Announcements (1)

- Project 1 (P1) is coming out this week (dates on course website)
  - P1 recitation next week

- Find your partner for P1
  - Solo teams not allowed
  - **Fill out the survey on team declaration before Wed 11:59pm (pinned post on Piazza)**
    - Only one entry per team needed – please do not respond multiple times
    - Via the form you can also inform us if you want us to find a partner for you

- HW1 is coming out this week (dates on course website)

- Recall: No debugging help on the day of the deadline
  - Some TA office hours moved earlier – thanks to the amazing TA team!

Announcements (2)

- For everyone's safety please do not congregate after the class for Q/A
  - Ask during the lecture or make use of Piazza and office hours

- For any private communication, use course staff email < ds-staff-f21-private@lists.andrew.cmu.edu>. Not individual instructor email addresses!

15-440/15-640 Distributed Systems

Time Synchronization

Agenda

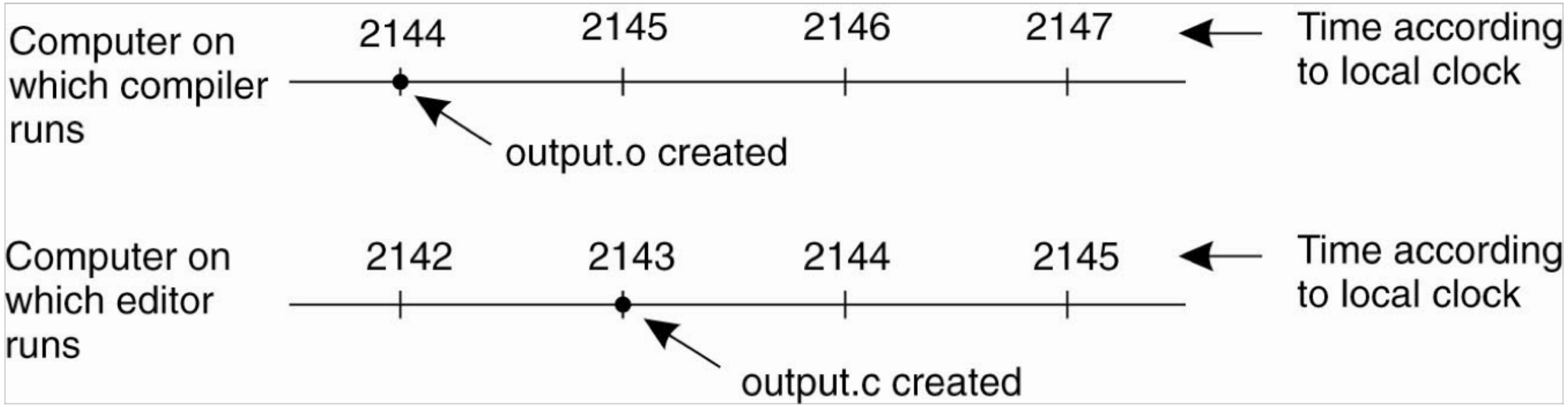
- 👉 Need for time Synchronization
- 👉 Basic Time Synchronization Techniques
- 👉 Lamport Clocks
- 👉 Vector Clocks

Agenda

- 👉 Need for time Synchronization
- 👉 Basic Time Synchronization Techniques
- 👉 Lamport Clocks
- 👉 Vector Clocks

Impact of Clock Synchronization

Think of Unix make. 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.

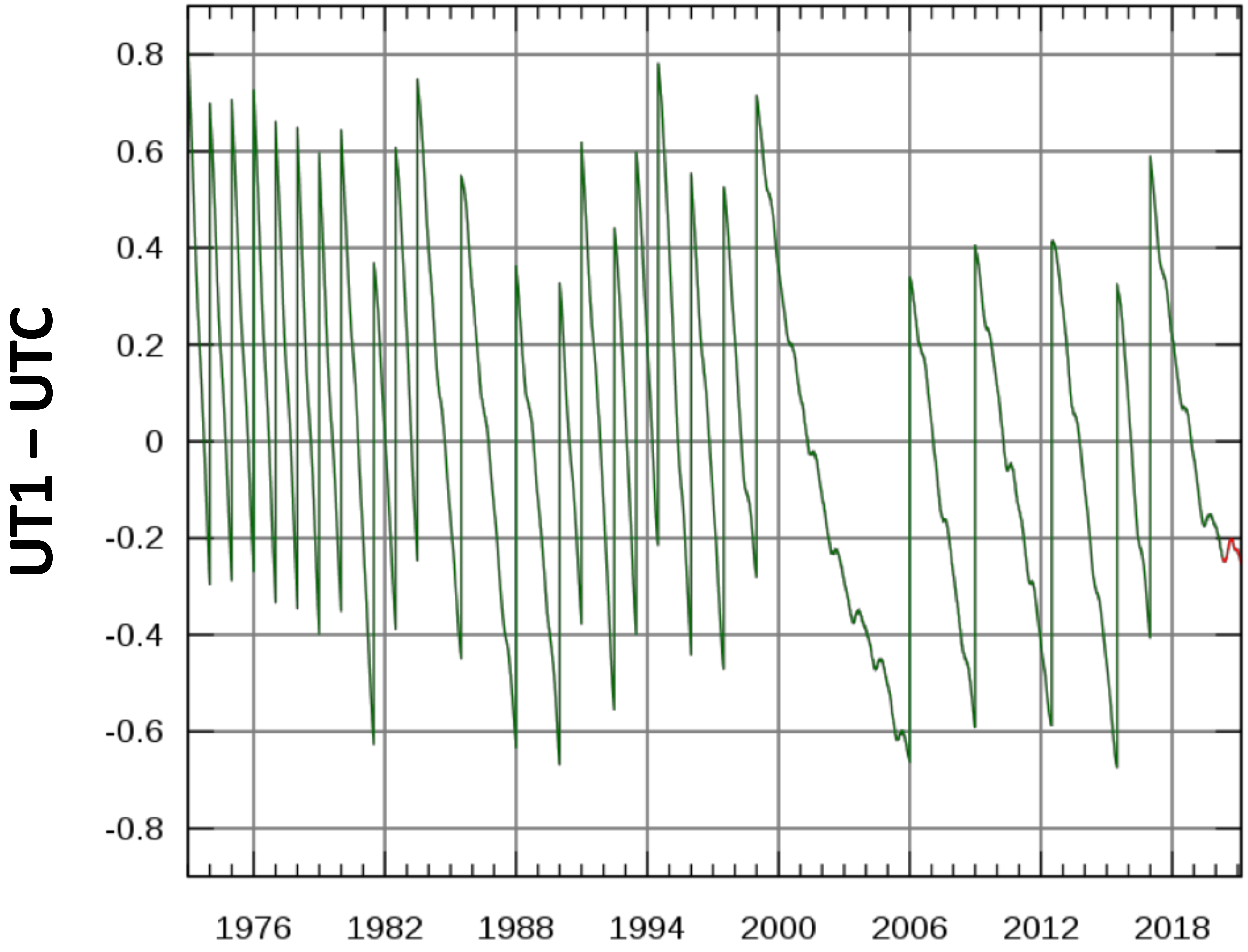
Time Standards

UT1 (**Universal Time**)
- Based on astronomical observations
- "Greenwich Mean Time"

TAI (**Temps Atomique International / 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 (**Temps universel coordonné/ Universal Coordinated Time**)
- TAI + leap seconds to be within 0.9s of UT1
- Currently 27 leap seconds
- Most recent: Dec 31, 2016

Comparing Time Standards

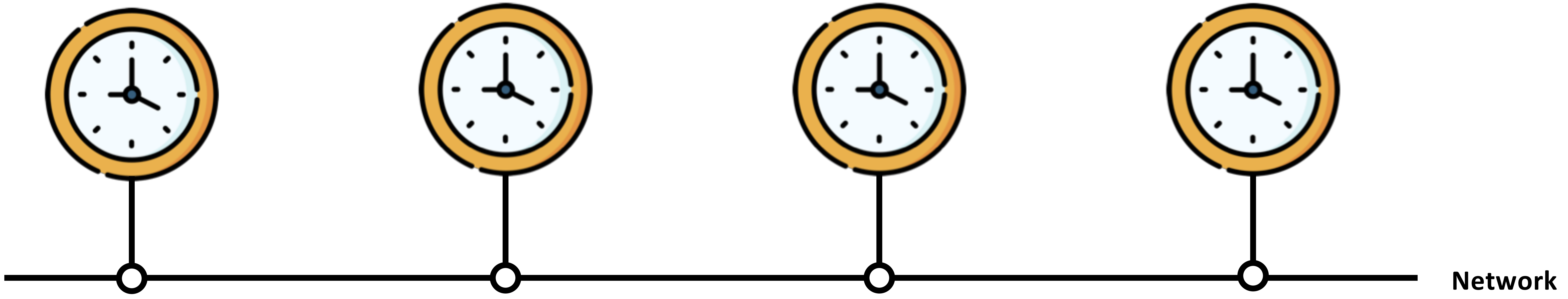


Universal Coordinated Time (UTC)

- 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

Q: Why can't we put GPS receivers on all our computers?

Clocks in a Distributed System



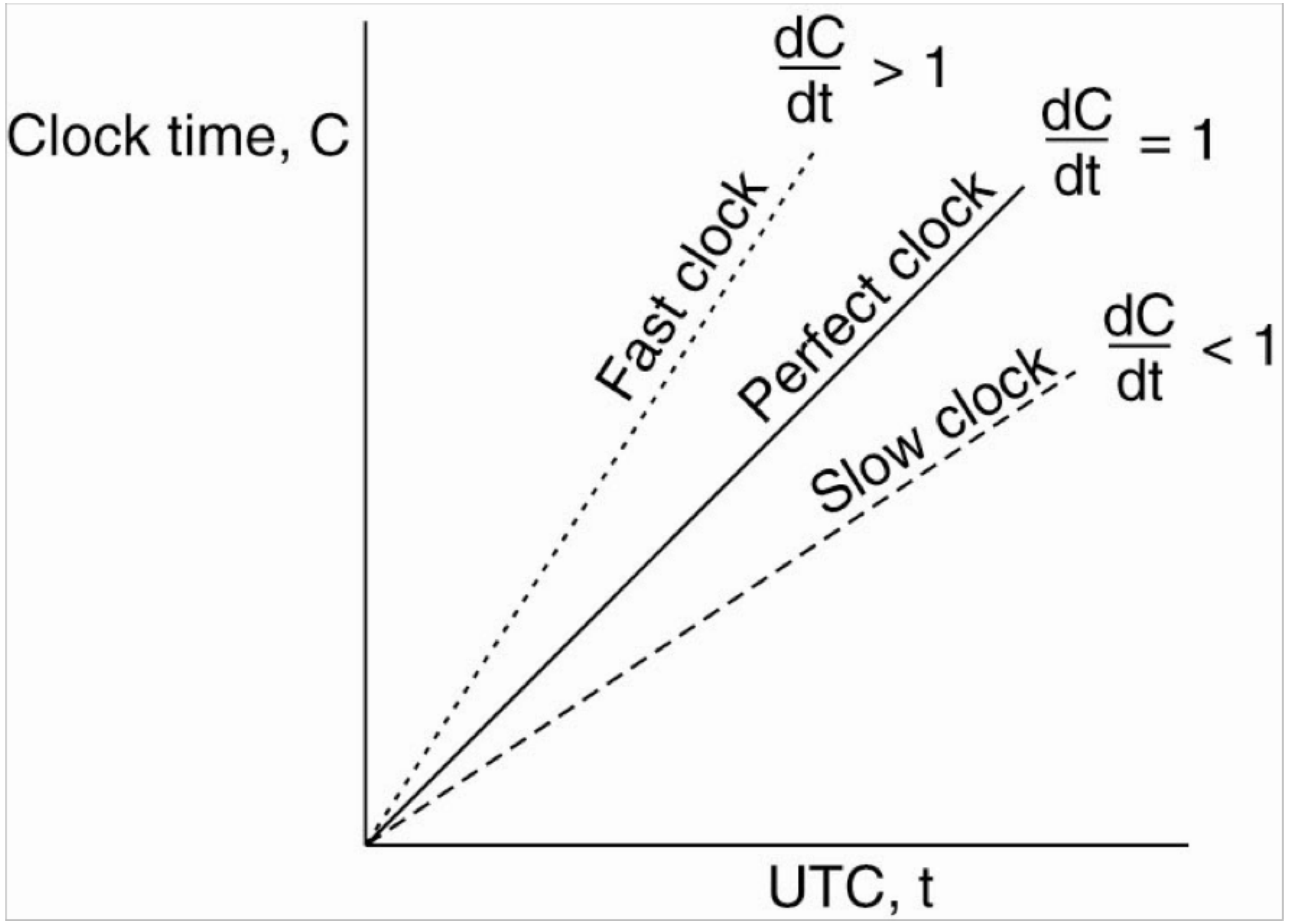
Computer clocks are not generally in perfect agreement

- **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)

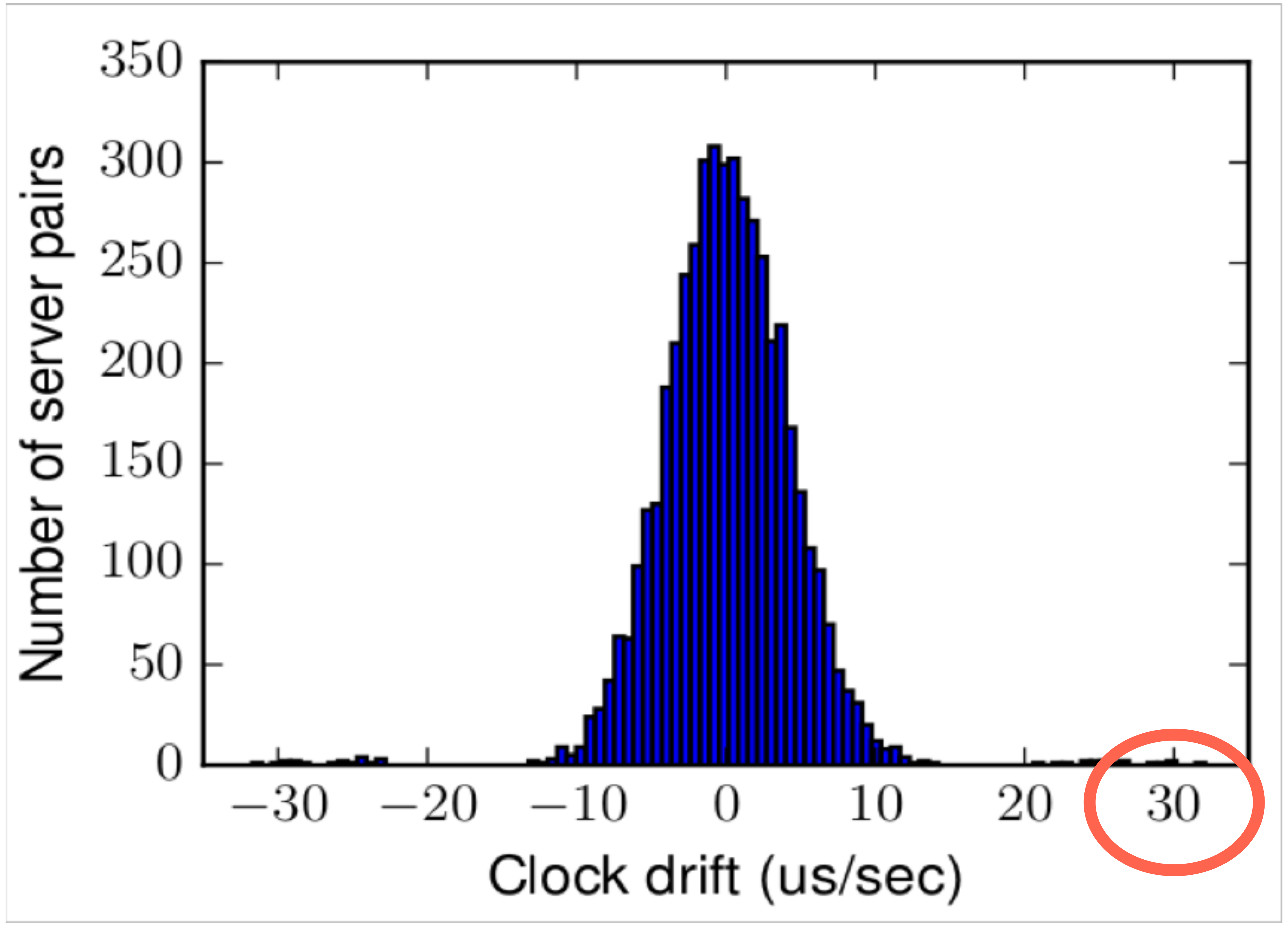
- **Clock drift rate**: the difference per unit of time from some ideal reference clock
- Ordinary quartz clocks drift by about 1 sec in 11-12 days (10^{-6} secs/sec).
- High precision quartz clocks drift rate is about 10^{-7} or 10^{-8} secs/sec

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.

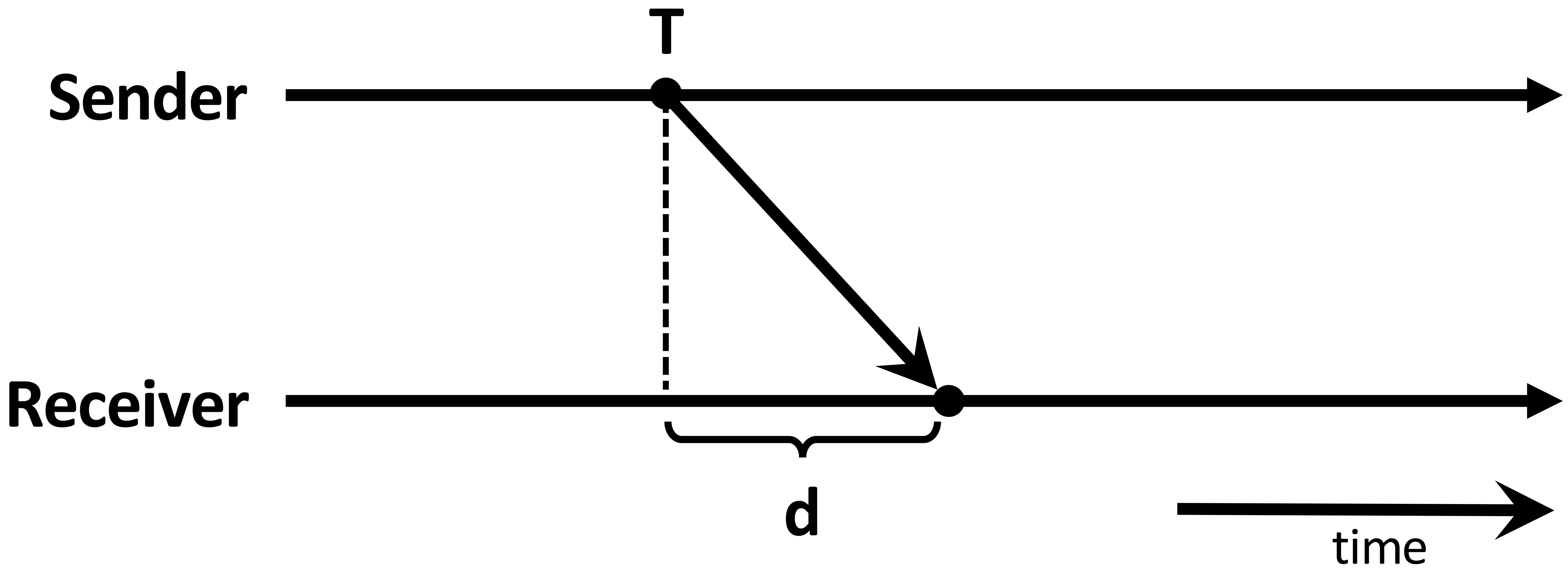
- 👉 After 1 minute, errors almost 2 milliseconds
- 👉 Still assumes constant temperature

Timestamping datacenter network packets: need nanosecond accuracy!

Agenda

- Need for time Synchronization
- **Basic Time Synchronization Techniques**
- Lamport Clocks
- Vector Clocks
- Time Synchronization in Recent Years

Perfect Networks



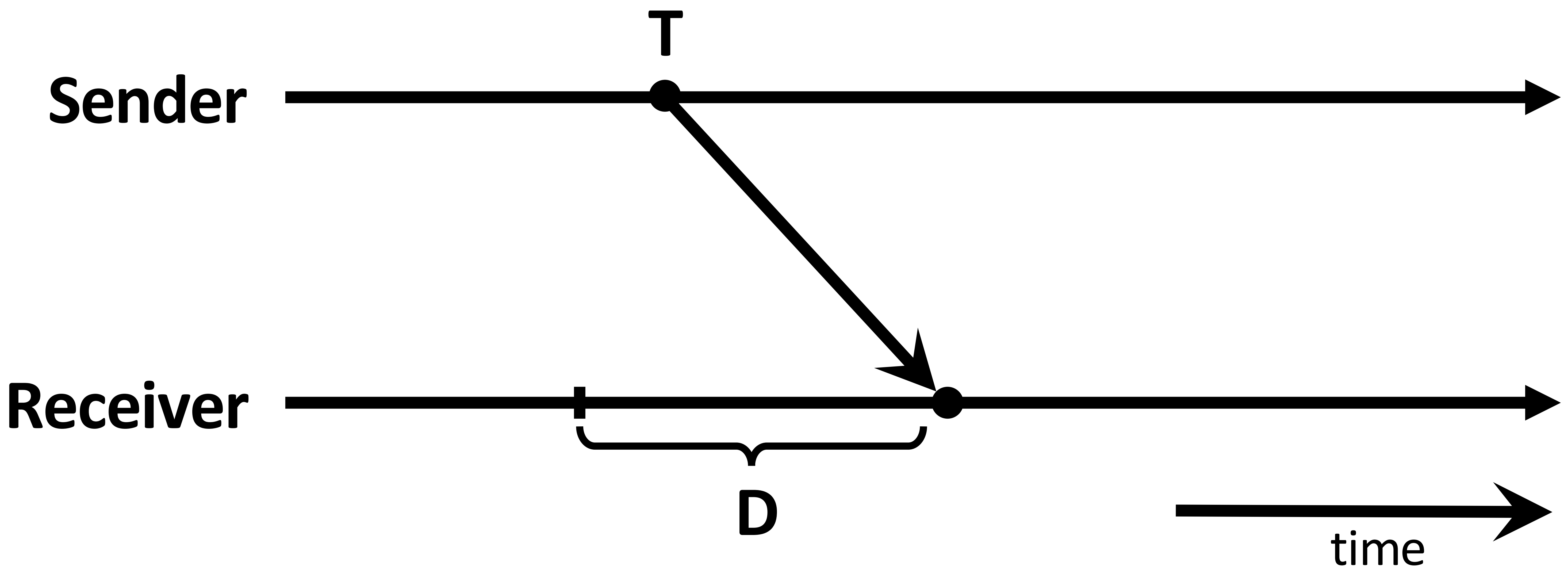
Say, messages always arrive with propagation delay exactly d

Sender sends time T in a message
Receiver sets clock to T + d

- Synchronization is exact

What is the problem here?

Synchronous Networks



Say, messages always arrive with propagation delay at most D

Sender sends time T in a message
Receiver sets clock to T + D/2

- What is the bound on synchronization error?

Synchronous 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

Setting:

A time server *S* receives signals from a UTC source

Process *p wants to know the time*

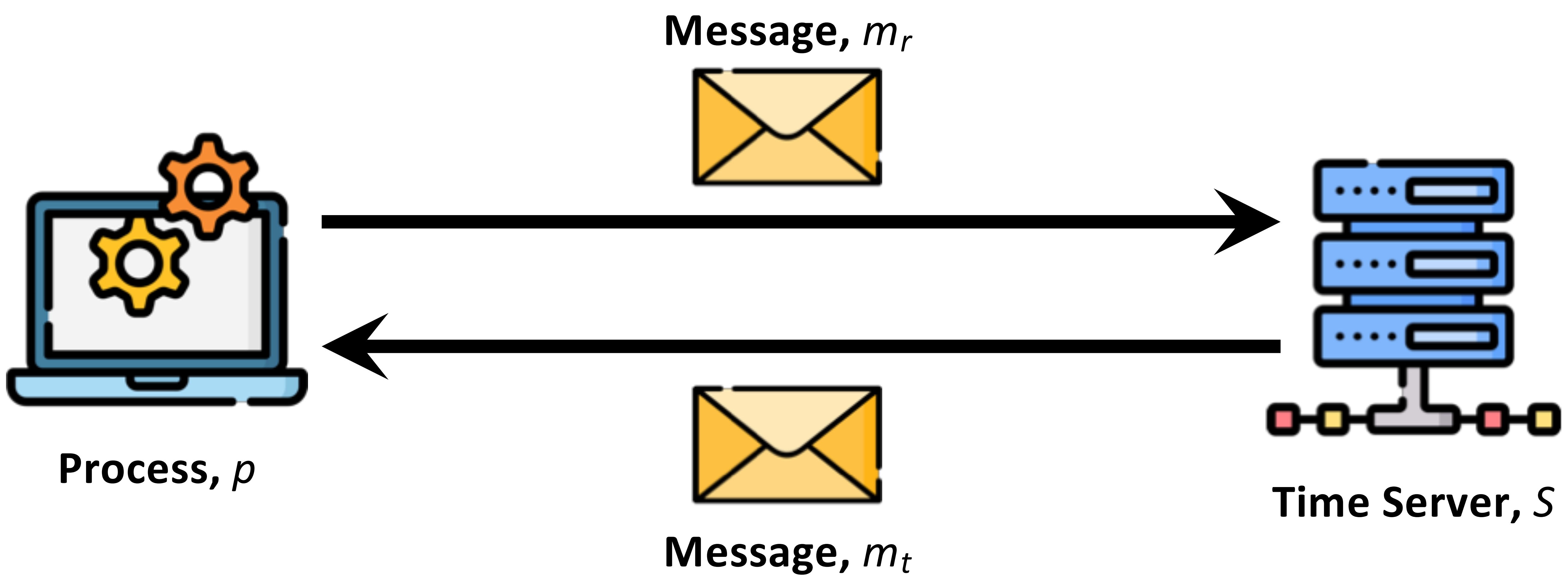
How can process p get to know the time?

Cristian's Time Sync

Setting:

A time server *S* receives signals from a UTC source

Process *p wants to know the time*



How?

- Process p requests time in m_r and receives time value t in m_t from S
- p sets its clock to $t + RTT/2$

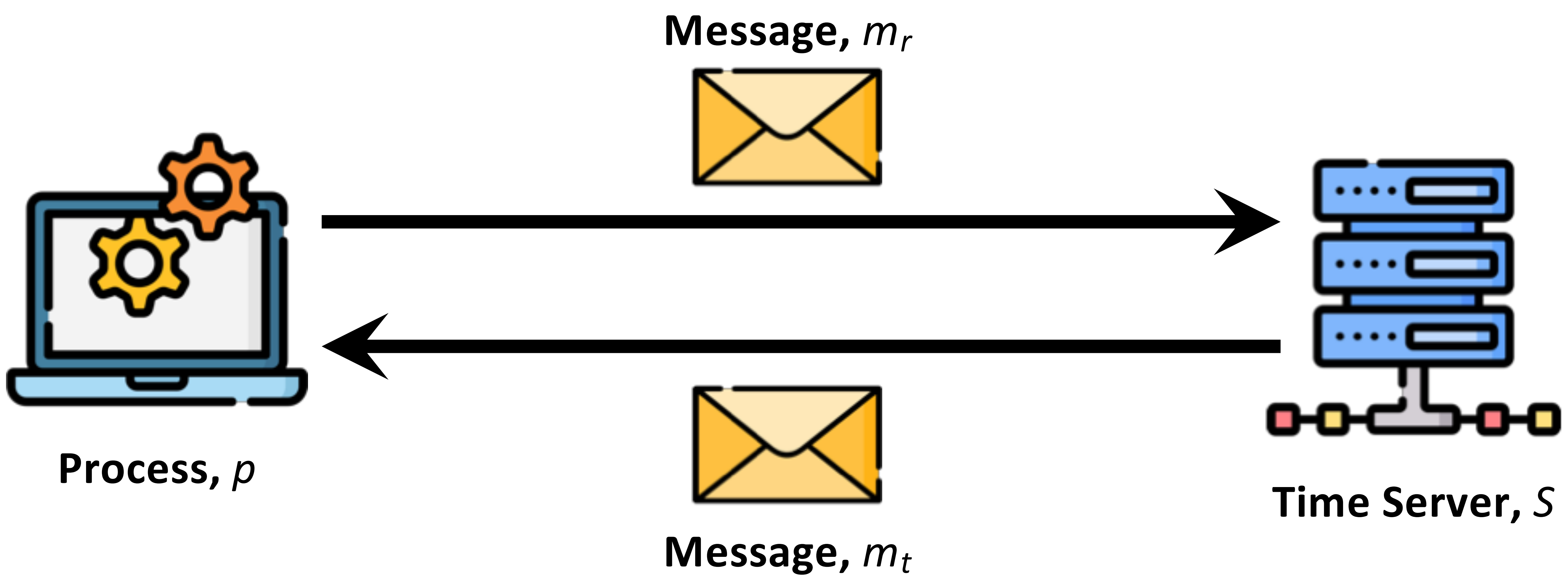
(RTT is the round trip time recorded by p)

Cristian's Time Sync

Setting:

A time server *S* receives signals from a UTC source

Process *p wants to know the time*



Process p sets its clock to *t + RTT/2*

(RTT is the round trip time recorded by p)

Accuracy?

- Say, ***min*** is an estimated minimum one way delay
- What is the possible range of time at S when the process p receives response?

  [t + min, t + RTT– min]

- Width of this range?

  RTT– 2*min

- Accuracy = ?

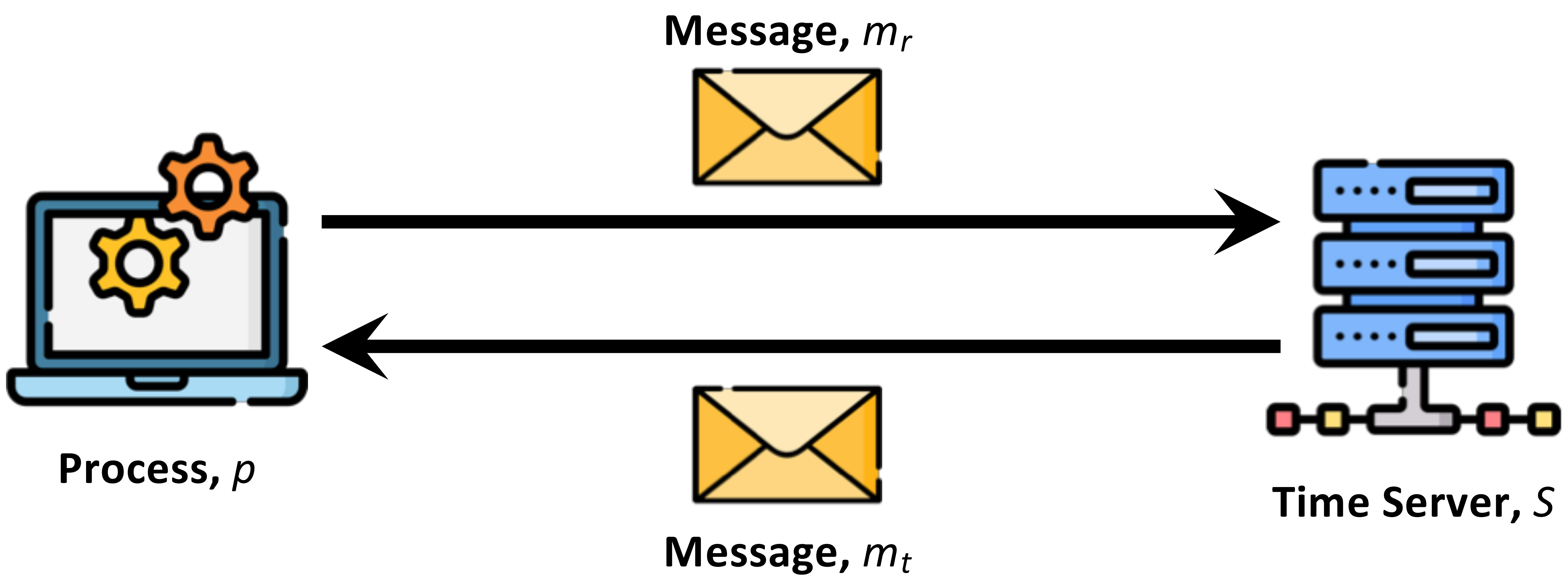
  RTT/2 – min

Cristian's Time Sync

Setting:

A time server *S* receives signals from a UTC source

Process *p wants to know the time*



Process p sets its clock to *t + RTT/2*

(RTT is the round trip time recorded by p)

Accuracy = RTT/2 – min

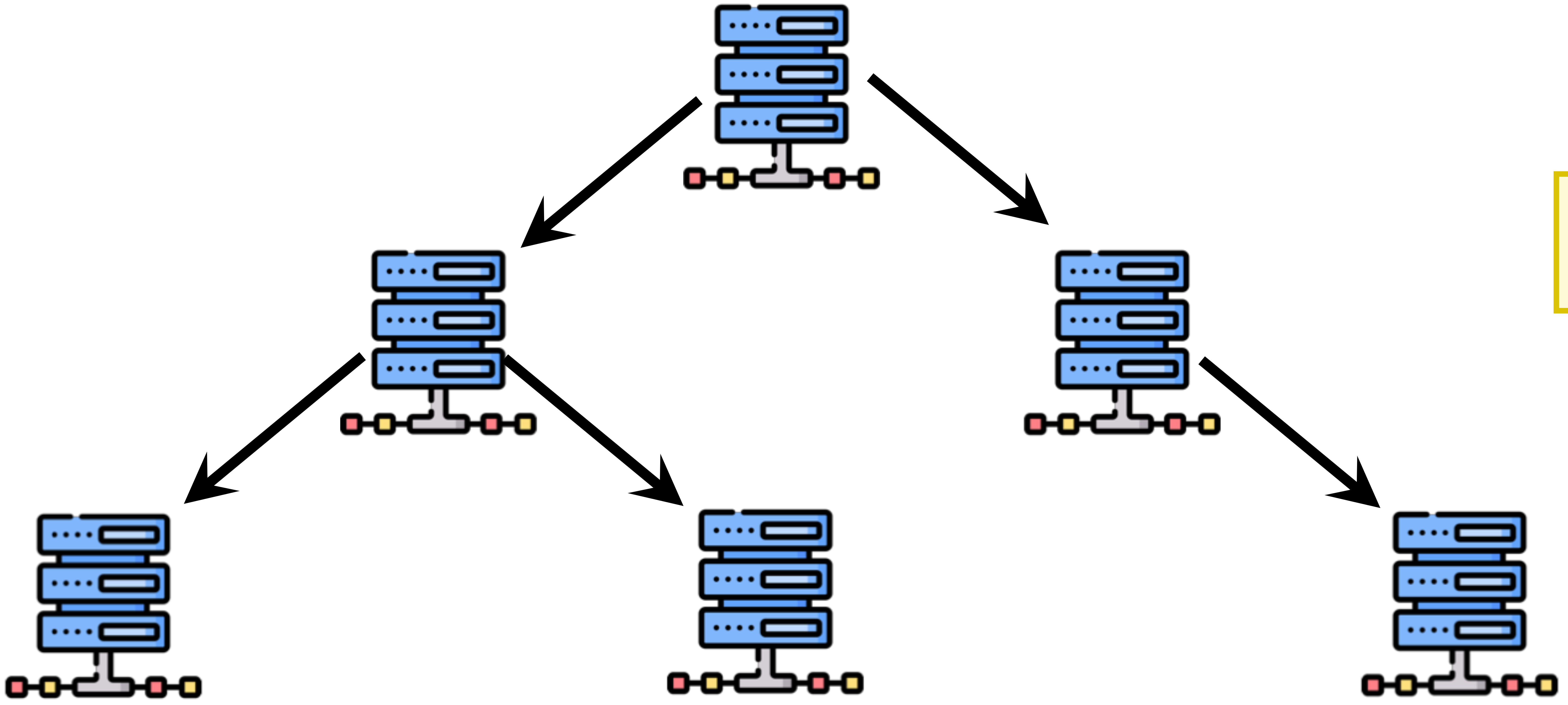
Q: Can you think of any problems with Cristian's Algorithm?

- Works well only for RTT << desired accuracy
- **Key issue: reliance on only one time server**

Network Time Protocol (NTP)

A time service for the Internet - synchronizes clients to UTC

Reliability from multiple, scalable, authenticated time sources



Servers arranged in a hierarchy

Network Time Protocol (NTP)

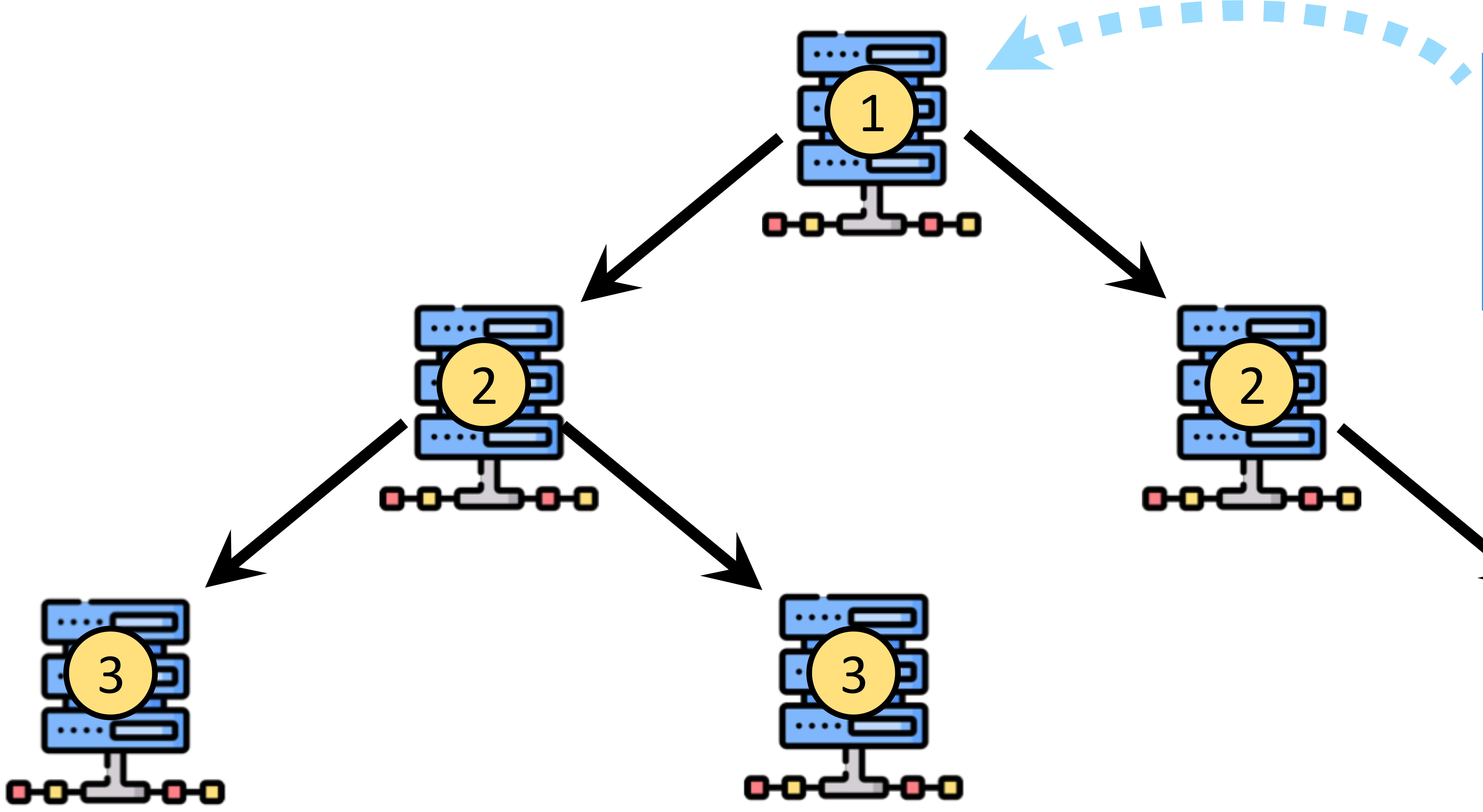
Uses a hierarchy of time servers

- Stratum 1 servers have highly-accurate clocks
  - connected directly to atomic clocks, etc.
- Stratum 2 servers get time from only Stratum 1 and Stratum 2 servers
- Stratum 3 servers get time from Stratum 2
- And so on …

Network Time Protocol (NTP)

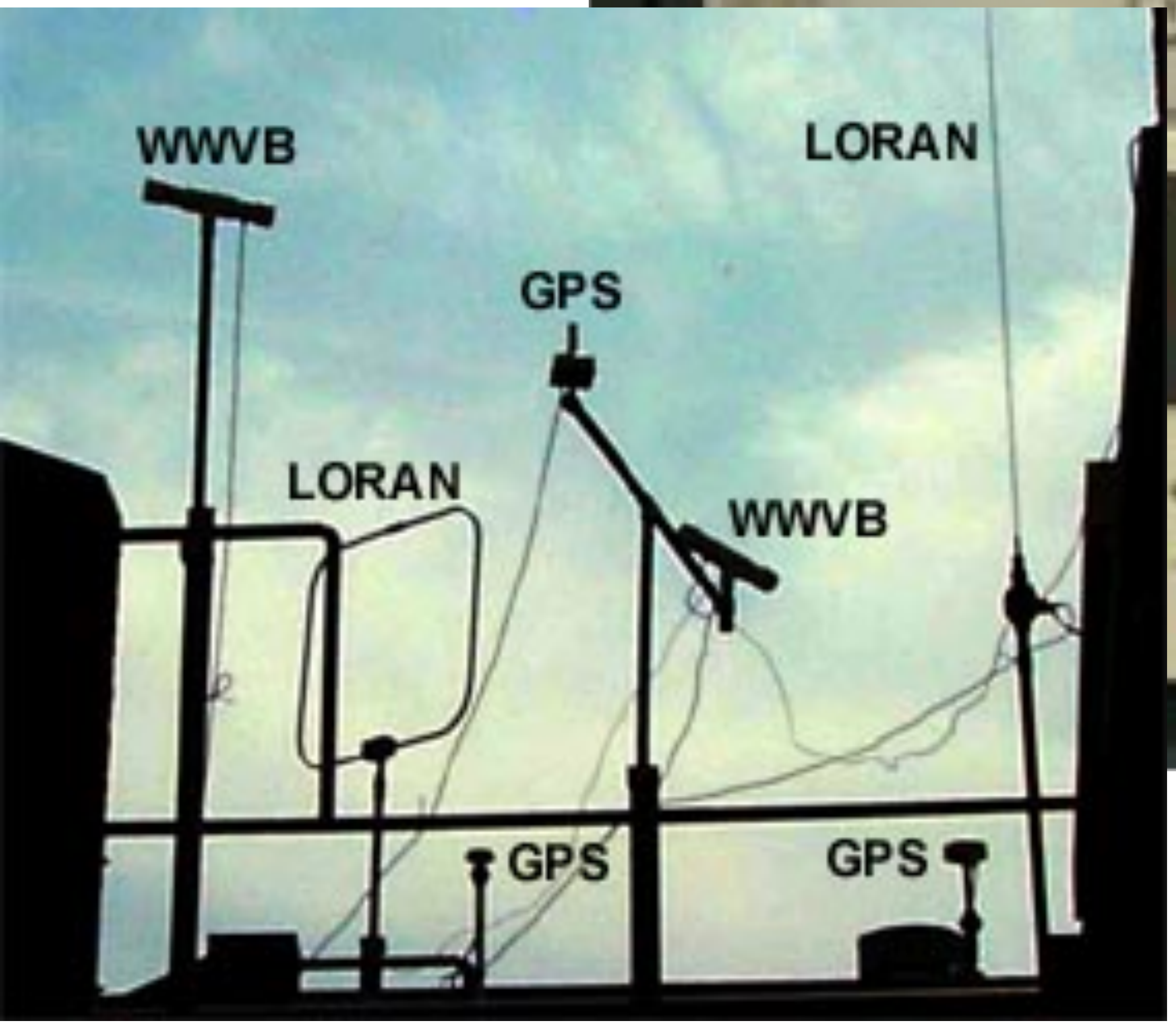
A time service for the Internet - synchronizes clients to UTC

Reliability from multiple, scalable, authenticated time sources



Stratum 1 servers are connected to UTC sources

Udel Master Time Facility (MTF)



Spectracom 8170 WWVB Receiver

Spectracom 8183 GPS Receiver

Spectracom 8170 WWVB Receiver

Spectracom 8183 GPS Receiver

Hewlett Packard 105A Quartz Frequency Standard

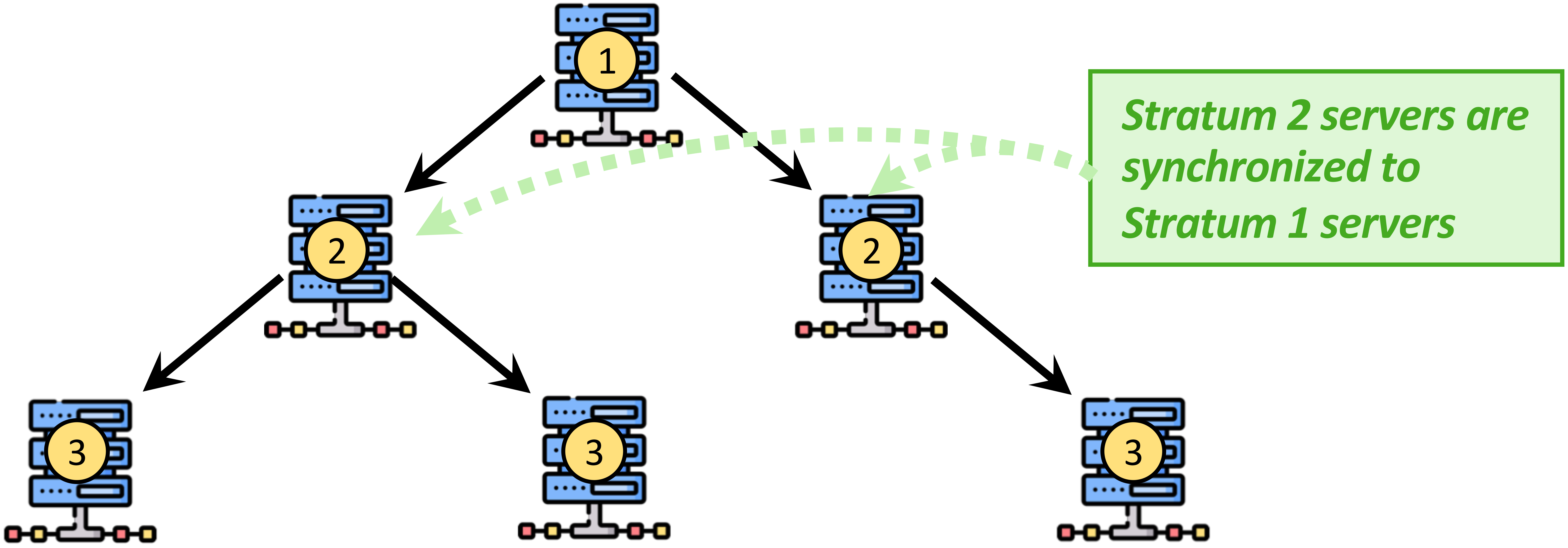
Hewlett Packard 5061A Cesium Beam Frequency Standard

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

Network Time Protocol (NTP)

A time service for the Internet - synchronizes clients to UTC

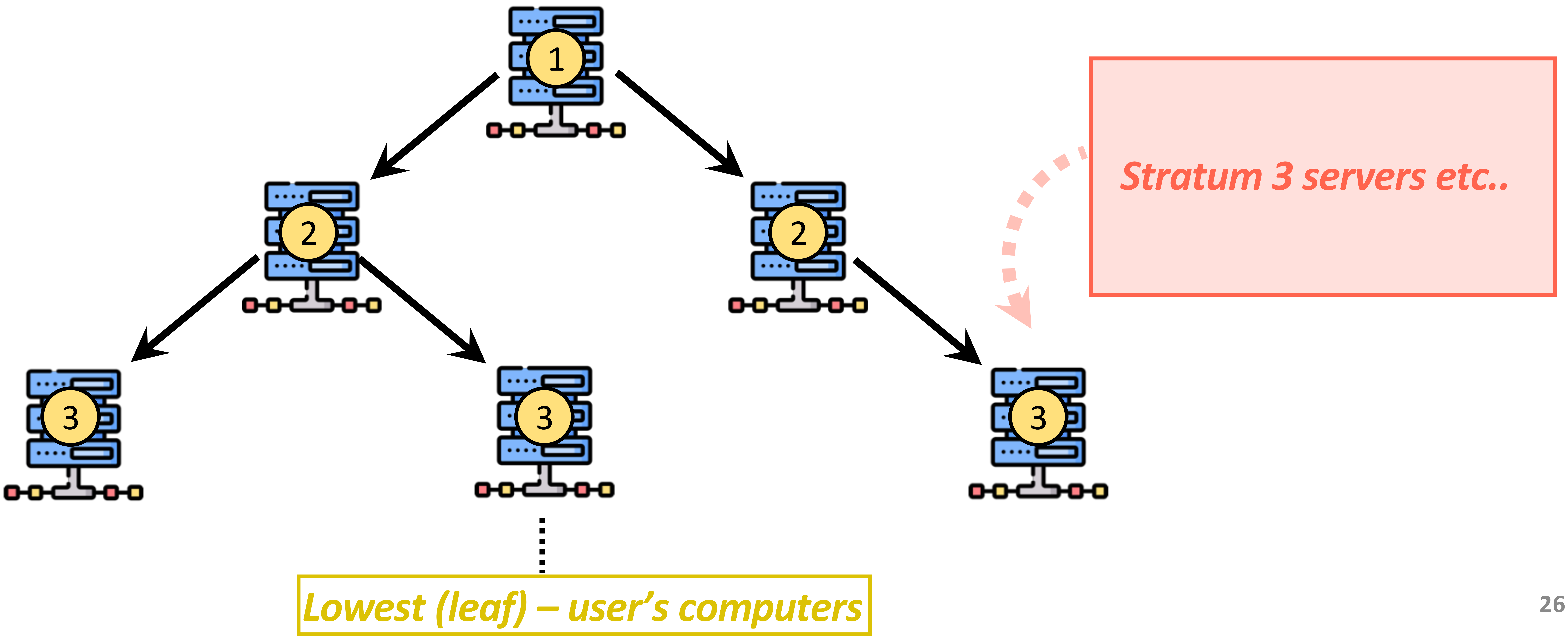
Reliability from multiple, scalable, authenticated time sources



Network Time Protocol (NTP)

A time service for the Internet - synchronizes clients to UTC

Reliability from multiple, scalable, authenticated time sources



Network Time Protocol (NTP)

Uses a hierarchy of time servers

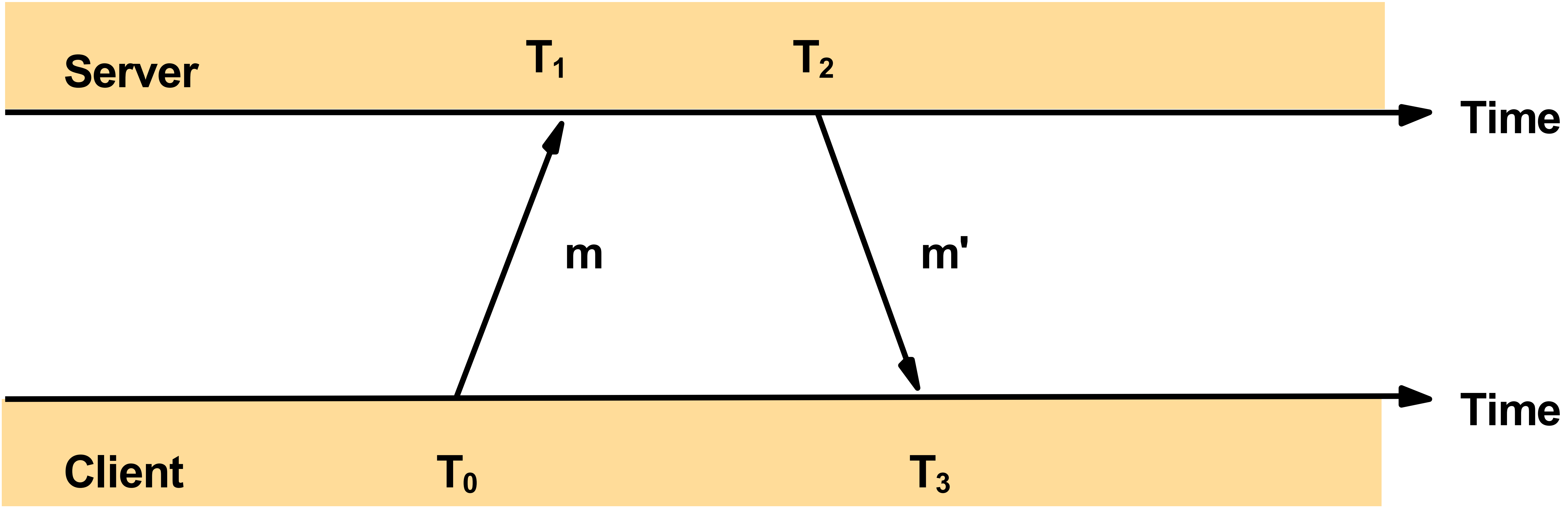
- Stratum 1 servers have highly-accurate clocks
  - connected directly to atomic clocks, etc.
- Stratum 2 servers get time from only Stratum 1 and Stratum 2 servers
- Stratum 3 servers get time from Stratum 2 servers
- So on …

Synchronization similar to Cristian's algorithm

- Modified to use multiple one-way messages instead of immediate round-trip

Accuracy: Local ~1ms, Global ~10ms

NTP 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 T3

(we have T0, T1, T2, T3)

NTP Protocol

Timestamps

- t0 is the client's timestamp of the request packet transmission,
- t1 is the server's timestamp of the request packet reception,
- t2 is the server's timestamp of the response packet transmission and
- t3 is the client's timestamp of the response packet reception.

RTT = wait_time_client – server_proc_time
= (t3-t0) – (t2-t1)

Time adjustment at client: t3 + Offset

Offset = t2 + RTT/2 - t3
= ((t1-t0) + (t2-t3))/2

NTP Protocol

Each server exchanges multiple such messages

Each such exchange give a <rtt, offset> pair

NTP servers filter pairs <rtt_i, offset_i>, estimating reliability from variation, allowing them to select peers

8 measurements $\Rightarrow$ take the one with minimum packet delay

Berkeley Algorithm

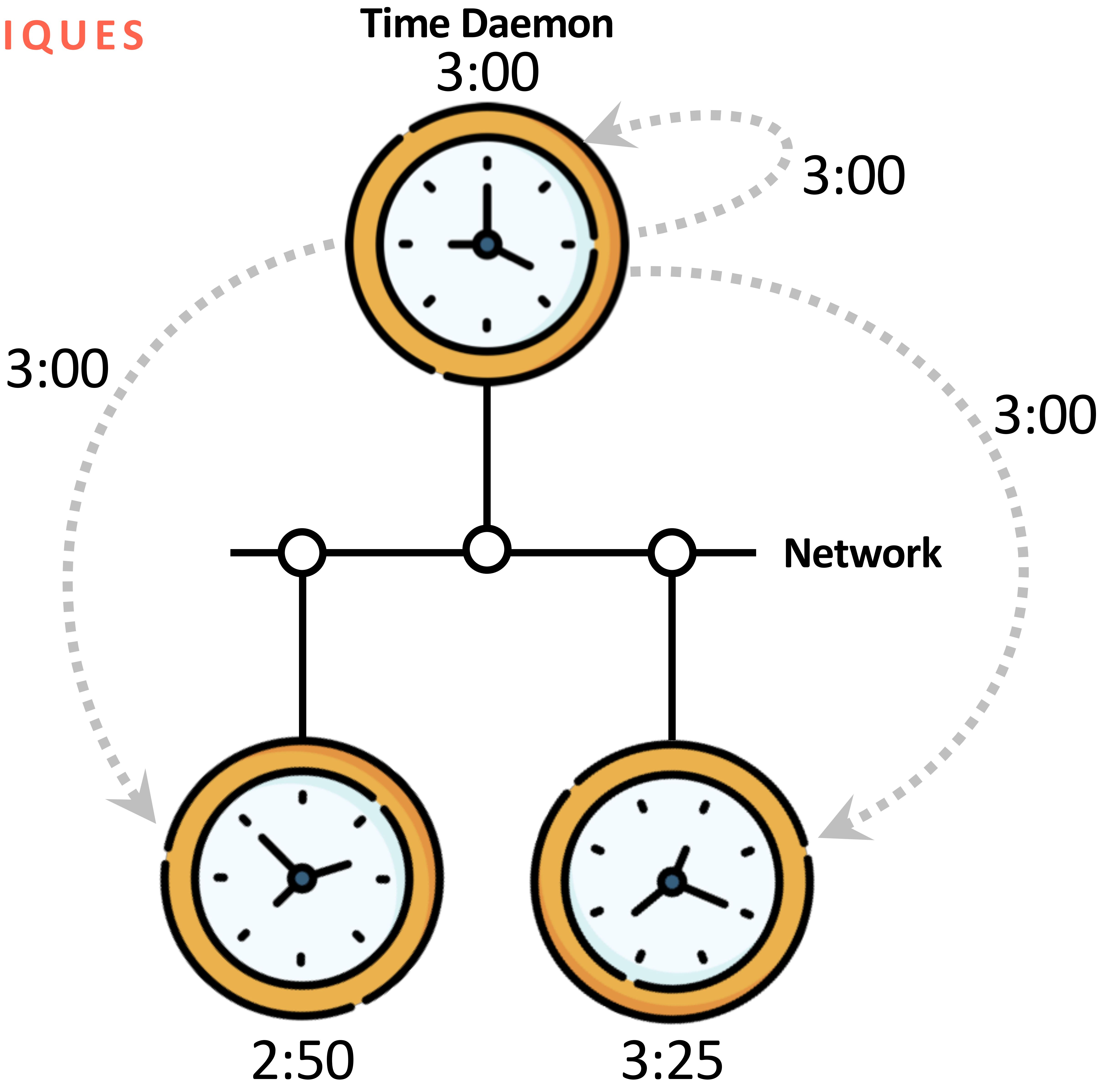
- An algorithm for internal synchronization of a group of servers
- In scenarios where no server has a UTC receiver

- A time server/daemon polls to collect clock values from the others (workers)
  - It's time manually set from time to time

- The daemon uses Christian's algorithm to estimate the workers clock values
- It takes an average (eliminating any above average round trip time or with faulty clocks)
- It sends the required adjustment to the workers (better than sending the time which depends on the round trip time)

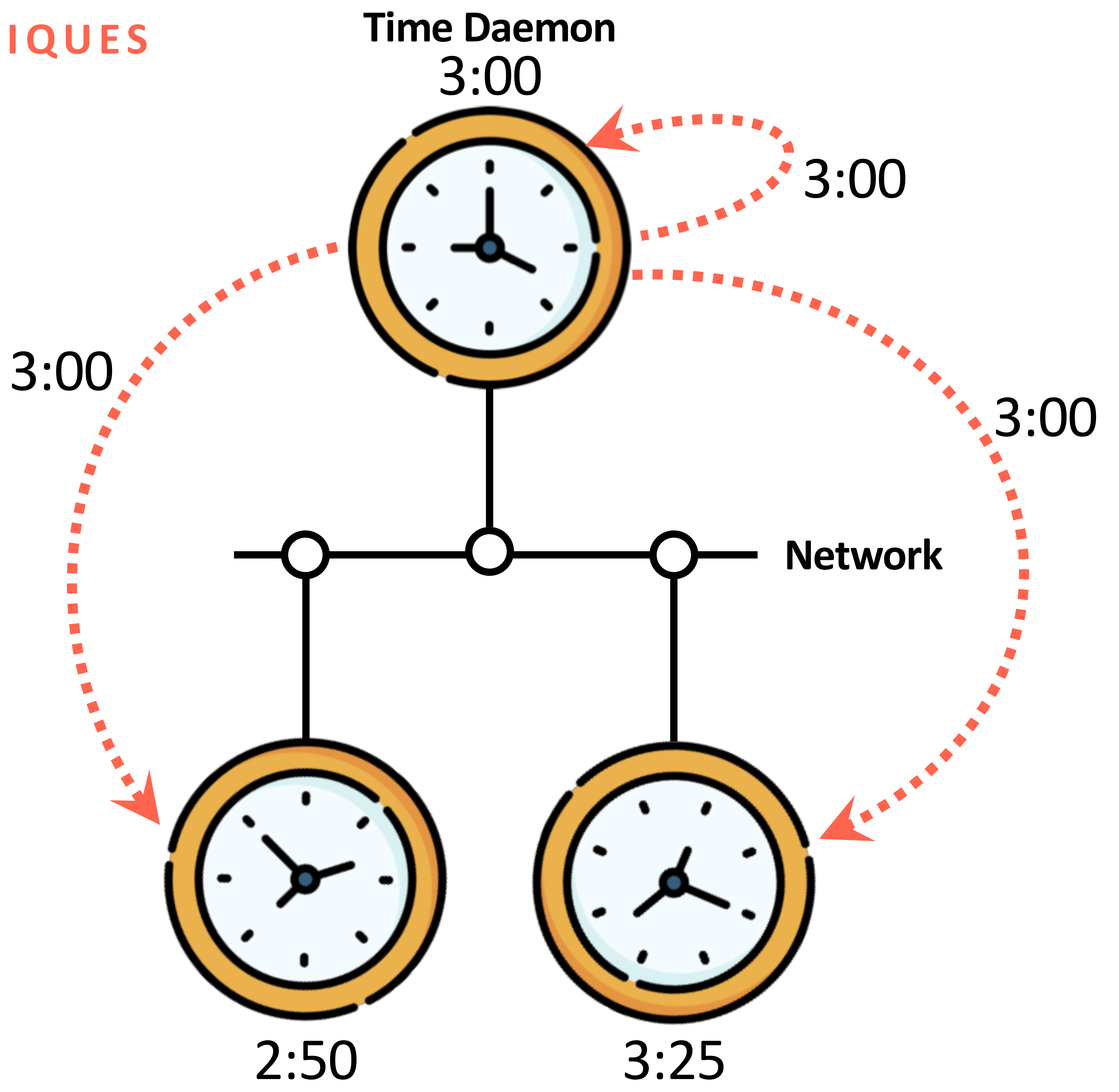
Berkeley Algorithm (1)

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



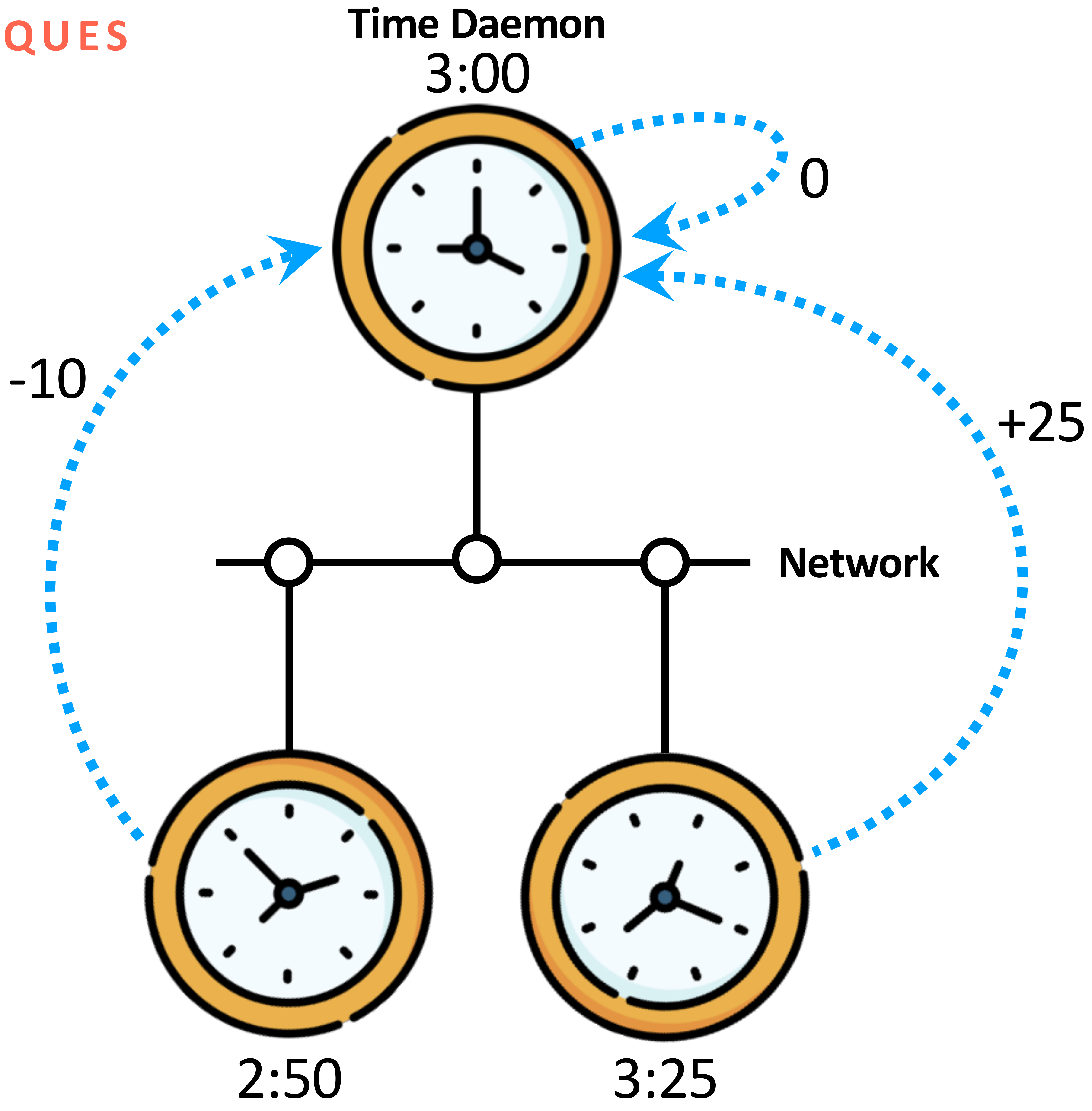
Berkeley Algorithm (1)

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



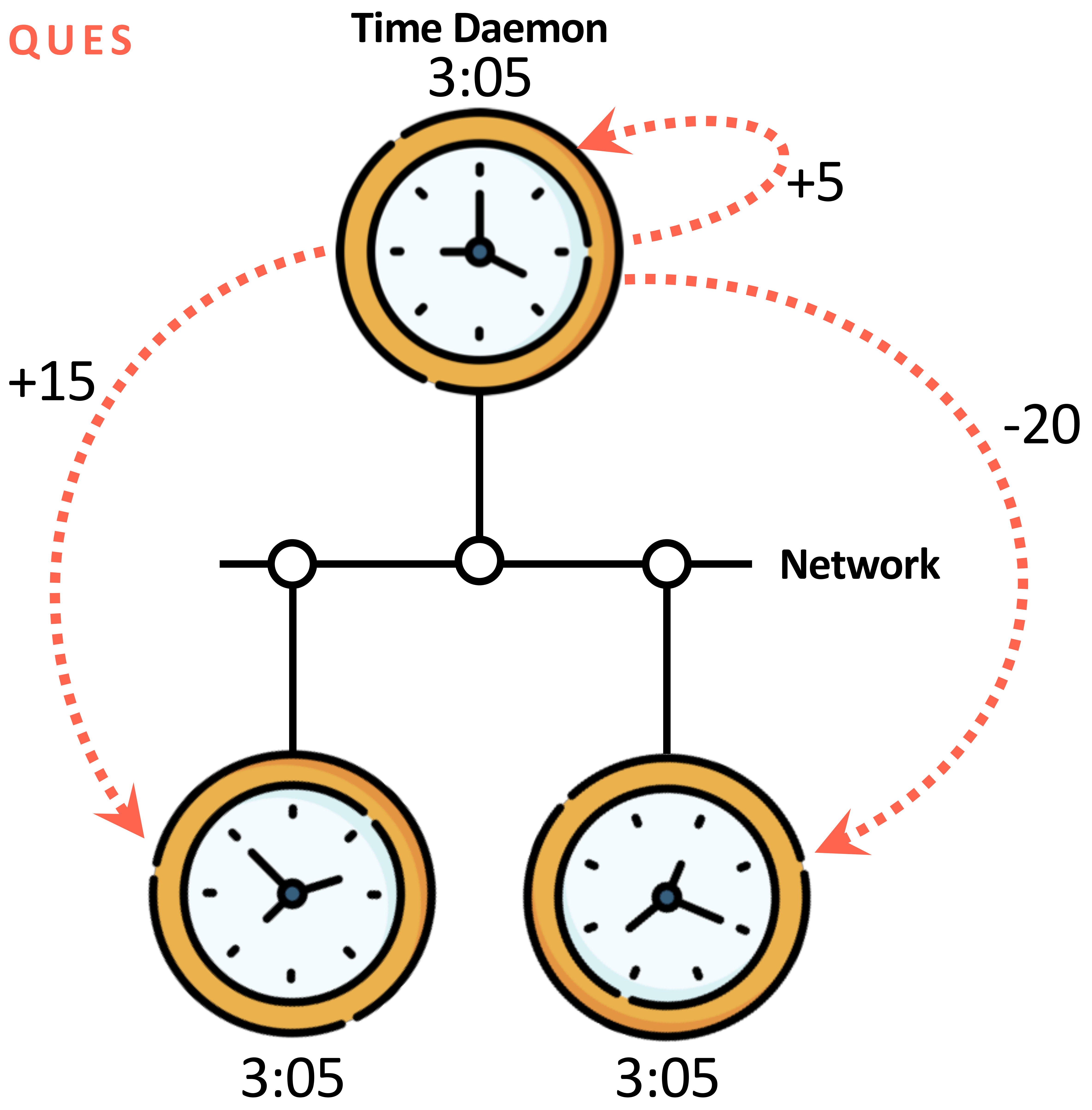
Berkeley Algorithm (2)

The machines answer.



Berkeley Algorithm (3)

The time daemon tells everyone how to adjust their clock.



Berkeley Algorithm

- An algorithm for internal synchronization of a group of servers
- In scenarios where no server has a UTC receiver

- A time server/daemon polls to collect clock values from the others (workers)
  - It's time manually set from time to time

- The daemon uses Christian's algorithm to estimate the workers clock values
- It takes an average (eliminating any above average round trip time or with faulty clocks)
- It sends the required adjustment to the workers (better than sending the time which depends on the round trip time)

- **If daemon fails?**
  - **Can elect a new one to take over (not in bounded time)**

How to Change Time

Can't just change time

- Why not?

Solution?

Change the update rate for the clock

- Changes time in a more gradual fashion
- Prevents inconsistent local timestamps

BUT WAIT,

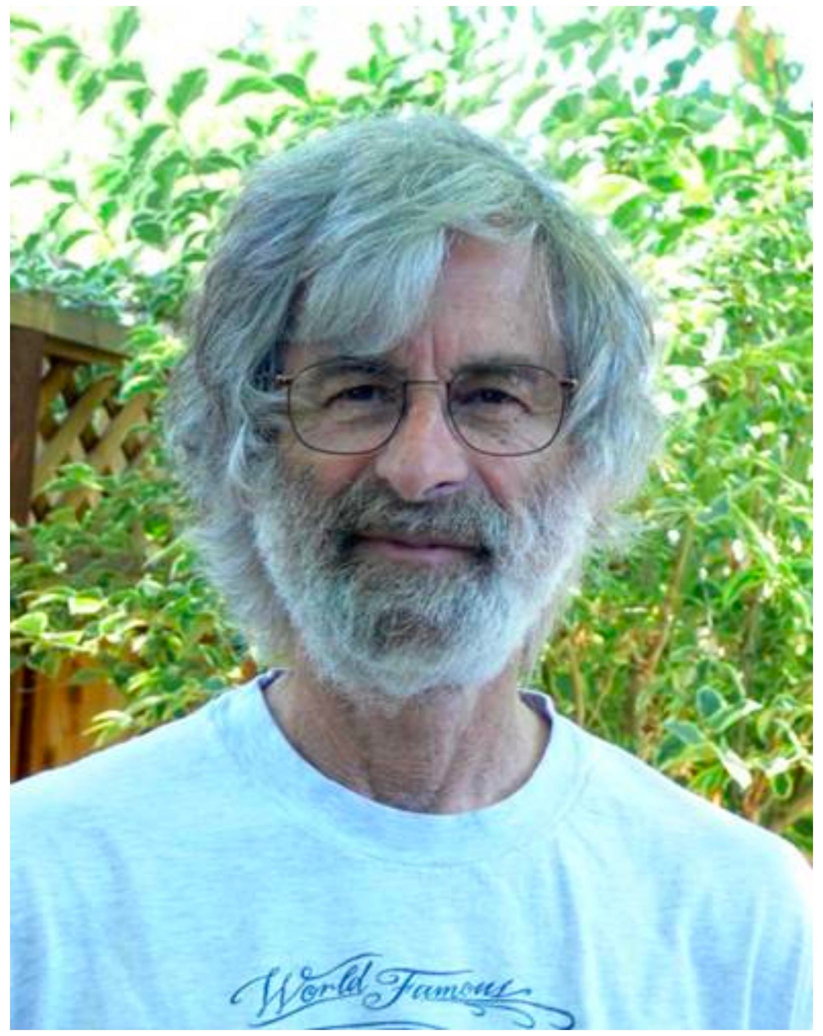
Do we actually need to know the exact time to manage a distributed system?

Agenda

- Need for time Synchronization
- Basic Time Synchronization Techniques
- **Lamport Clocks**
- Vector Clocks
- Time Synchronization in Recent Years

LAMPORT's

Logical Time



Lamport in 1978:

What usually matters is not that all processes agree on exactly what time it is, but rather that they agree on the order in which events occur.

Lamport Clocks

Capture just the "happens before" relationship between events

- Discard the infinitesimal granularity of time
- Corresponds roughly to causality

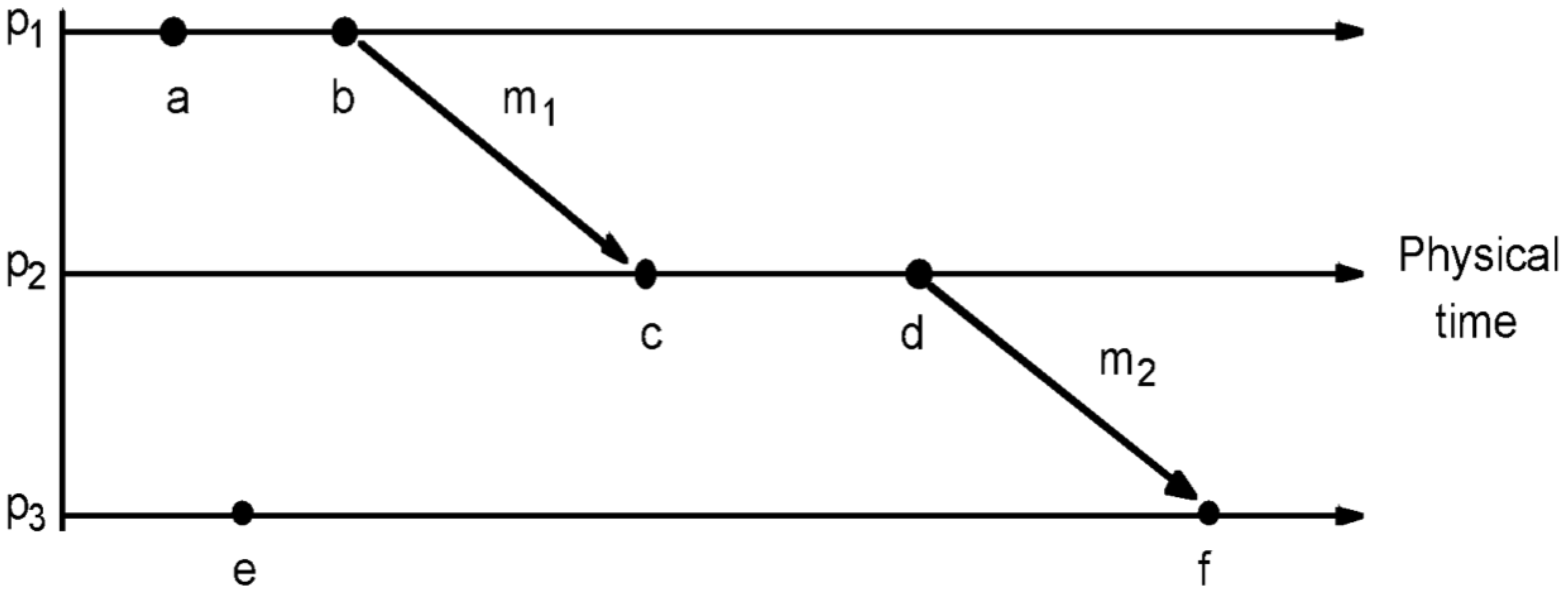
The expression a → b *is read* "event a ***happens before*** event b"

Means: ***All processes agree*** *that first event* a *occurs, then afterward, event* b *occurs.*

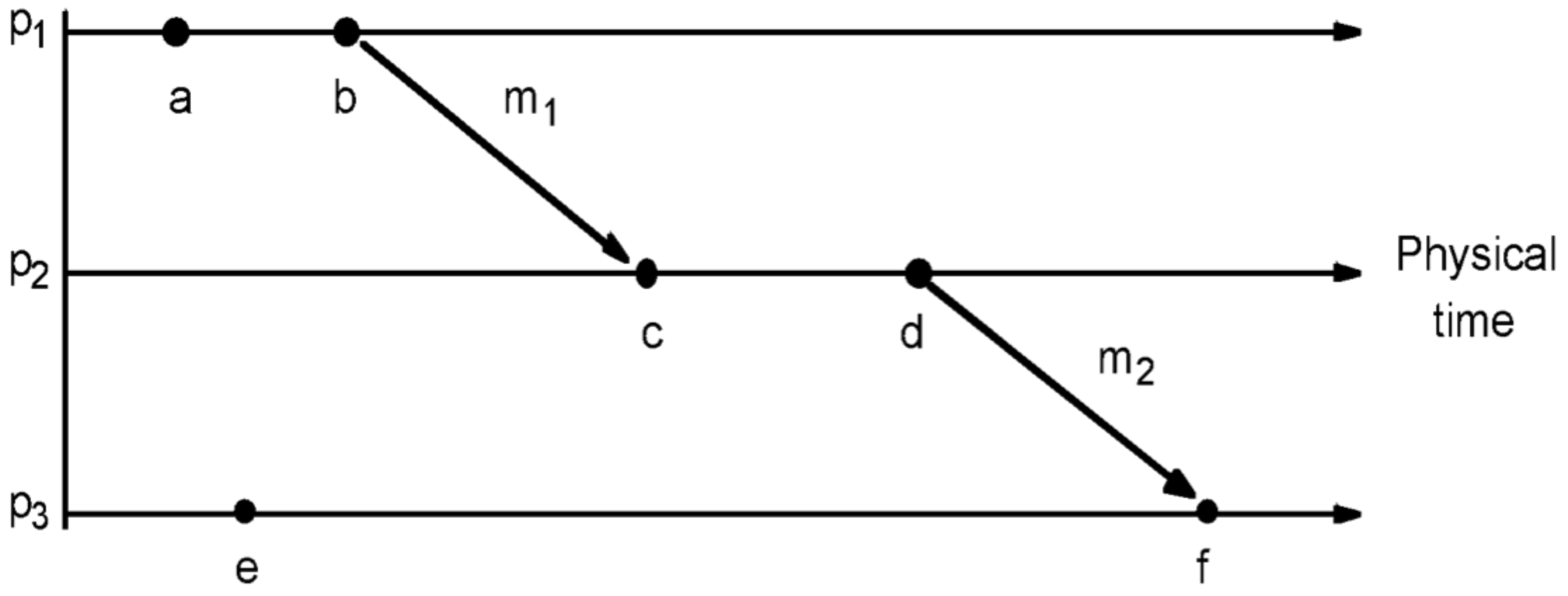
EXAMPLE

Logical time (Lamport 1978)

Events at three processes



Logical time (Lamport 1978)

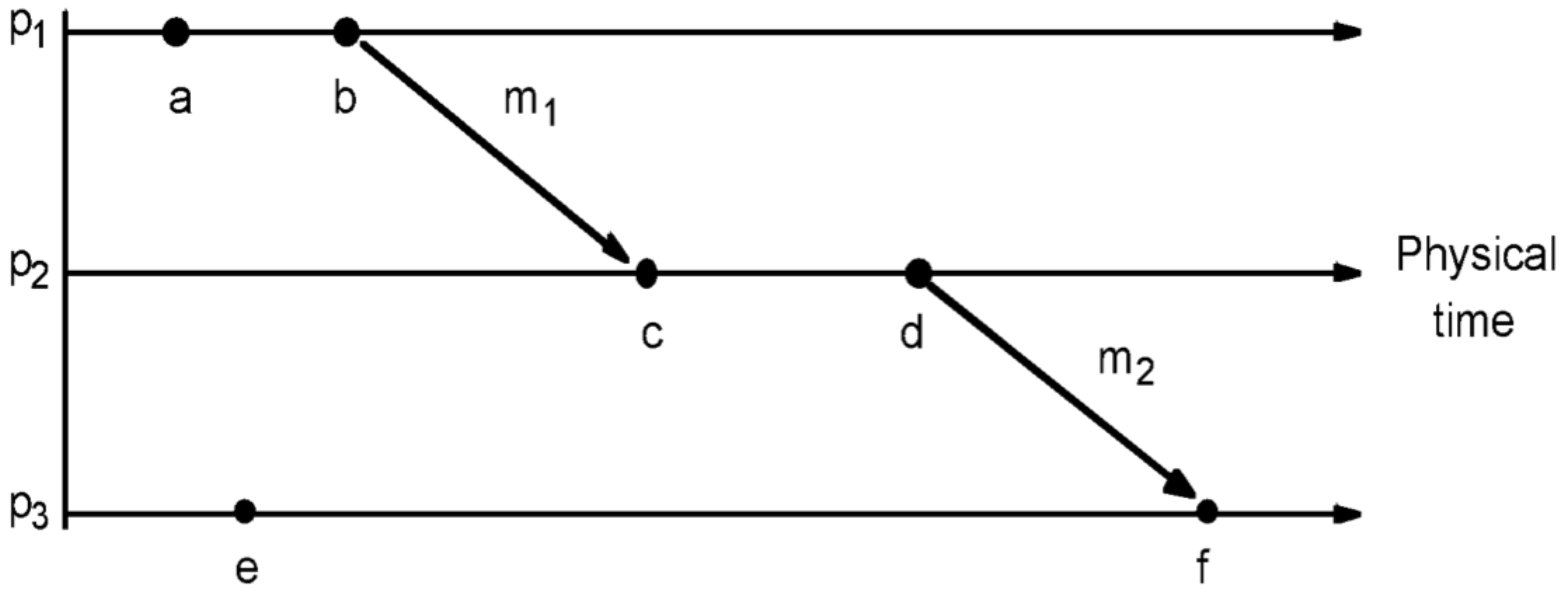


Instead of synchronizing clocks, event ordering can be used

Two scenarios where "happens-before" relation can be directly observed:

1. Two events occurred at same process p_i (i = 1, 2, … N): then they occurred in the order observed by p_i.
2. When a message, m, is sent between two processes: send(m) happens before receive(m).

Logical time (Lamport 1978)



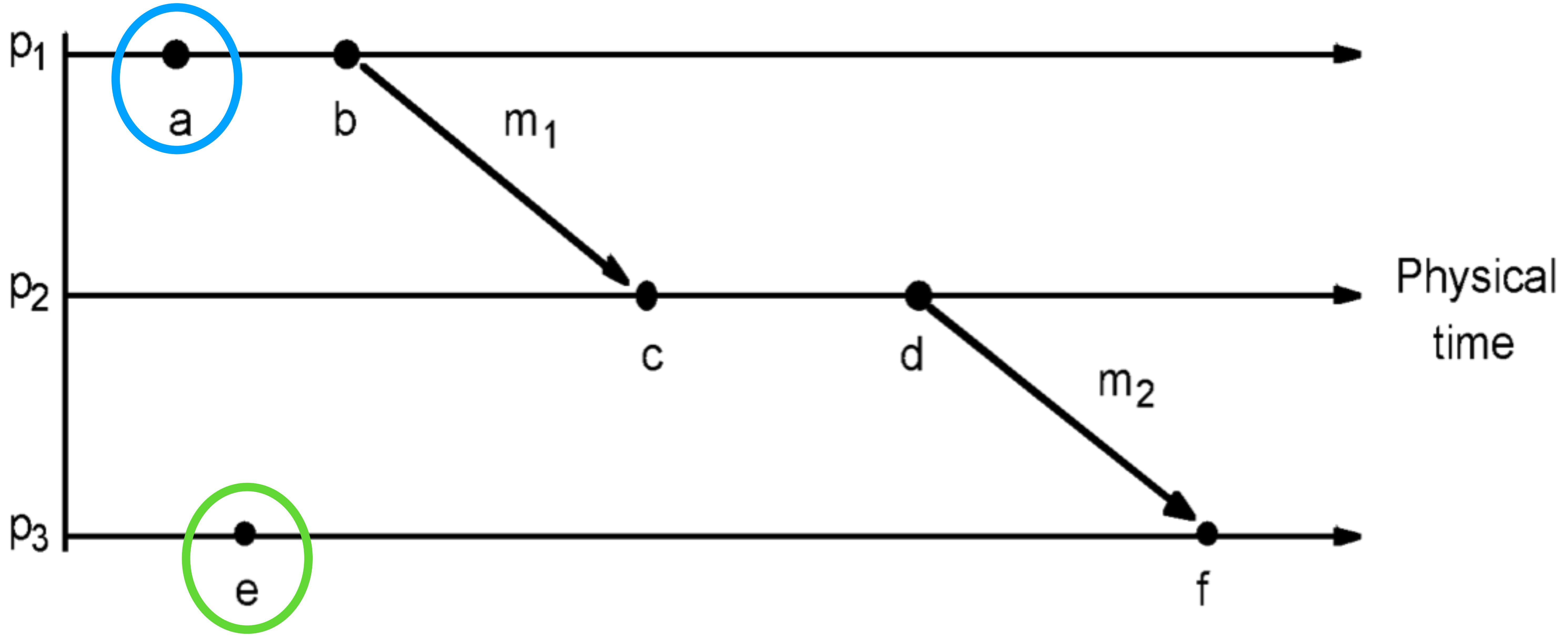
The "happened before" relation is transitive.

$a \rightarrow b$ (at p_1) and $b \rightarrow c$ because of m_1 => $a \rightarrow c$

$c \rightarrow d$ (at p_2) and $d \rightarrow f$ because of m_2

=> $a \rightarrow f$

Logical time (Lamport 1978)



Not all events are related by "happens before" (→)

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 Clocks (1)

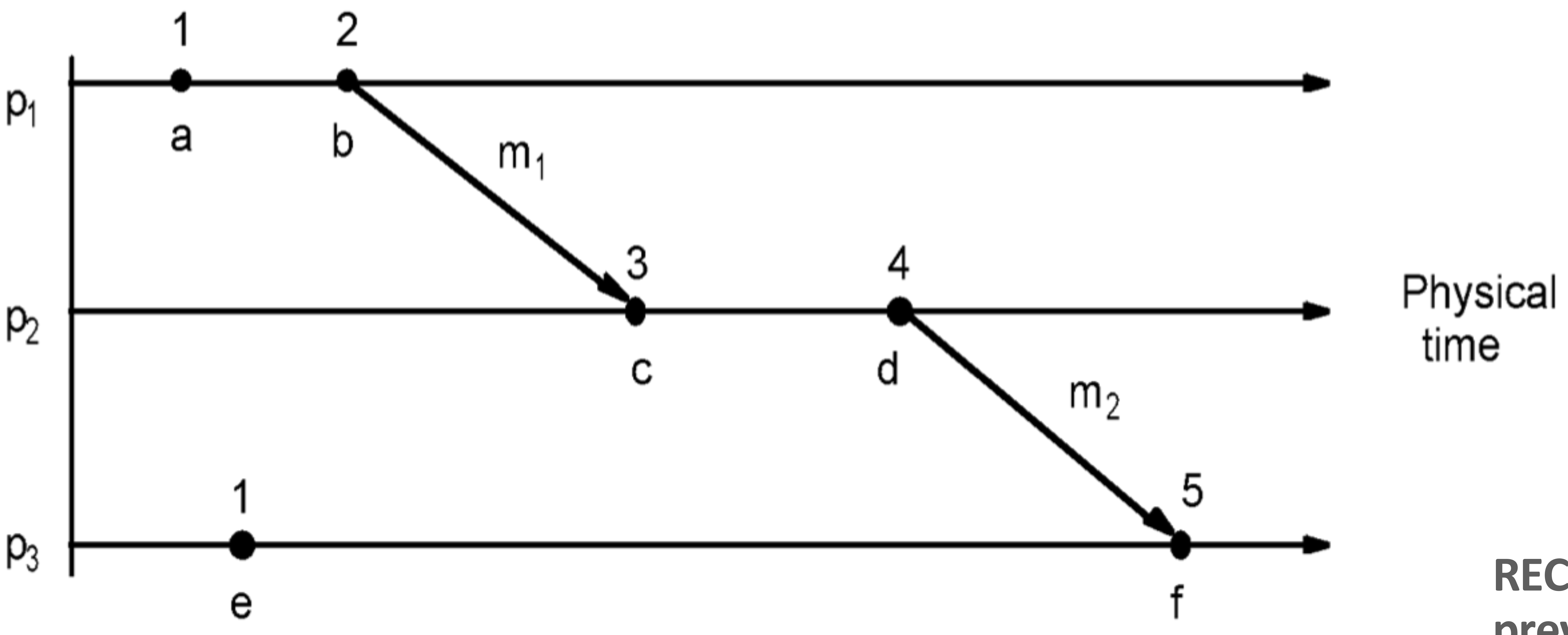
A logical clock is a monotonically increasing software counter

- 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_j 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 initialized to zero,

(The clock values are shown by the numbers immediately after the event.)

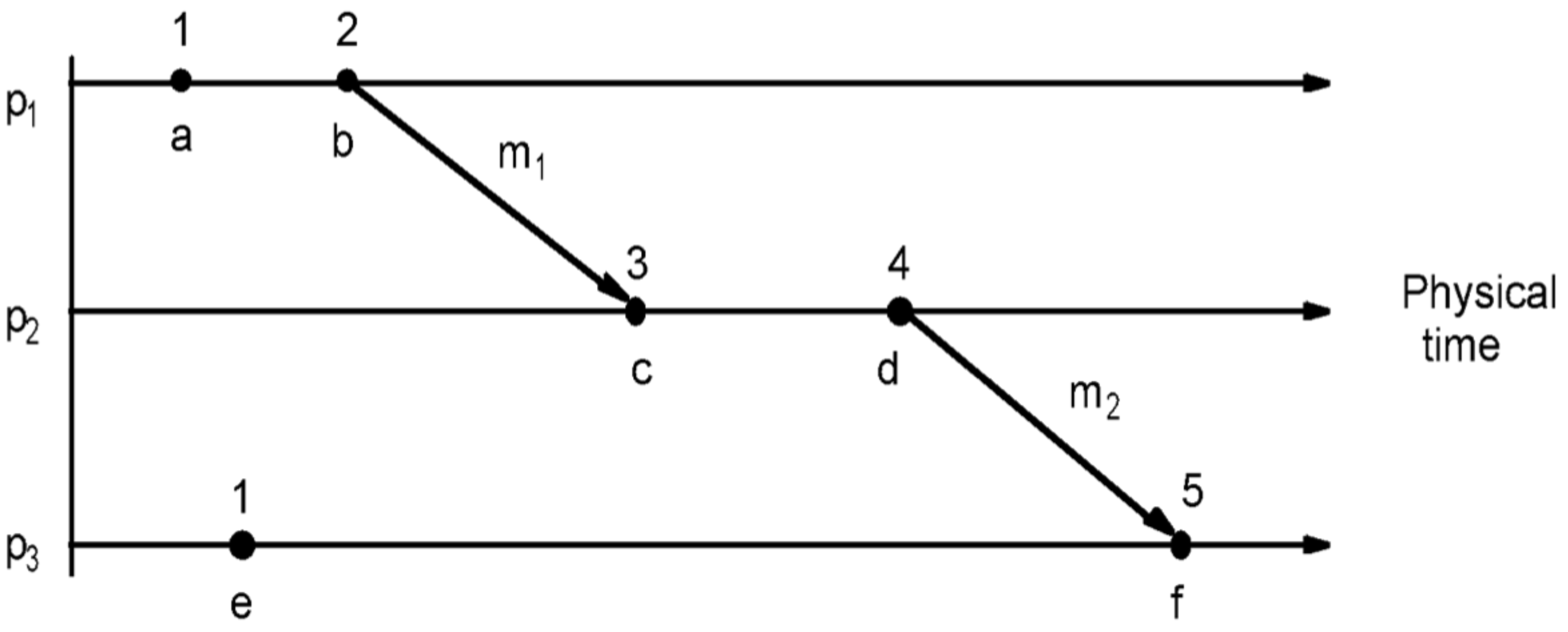
E.g. 1 for a, 2 for b.

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

RECALL Rule 2(b) from previous slide:

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 Clocks (3)

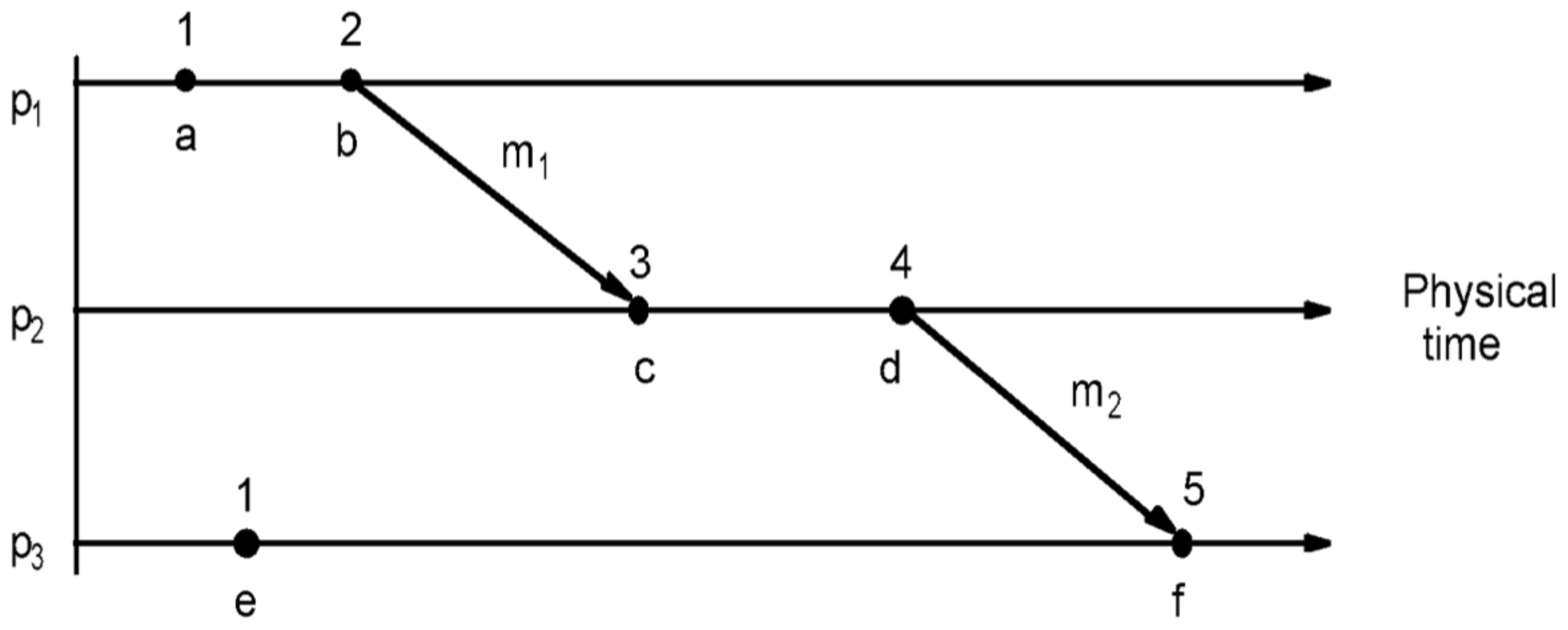


e →e′ **implies** *L(e)<L(e′)*

The converse is not true, that is *L(e)<L(e')* **does not imply** *e →e′*

e.g. *L(b) > L(e) but b || e*

Lamport Clocks (4)



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

Is Lamport's algorithm sufficient?

Use Lamport's algorithm, but break ties using the process ID

Mathematically,

- $L(e) = M * L_i(e) + i$
  - M = maximum number of processes
  - i = process ID

Practice a few examples of Lamport clocks!

Agenda

- Need for time Synchronization
- Basic Time Synchronization Techniques
- Lamport Clocks
- **Vector Clocks**
- Time Synchronization in Recent Years

Vector Clocks

A shortcoming of Lamport logical clocks:

e happened before *e' implies L(e) < L(e')*

But 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'*

Vector clocks!

Label each event by vector $V(e)$ $[c_1, c_2 \ldots, c_n]$

c_i = # events in process *i* that causally precede *e*

Vector Clock Algorithm

Initially, all vectors $[c_1, c_2 \ldots, c_n] = [0,0,\ldots,0]$

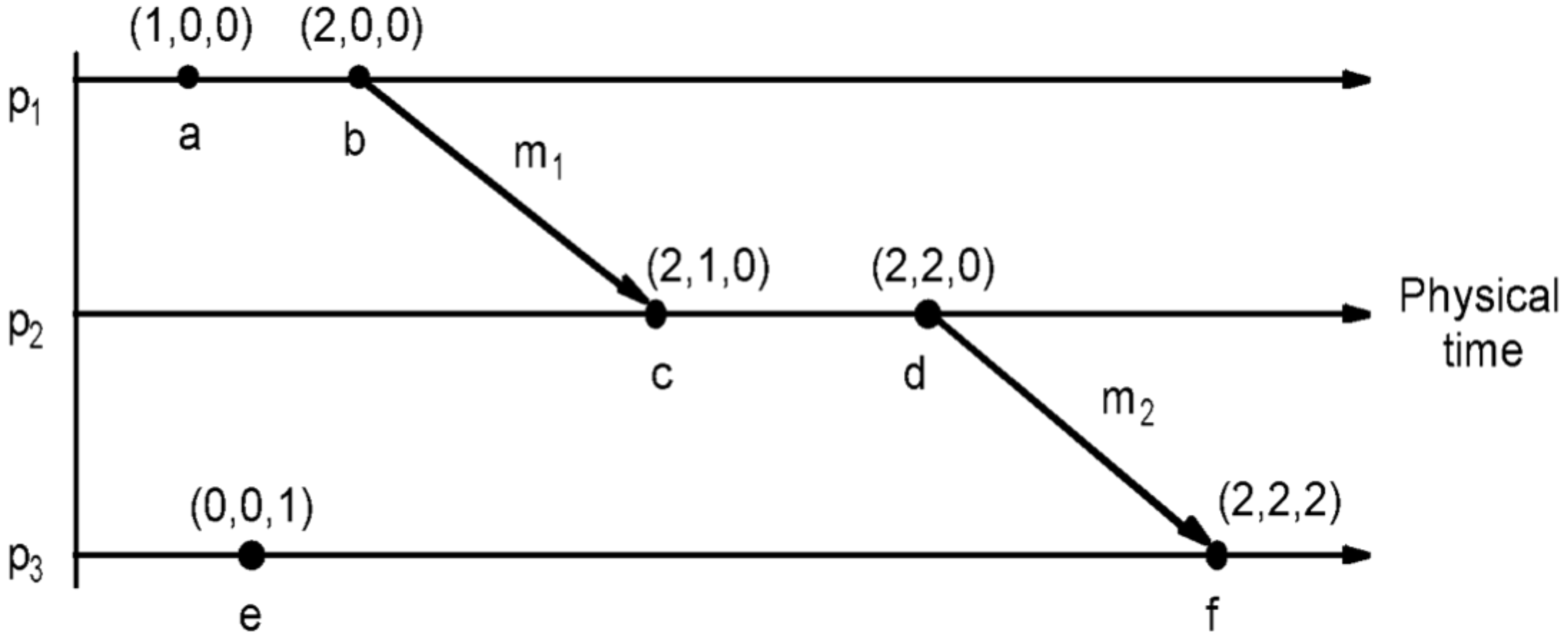
For event on process i**, increment the vector element corresponding to** c_i

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

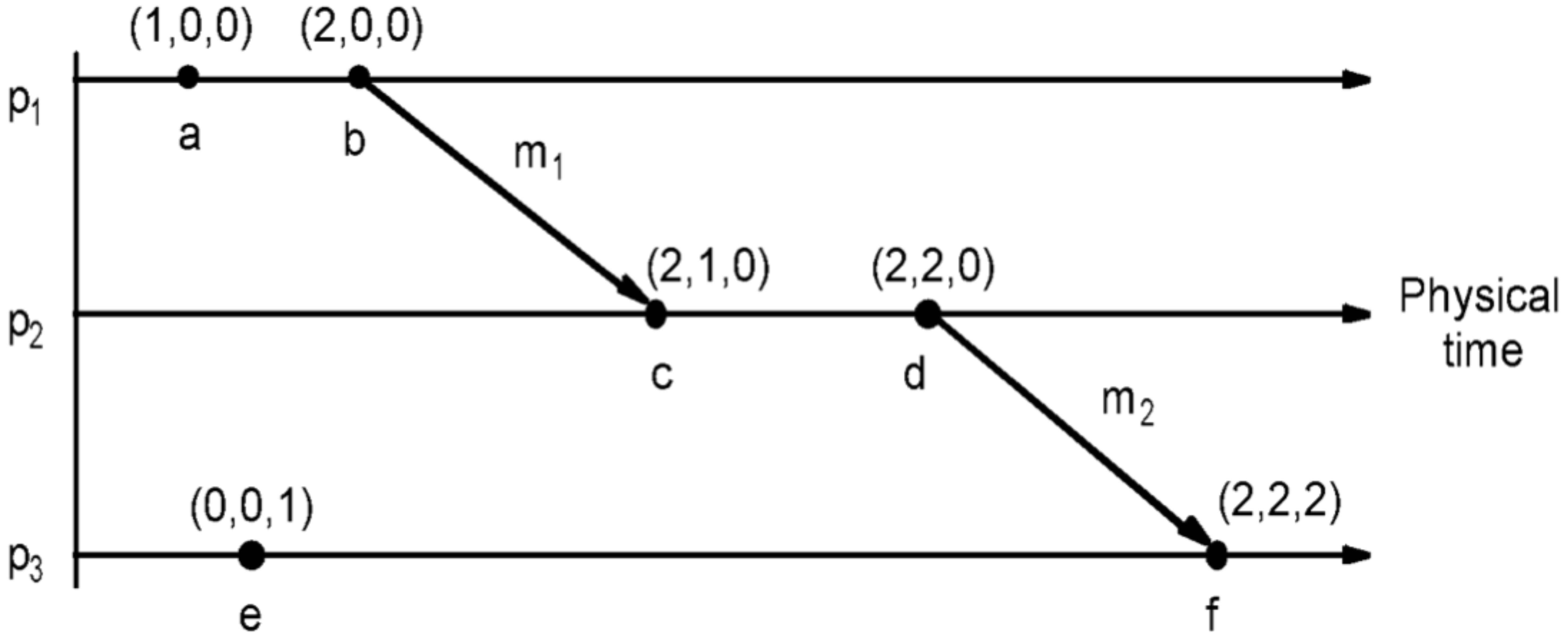


At p_1

- *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)*

Vector Clocks



Meaning of *=, <=, max* etc **for vector timestamps:** *compare elements pairwise*

Properties:

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)*

Clock Sync Important Lessons

Clocks on different systems can (will almost 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 for event ordering

IN SUMMARY

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 between events
- Lamport clocks provide only one-way encoding
- Vector clocks provide exact causality information