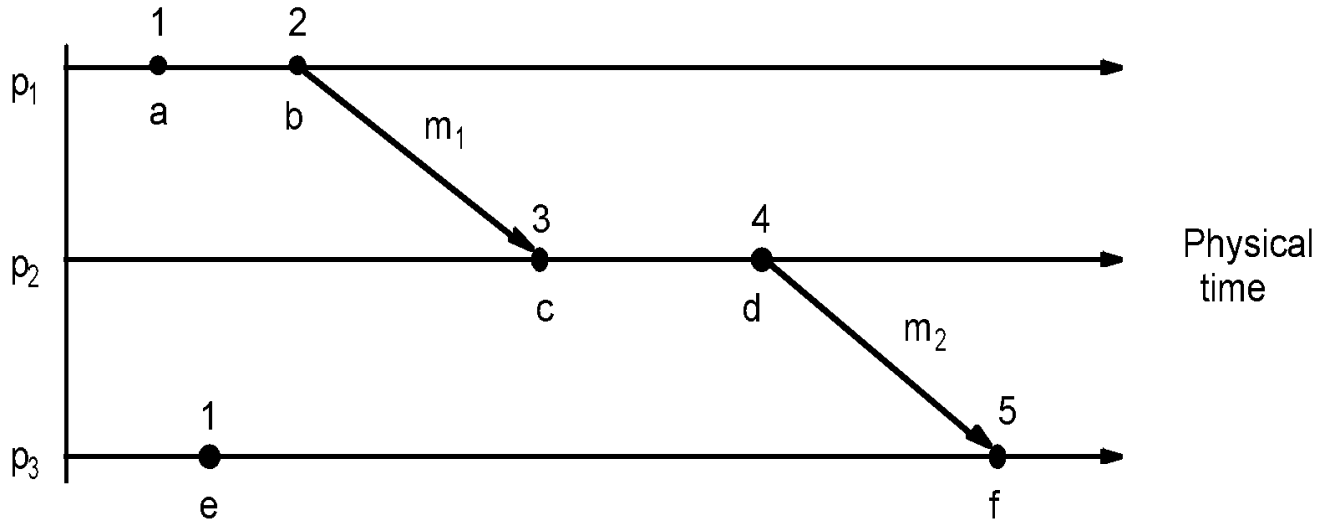


- Not all events are related by →
- Consider *a* and *e* (different processes and no chain of messages to relate them)
  - they are not related by → ; they are said to be concurrent
  - written as *a* || *e*

Lamport's algorithm

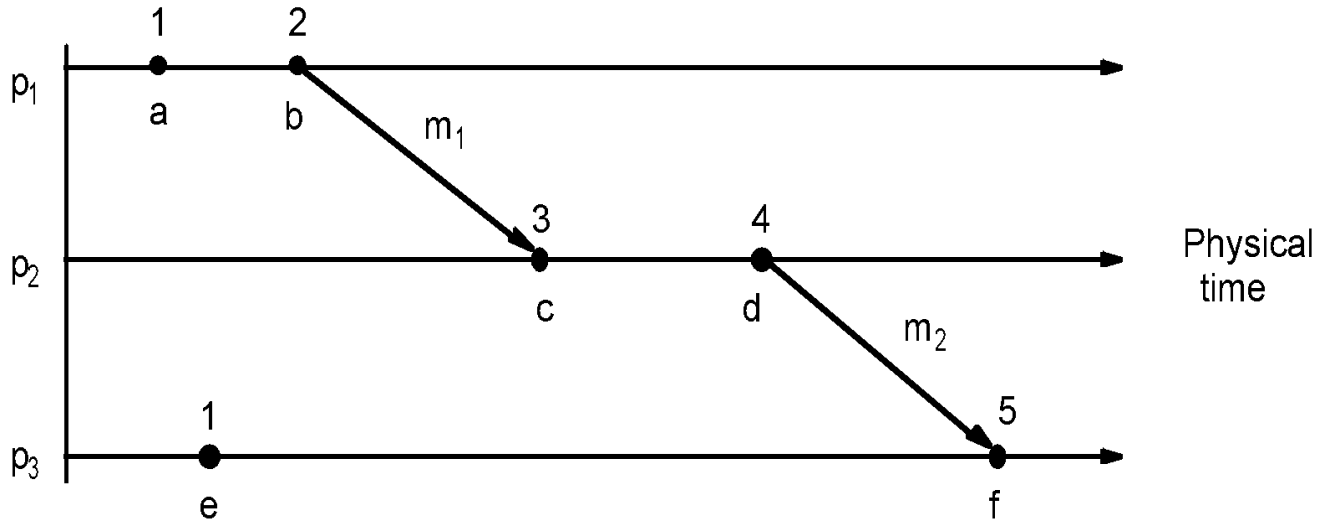
- Each process *i* keeps a local clock, L_i
- Three rules:
  1. At process *i*, increment L_i before each event
  2. To send a message *m* at process *i*, apply rule 1 and then include the current local time in the message: i.e., $send(m,L_i)$
  3. To receive a message (m,t) at process *j*, set $L_j = max(L_j,t)$ and then apply rule 1 before time-stamping the receive event
- The global time $L(e)$ of an event *e* is just its local time
  - For an event *e* at process *i*, $L(e) = L_i(e)$

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 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_i receives (m,t) it sets $L_j := max(L_j, t)$ and applies rule 1 before timestamping the event *receive* (m)

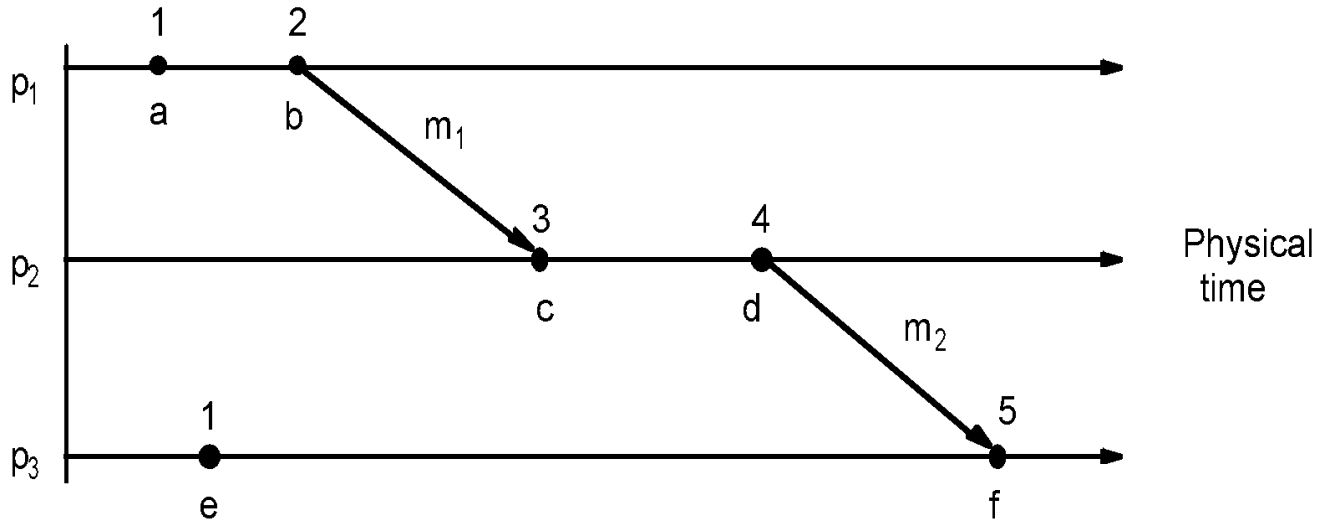
Lamport Clocks (2)



- 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.

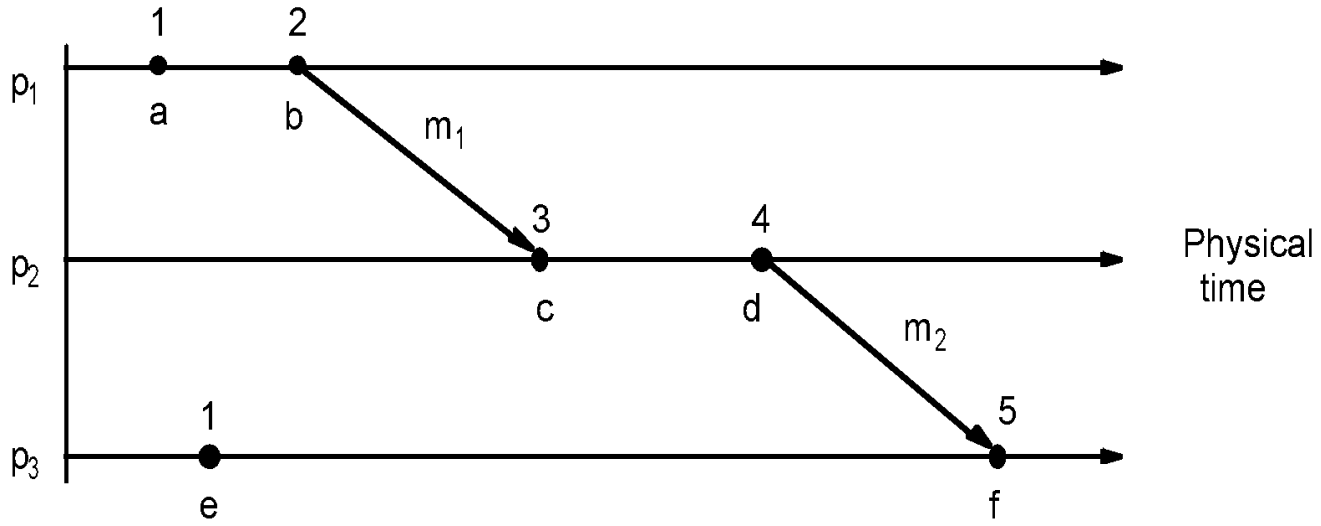
- For m_1, 2 is piggybacked and c gets max(0,2)+1 = 3

Lamport Clocks (3)



- $e \rightarrow e'$ 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 Clocks (4)



- Similar rules for concurrency
  - *L(e) = L(e')* implies $e \parallel e'$ (for distinct *e,e'*)
  - $e \parallel e'$ 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
  - $L(e) = M * L_i(e) + i$
    - M = maximum number of processes
    - i = process ID

Practice a few examples of total-order Lamport clocks at home!

Today's Lecture

- Need for time Synchronization
- Basic Time Synchronization Techniques
- Lamport Clocks
- Vector Clocks
- Time Synchronization in 2018

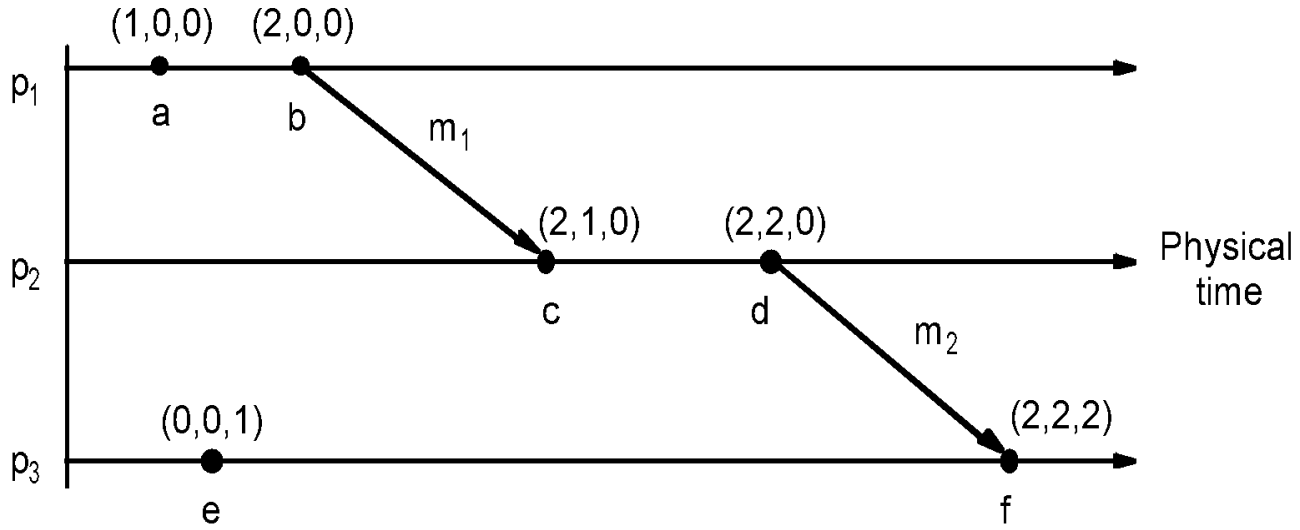
Vector Clocks

- 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 causality
  - 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 causally precede e

Vector Clock Algorithm

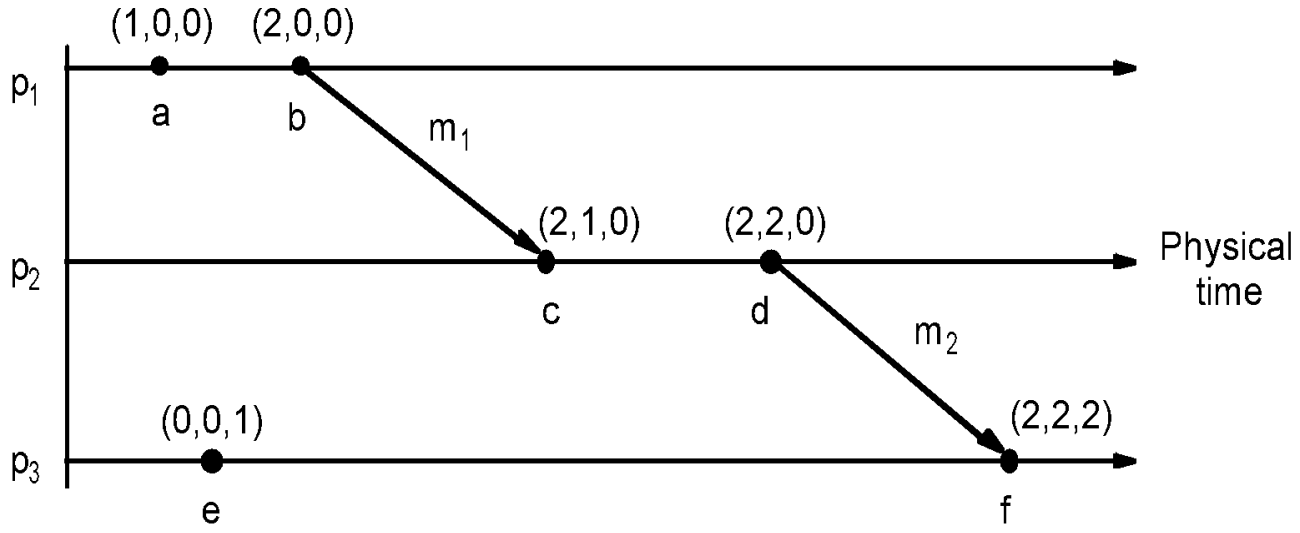
- Initially, all vectors [0,0,…,0]
- For event on process i, increment own c_i
- Label message sent with local vector
- When process j receives message with vector $[d_1, d_2, \ldots, d_n]$:
  - Set local each local entry k to $\max(c_k, d_k)$
  - Increment value of c_j

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 parallel events?
  - *c* || *e* (parallel) because neither *V*(c) <= *V*(*e*) nor *V*(*e*) <= *V*(*c*)

Today's Lecture

- Need for time Synchronization
- Basic Time Synchronization Techniques
- Lamport Clocks
- Vector Clocks
- Time Synchronization in 2018

The State of Time Synchronization in 2018

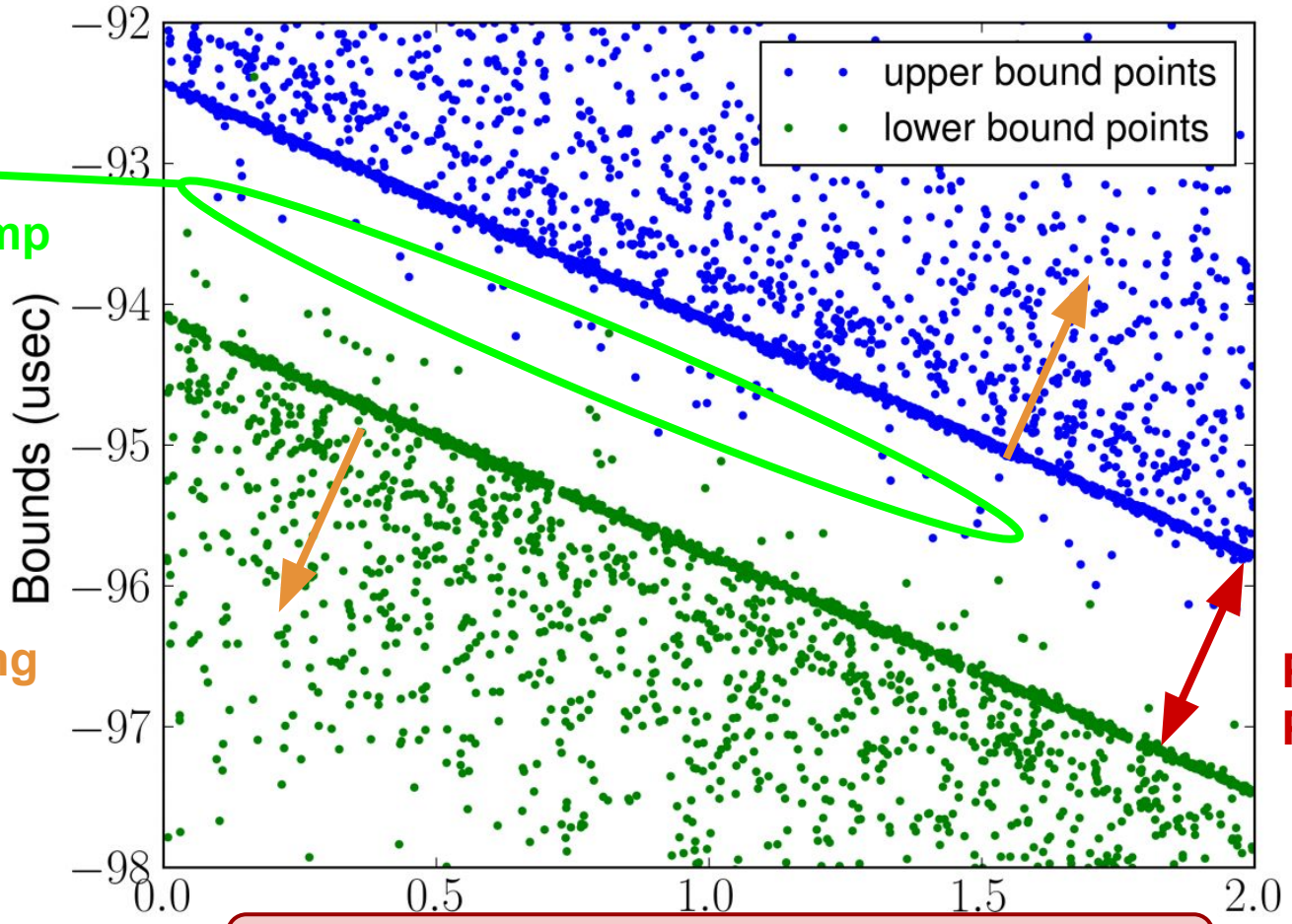
Active Research Field

- With hardware support [DTP, SIGCOMM 2016]
  - Exploit PHY-level (Ethernet) hop-by-hop synchronization
  - 25.6 nanosecond-accuracy (single hop)

- (Mostly) in software [HUYGENS, NSDI 2018]
  - NIC timestamp support
  - ML-based software filter
  - 100 nanosecond-accuracy (within small datacenter)

Problem Observations in HUYGENS 1

Bidirectional probes ⇒ upper/lower bounds on clock delta



???

Timestamp errors

Queueing delay

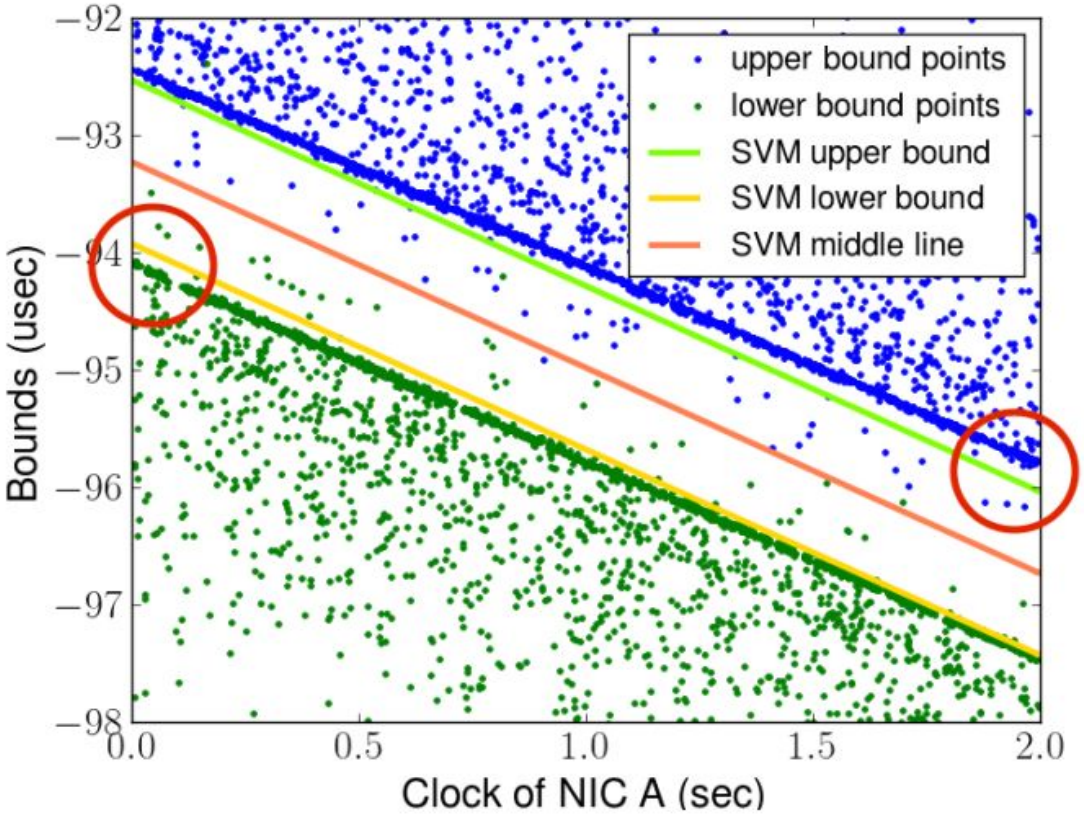
Queueing delay

Proc. + Prop. delay

Very noisy! How to detect boundary?

Problem Observations in HUYGENS 2

Why not use statistical (ML) techniques to denoise?

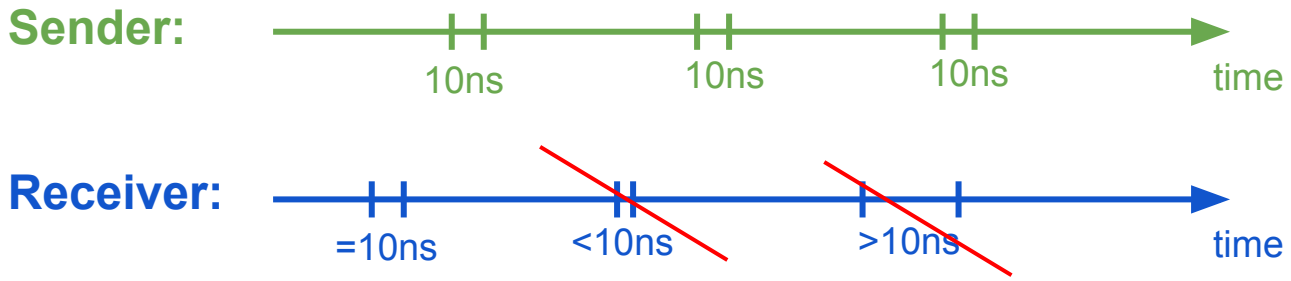


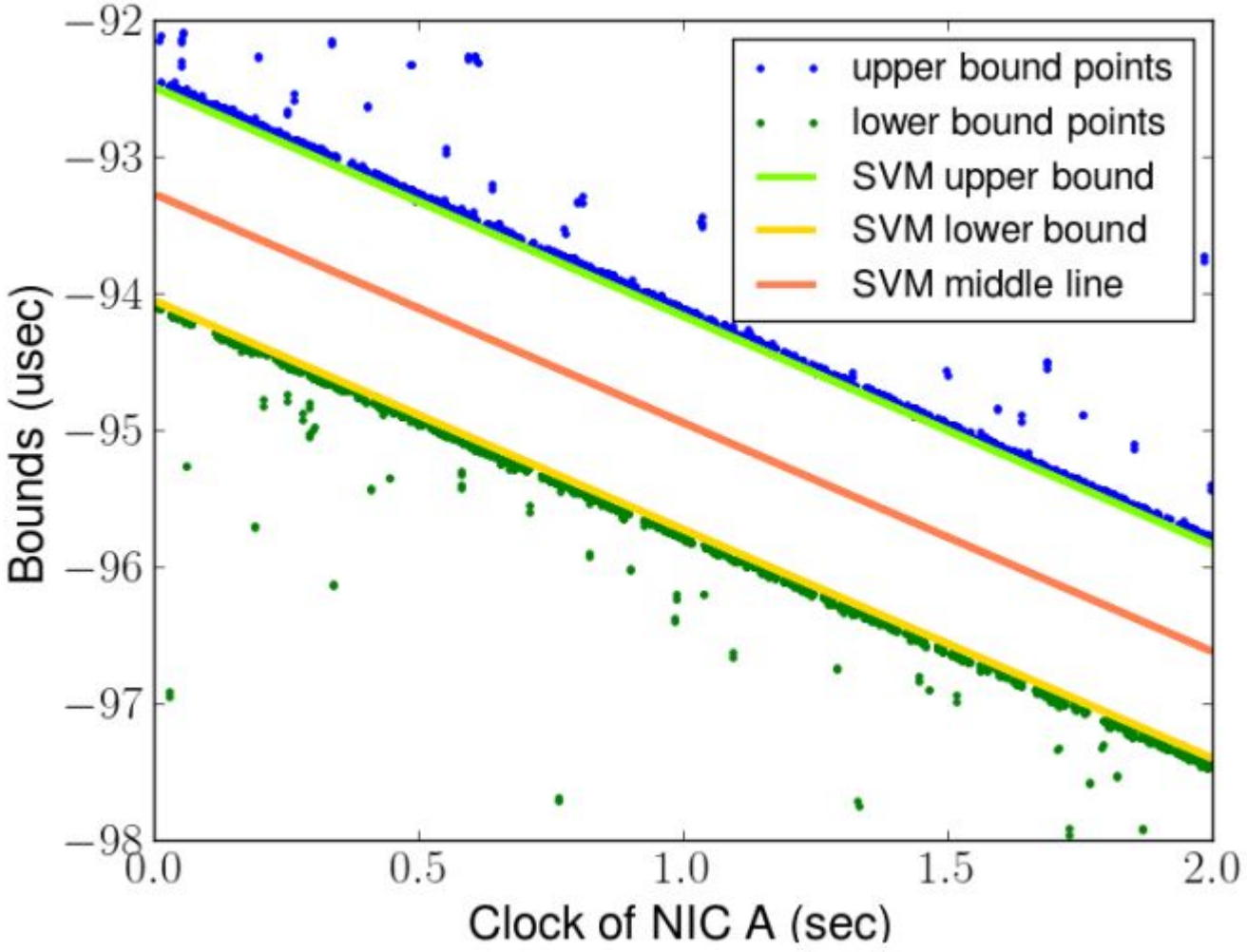
Incorrectly detects boundary!

What else can we do? Can DS ideas help?

Key Idea:

Coded probes





Today's Lecture

- Need for time Synchronization
- Basic Time Synchronization Techniques
- Lamport Clocks
- Vector Clocks
- Time Synchronization in 2018

Clock Sync Important Lessons

- Clocks on different systems will always behave differently
  - Skew and drift between clocks
- Time disagreement between machines can result in undesirable behavior
- Two paths to solution: synchronize clocks or ensure consistent clocks
- We will revisit this when we get to the Spanner lecture
  - Basically, can we slow down the system to make synchronizing clocks an effective solution

Clock Sync Important Lessons

- 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 very high precision

- Logical Clocks
  - Encode causality relationship
  - Lamport clocks provide only one-way encoding
  - Vector clocks provide exact causality information