SYNCHRONIZATION

CS435 Distributed Systems

Basit Qureshi PhD, FHEA, SMIEEE, MACM
https://www.drbasit.org/

TOPICS

- The Leap second glitch!
- Clocks
- Clock Synchronization
- Ordering of messages in Dist. Systems
- Clock Algorithms

THE LEAP SECOND GLITCH

THE LEAP SECOND GLITCH

- 2012: Reddit outage
- 2012: Mozilla, LinkedIn, Yelp!, Amadeus (airline booking) problems!
- 2017: Cloudflare (servers offline)

- Servers locked up! Non-responsive

- Some airlines processes stopped (Cannot reserve seats, check-ins delays for several hours)

- Chaos: Whats happening!
- Server reboots -> No help!

THE LEAP SECOND GLITCH

- **Culprit: System Clocks**

- Why: Global clocks are sync’ed with Coordinated Universal Time (UTC)
  - Leap seconds are added to account for the slight variations in the Earth's rotation
  - Why: Atomic Time stays in sync with astronomical time
  - Glitch happens when systems encounter difficulties adjusting to the extra second
  - So: System Admins have to manually “add” time to the clocks

WHY CLOCKS ARE IMPORTANT

Distributed systems need to measure time:

- Scheduling: Timeouts, Failure detectors, Retry counters
- Performance: Measurements, statistics, profiling
- Databases and Transactions: Record event occurrence time
- Data with Time to Live (TTL): Cache entries, replicas
- Order of events: Communication between nodes

Two types of clock:

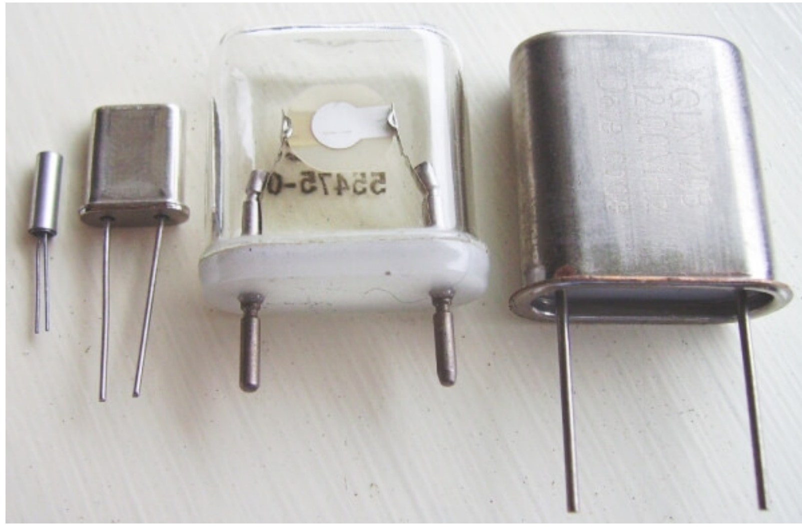
- **Physical clocks:** count number of seconds elapsed
- **Logical clocks:** count events, e.g. messages sent

WHY CLOCKS ARE IMPORTANT

Physical Clocks

- Measure time in seconds
- **Analog** clocks based on mechanical mechanism (Pendulum)
- **Digital** clocks based on vibrating quartz crystal

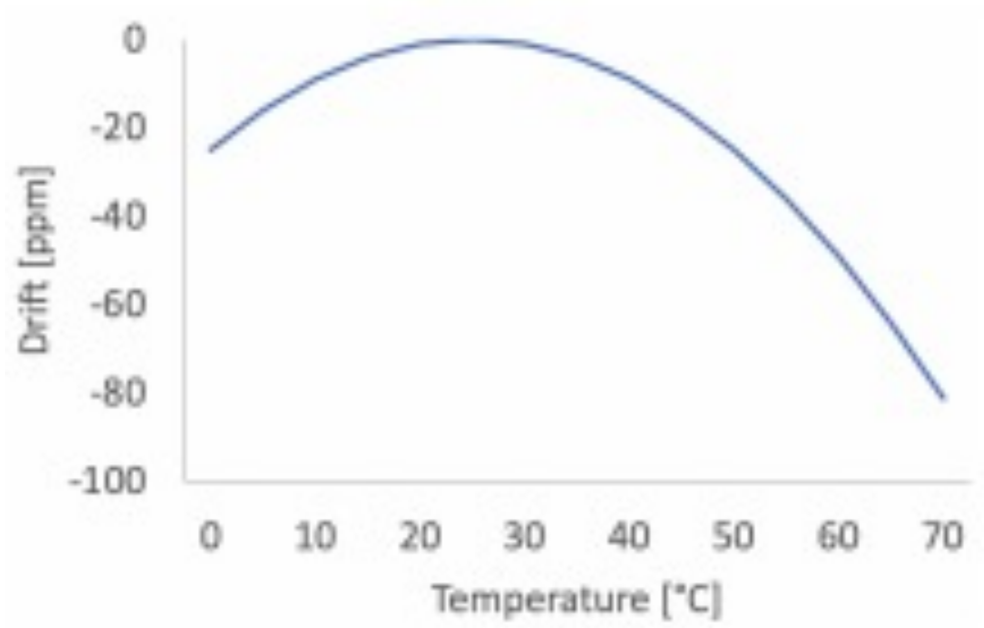
  - Quartz is a hard, crystalline mineral composed of silica (silicon dioxide)
  - Quartz crystals exhibit the piezoelectric effect, which means they can generate an electric charge when mechanical stress is applied to them
  - Oscillator circuit measures resonance frequency when electric voltage is applied to the crystal
  - The current time is based on counting the oscillations of the quartz crystal



WHY CLOCKS ARE IMPORTANT

Quartz Clocks

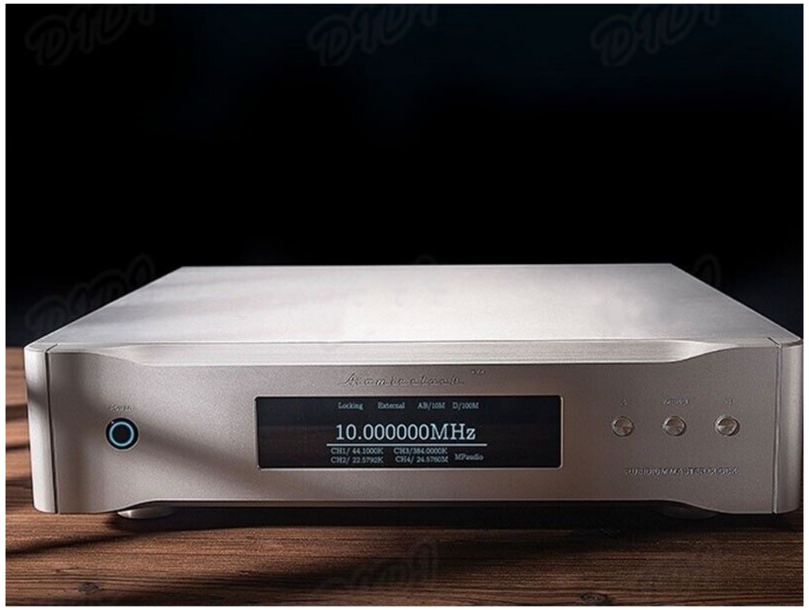
- Cheap but not very accurate!
- One clock runs slightly fast, another slightly slow
- **Drift** measured in parts per million (ppm)
- 1 **ppm** = 1 microsecond/second = 86 ms/day = 32 s/year
- Temperature affects accuracy!



WHY CLOCKS ARE IMPORTANT

Atomic Clocks

- Extraordinarily accurate!
- Uses the vibrations of atoms to measure time (cesium-133, or rubidium-87)
- Caesium-133 has a resonance (“hyperfine transition”) at ≈ 9 GHz
- 1 second = 9,192,631,770 periods of that signal
- Accuracy ≈ 1 in 10^{-14} (1 second in 3 million years)



Price: 2,999 USD

WHY CLOCKS ARE IMPORTANT

GPS based time

- GPS Satellite based systems (Galileo, GLONASS)
  - Operates with 24 satellites distributed in three orbital planes.
  - Each GLONASS satellite continuously broadcasts signals [satellite's current position and the current time]
  - GPS navigation device or a smartphone with GNSS capability, receives signals from multiple GLONASS satellites
  - Receiver can calculate its own position on Earth through a process called tri-lateration
  - Combines results from 3 satellites for improved accuracy
  - **Problem**: the speed of rotation of the planet is not constant; it fluctuates due to the effects of tides, earthquakes, glacier melting, and some unexplained factors.

Glonass-K
2011
Glonass-M
2003
Glonass-K2
2023
Глонасс
1982
Glonass-K2
(modification)

WHY CLOCKS ARE IMPORTANT

Coordinated Universal Time (UTC)

- Based on atomic time BUT
  - Needs periodic corrections due to variations in earths rotation.
  - **International Atomic Time** (TAI): 1 day is 24 × 60 × 60 × 9,192,631,770 periods of caesium-133's resonant frequency
  - **Problem**: speed of Earth's rotation is not constant
    Compromise: UTC is TAI with corrections to account for Earth rotation
  - Solution: Add a leap second

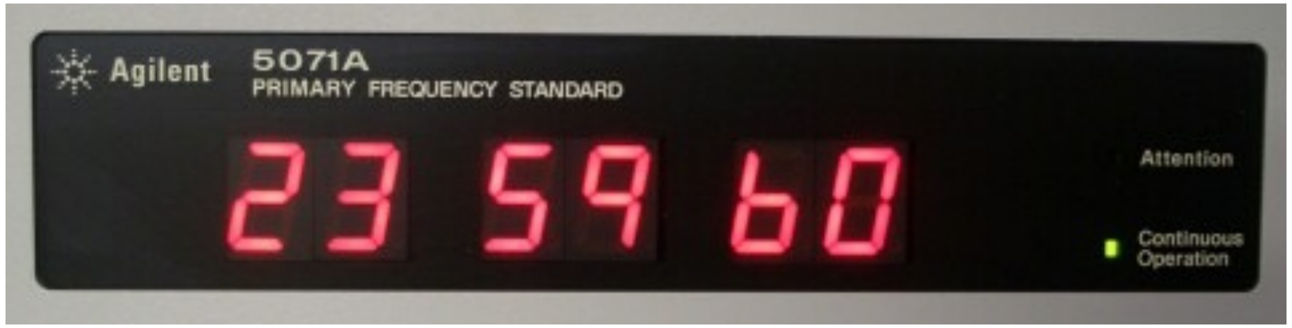
WHY CLOCKS ARE IMPORTANT

Leap second

- Every Six months (June 30 and December 31)
  - Skip One second (23:59:58 -> 00:00:00)
  - Usual (23:59:59 -> 00:00:00)
  - Add One second (23:59:59 -> 23:59:60)

  - This is announced several months beforehand

http://leapsecond.com/notes/leap-watch.htm



WHY CLOCKS ARE IMPORTANT

How Computers see time

- Unix time
  - Number of seconds since 1 January 1970, 00:00:00 UTC (epoch). “Leap seconds not added”
- ISO 8601
  - YY:MM:DD:HH:MM:SS+Offset
  - Eg. 2024-02-29T09:50:17+03:00
- Conversion
  - Gregorian calendar: 365 days in a year, except leap years

    (year % 4 == 0 && (year % 100 != 0 || year % 400 == 0))

WHY CLOCKS ARE IMPORTANT

How Computers see time

- Java's `System.currentTimeMillis()` is like Unix time, but uses milliseconds rather than seconds

```
long T1 = System.currentTimeMillis();
DoSomeErrand();
long T2 = System.currentTimeMillis();
System.out.print(T2-T1);
```

- Was the leap second counted??
  - Unix timestamps, and POSIX standard ignore leap seconds!
  - Difference of a few seconds is not significant.
- However:
  - Dist. Systems rely on time-stamps;
  - A millisecond can cause errors

WHY CLOCKS ARE IMPORTANT

Poor handling of the leap second on 30 June 2012 is what caused the simultaneous failures of many services on that day.

Due to a bug in the Linux kernel, the leap second had a high probability of **triggering** a **livelock** condition when running a multithreaded process

Even a reboot did not fix the problem, but setting the system clock reset the bad state in the kernel.

Fix!

- **Leap smearing**: gradually distribute the adjustment over a longer period of time rather than adding the extra second all at once.

CLOCK SYNCHRONIZATION

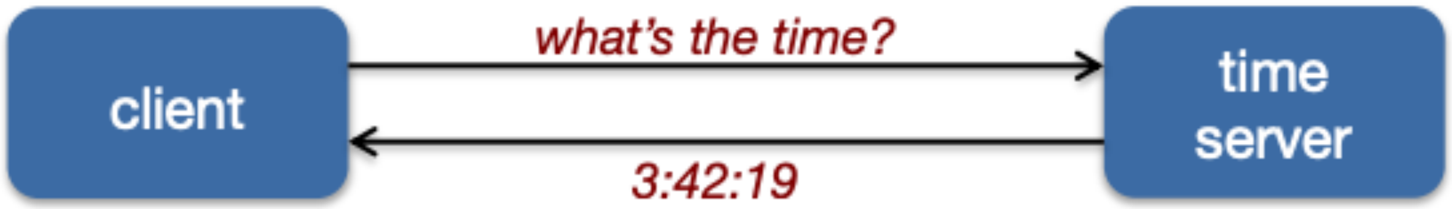
```
//so for Java programmers: The BAD
    long T1 = System.currentTimeMillis();
    DoSomeErrand(); // NTP Client may update time
    long T2 = System.currentTimeMillis();
    System.out.print(T2-T1);
//elapsed time may be negative!


//The GOOD
    long T1 = System.nanoTime();
    DoSomeErrand();
    long T2 = System. nanoTime();
    System.out.print(T2-T1);
//elapsed time is always >=0
```

CLOCK SYNCHRONIZATION

CLOCK SYNCHRONIZATION

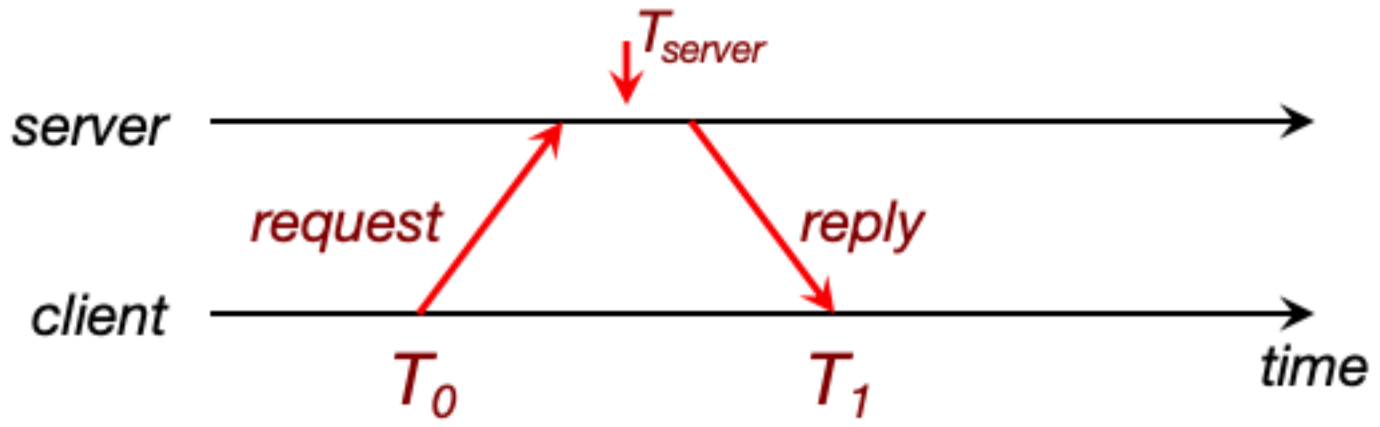
- Simplest synchronization technique
  - Send a Server, request to obtain the time
  - Set the time to the returned value



Problem: What about network latency??

CLOCK SYNCHRONIZATION

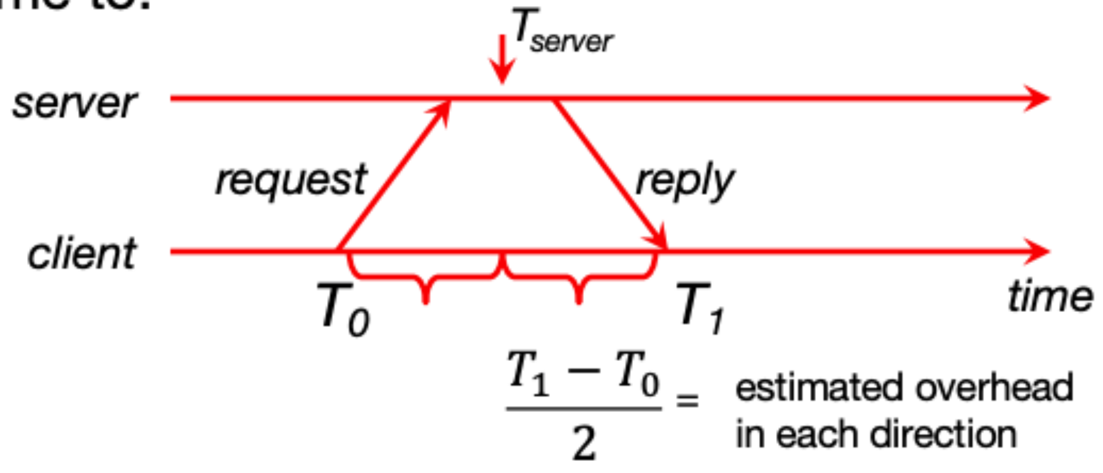
- **Christians method**
  - Compensate for delays
    - Note times:
      - request sent: T0
      - reply received: T1
    - Assume network delays are symmetric



CLOCK SYNCHRONIZATION

- **Christians method**

Client sets time to:


T_{server}
server
request
reply
client
T_0
T_1
time
$\frac{T_1 - T_0}{2}$ = estimated overhead in each direction

$$\boxed{T_{new} = T_{server} + \frac{T_1 - T_0}{2}}$$

What about errors and accuracy??

CLOCK SYNCHRONIZATION

- Christians method example:
  - Client sent request at 5:08:15.**100** (T_0)
  - Client receives response at 5:08:15.**900** (T_1)
  - Response contains **5:09:25.300** (T_{server})

Elapsed time is T_1 -T_0 = 5:08:15.900 - 5:08:15.100 = **800 ms**

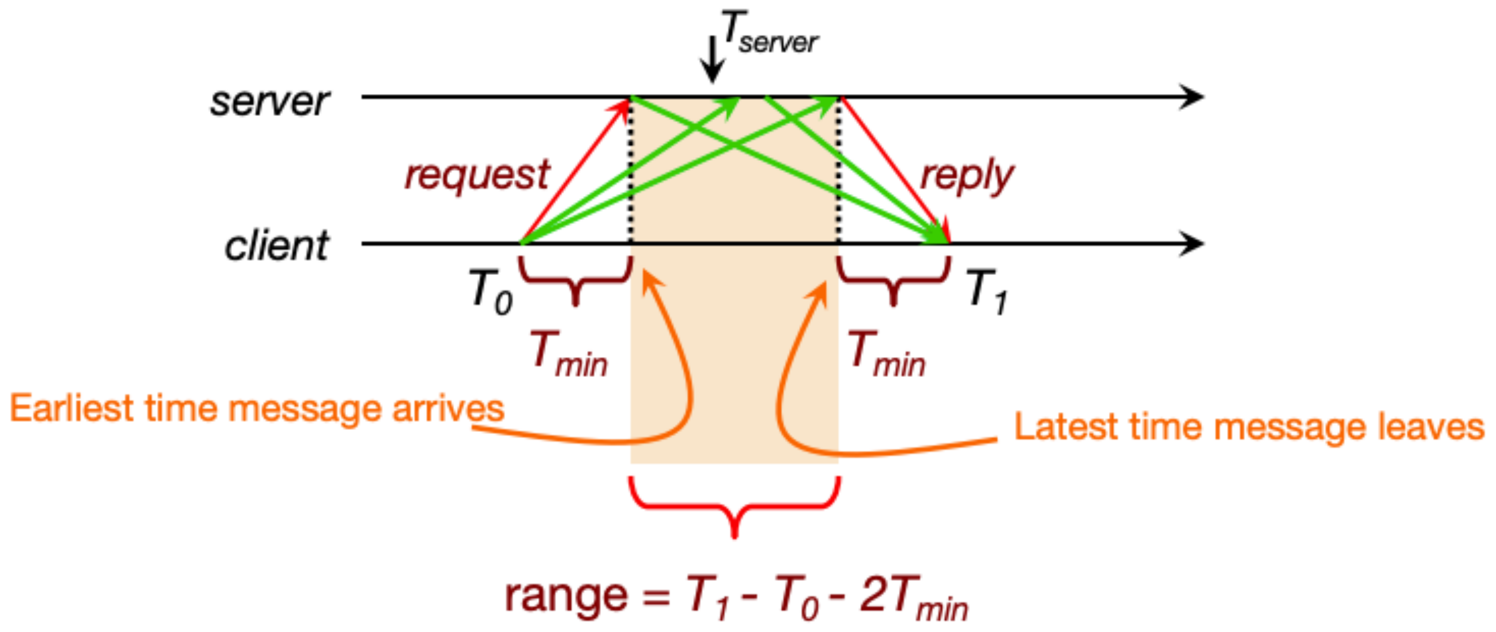
Best guess timestamp was generated: 800ms / 2 = 400 ms ago

Set time to T_{server}+ elapsed time = **5:09:25.300** + **0.400** =
5:09:25.700

Note: 1000ms = 1 second

CLOCK SYNCHRONIZATION

- **Christians method**



T_{server}
server
request
reply
client
T_0
T_1
T_{min}
T_{min}
Earliest time message arrives
Latest time message leaves
range = $T_1 - T_0 - 2T_{min}$

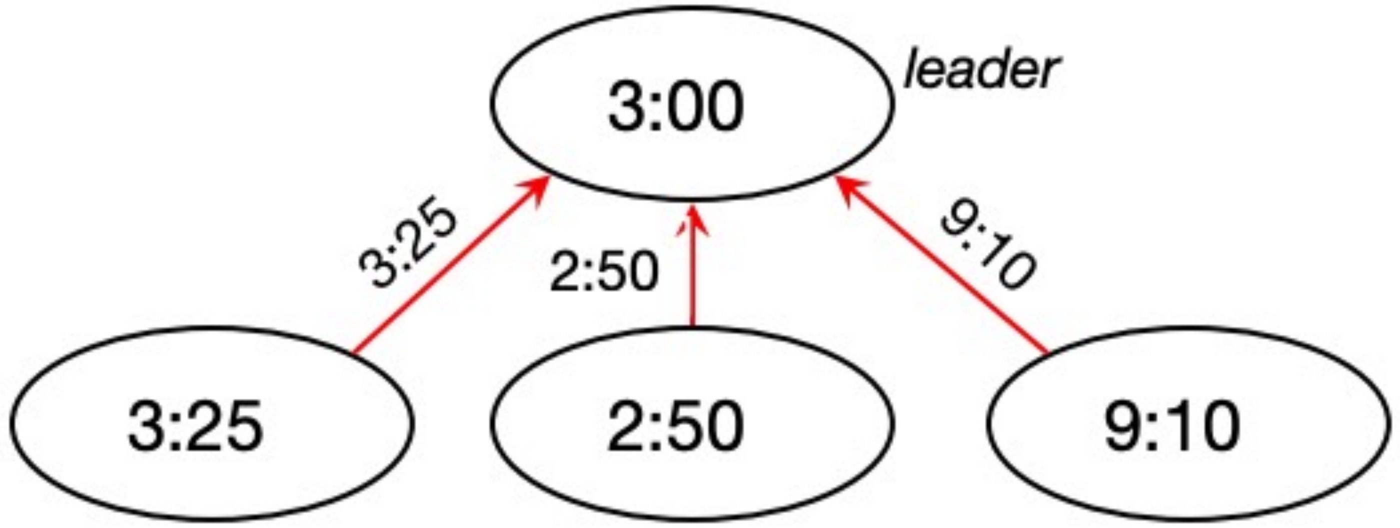
$$\boxed{\text{accuracy of result} = \pm \frac{T_1 - T_0}{2} - T_{min}}$$

CLOCK SYNCHRONIZATION

- **Berkeley Algorithm [Gusella & Zatti, 1989]**
  - Designed for intranets
  - Assumes no machine has an accurate time source
  - Obtains time from participating computers
  - Synchronizes all clocks to a **fault-tolerant average**
    - Select the largest set of time values that don't differ from each other by some quantity
    - Avoids averaging values of malfunctioning clocks or clocks that drifted too far

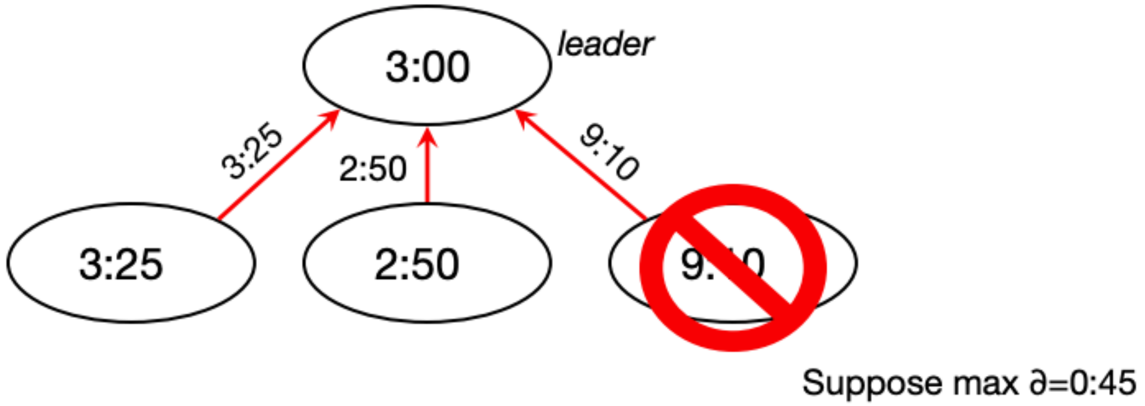
CLOCK SYNCHRONIZATION

- **Berkeley Algorithm [Gusella & Zatti, 1989]**
  - Example:
  - 1. Request timestamps from all followers



CLOCK SYNCHRONIZATION

- **Berkeley Algorithm [Gusella & Zatti, 1989]**
  - Example:
  - 2. Compute Fault-tolerant average:

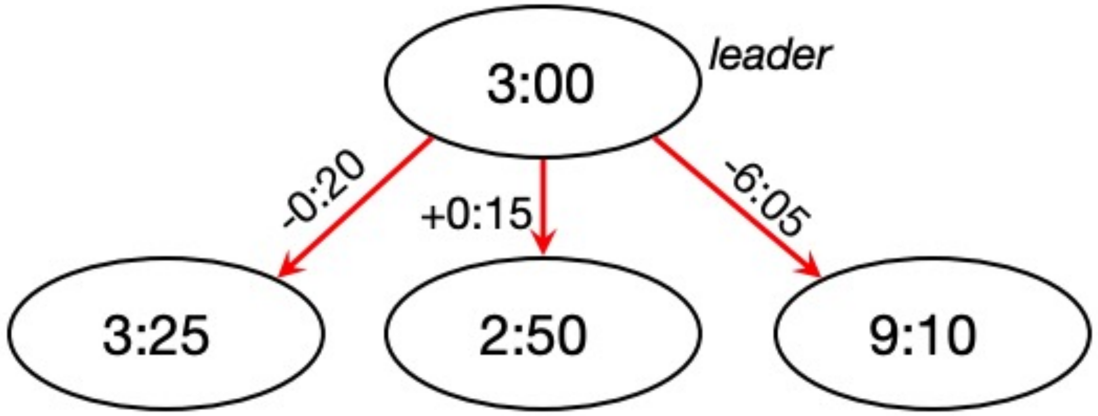


2. Compute fault-tolerant average:

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

CLOCK SYNCHRONIZATION

- **Berkeley Algorithm [Gusella & Zatti, 1989]**
  - Example:
  - 3. Send off-set to each client



CLOCK SYNCHRONIZATION

- **Berkeley Algorithm [Gusella & Zatti, 1989]**
  - Problems:
  - The Berkeley Algorithm relies on a **centralized time source** [Single point of failure]
  - The algorithm assumes that the **network delay** between the time server and all other nodes is symmetric [Not realistic]
  - Does not explicitly account for **clock drift**, which refers to the tendency of clocks to gain or lose time over time due to inaccuracies in their oscillators
  - **Scalability challenges** in larger networks with a high number of nodes

CLOCK SYNCHRONIZATION

- Computers track physical time/UTC with a quartz clock (with battery, continues running when power is off)
- Due to clock drift, clock error gradually increases
- **Clock skew:** difference between two clocks at a point in time
- Solution: Periodically get the current time from a server that has a more accurate time source (atomic clock or GPS receiver)
  - Atomic clocks are too expensive
  - Too bulky to build into every computer and phone
  - Use quartz clocks **BUT** adjust for **Clock drifts**
  - Use **Network Time Protocol (NTP), Precision Time Protocol (PTP)** for clock re-adjustment

CLOCK SYNCHRONIZATION

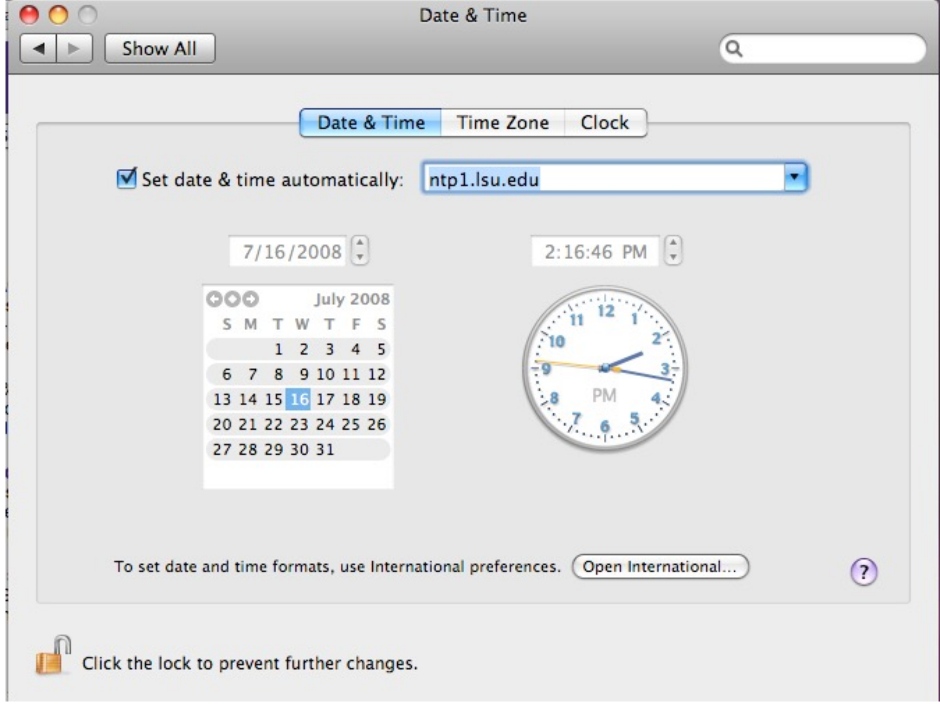
- Once the client has estimated the clock skew θ, it needs to apply that correction to its clock.
  - If $|\theta| < 125$ ms, **slew** the clock: slightly speed it up or slow it down by up to 500 ppm (brings clocks in sync within ≈ 5 minutes)
  - If $125 \text{ ms} \leq |\theta| < 1{,}000$ s, **step** the clock: suddenly reset client clock to estimated server timestamp
  - If $|\theta| \geq 1{,}000$ s, **panic and do nothing** (leave the problem for a human operator to resolve)

Systems that rely on clock sync need to monitor clock skew!

CLOCK SYNCHRONIZATION

NTP

- OS vendors run NTP Servers
- Mainstream OS have NTP clients built-in
- OS connects to NTP server for time correction


Date & Time

Date & Time | Time Zone | Clock

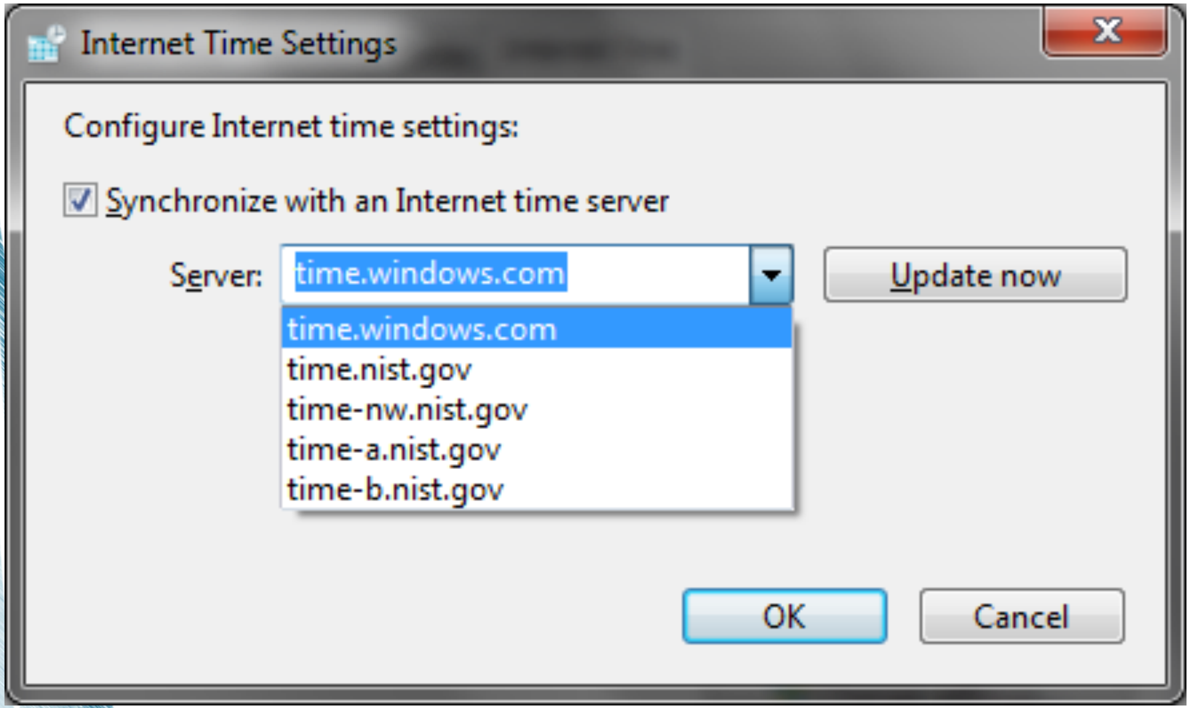
☑ Set date & time automatically: ntp1.lsu.edu

7/16/2008 2:16:46 PM

July 2008

To set date and time formats, use International preferences. Open International...

Click the lock to prevent further changes.


Internet Time Settings

Configure Internet time settings:

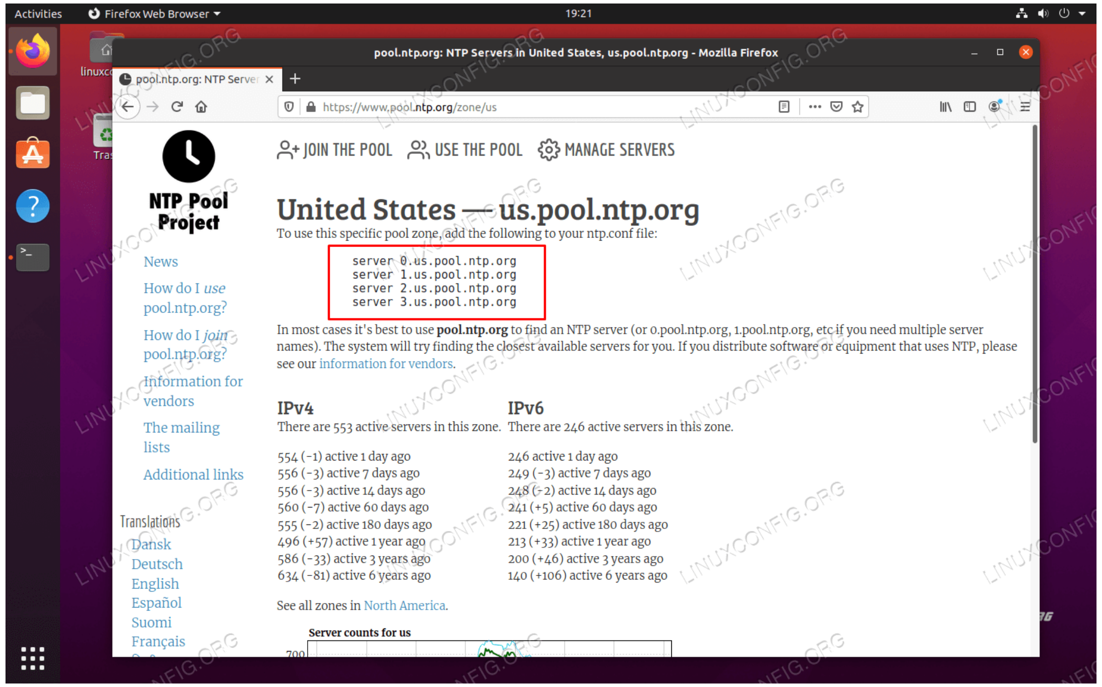
☑ Synchronize with an Internet time server

Server: time.windows.com

- time.windows.com
- time.nist.gov
- time-nw.nist.gov
- time-a.nist.gov
- time-b.nist.gov

Update now

OK Cancel


pool.ntp.org: NTP Servers in United States, us.pool.ntp.org - Mozilla Firefox

https://www.pool.ntp.org/zone/us

JOIN THE POOL USE THE POOL MANAGE SERVERS

NTP Pool Project

United States — us.pool.ntp.org

To use this specific pool zone, add the following to your ntp.conf file:

```
server 0.us.pool.ntp.org
server 1.us.pool.ntp.org
server 2.us.pool.ntp.org
server 3.us.pool.ntp.org
```

In most cases it's best to use **pool.ntp.org** to find an NTP server (or 0.pool.ntp.org, 1.pool.ntp.org, etc if you need multiple server names). The system will try finding the closest available servers for you. If you distribute software or equipment that uses NTP, please see our information for vendors.

IPv4	IPv6
There are 553 active servers in this zone.	There are 246 active servers in this zone.
554 (-1) active 1 day ago	246 active 1 day ago
556 (-3) active 7 days ago	249 (-3) active 7 days ago
556 (-3) active 14 days ago	248 (-2) active 14 days ago
560 (-7) active 60 days ago	241 (+5) active 60 days ago
555 (-2) active 180 days ago	221 (+25) active 180 days ago
496 (+57) active 1 year ago	213 (+33) active 1 year ago
586 (-33) active 3 years ago	200 (+46) active 3 years ago
634 (-81) active 6 years ago	140 (+106) active 6 years ago

See all zones in North America.

Server counts for us

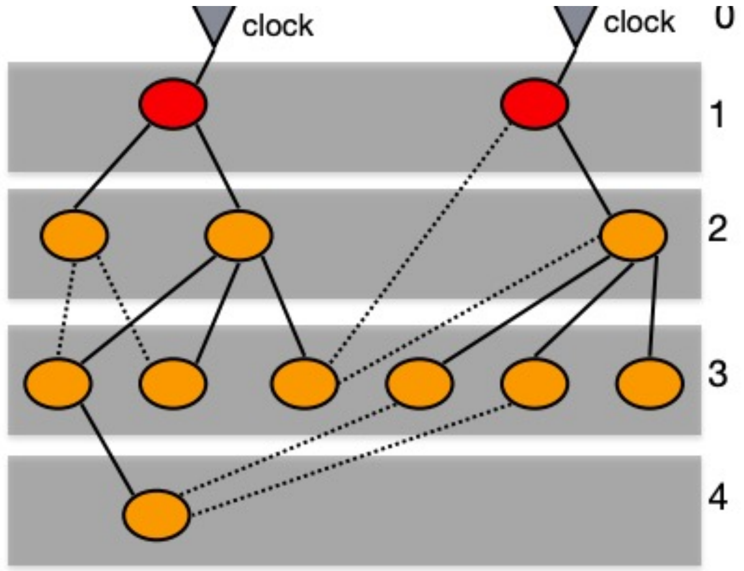
CLOCK SYNCHRONIZATION

- Idea: NTP client contact multiple servers, discard outliers, average results
- Problem: Network latency!
- Reduces clock skew to a few milliseconds in good network conditions, but can be much worse!
  - Latency is unpredictable!
  - Geographical local of server
  - Servers with long queues: Slow response

CLOCK SYNCHRONIZATION

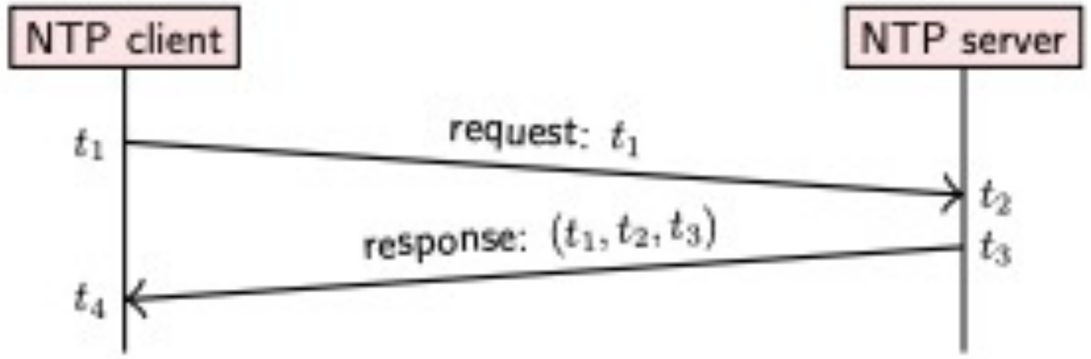
Arranged in ***strata***

- **Stratum 0** = master clock
- **Stratum 1**: systems connected directly to accurate time source
- **Stratum 2**: systems synchronized from 1st stratum systems
- …
- **Stratum 15**: systems synchronized from 14th stratum systems



Synchronization Subnet

CLOCK SYNCHRONIZATION



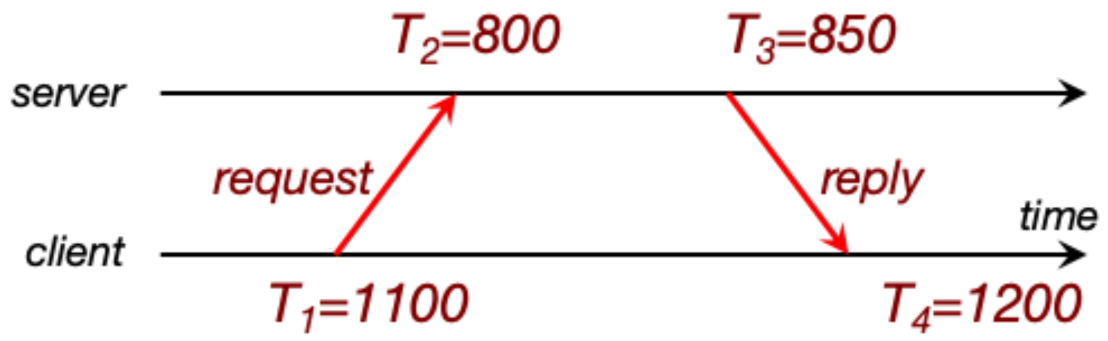
Round-trip network delay: $\delta = (t_4 - t_1) - (t_3 - t_2)$

Estimated server time when client receives response: $t_3 + \frac{\delta}{2}$

Estimated clock skew: $\theta = t_3 + \frac{\delta}{2} - t_4 = \frac{t_2 - t_1 + t_3 - t_4}{2}$

CLOCK SYNCHRONIZATION

SNTP Example



Offset = ((800 - 1100) + (850 - 1200)) / 2

= ((-300) + (-350)) / 2

= -650 / 2 = **-325**

Time offset: $t = \dfrac{(T_2 - T_1) + (T_3 - T_4)}{2}$

Set time to $T_4 + t$ = 1200 - 325 = 875

CLOCK SYNCHRONIZATION

Precision Time Protocol (PTP): IEEE 1588 Precision Time Protocol

- Designed to synchronize clocks on a LAN to sub-microsecond precision
  - Designed for LANs, not global: low jitter, low latency
  - Timestamps ideally generated at the MAC or PHY layers to minimize delay and jitter
- Determine master clock (called the Grandmaster)
  - Use a Best Master Clock algorithm to determine which clock is most precise
  - The Grandmaster sends periodic synchronization messages to others (slave devices)
- Two phases in synchronization
  - 1. Offset correction
  - 2. Delay correction

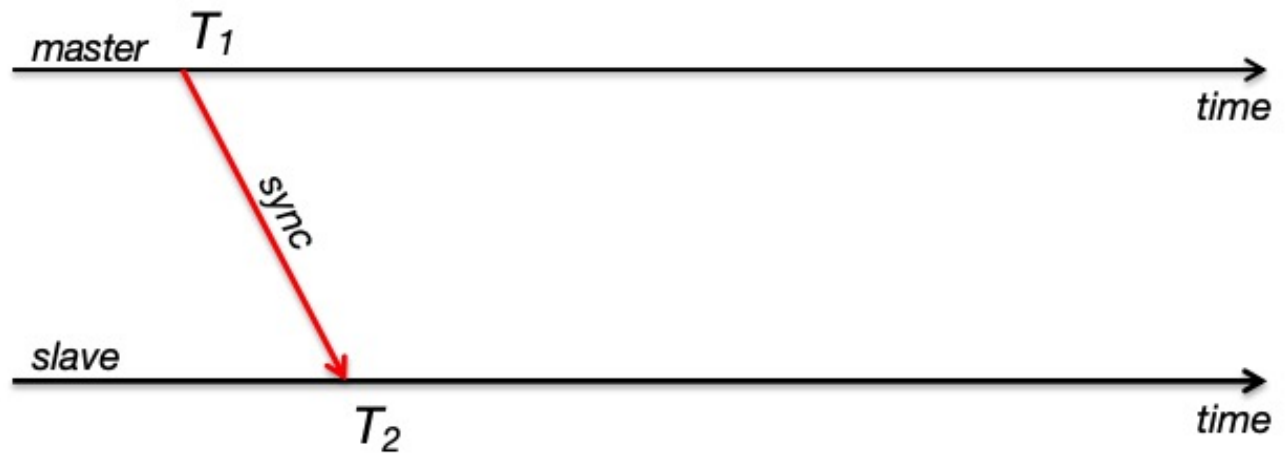
CLOCK SYNCHRONIZATION

Precision Time Protocol (PTP): IEEE 1588 Precision Time Protocol

- Chooses the Best Master Clock
- Distributed election based on properties of clocks
- Criteria from highest to lowest:
  - Priority 1 (admin-defined hint)
  - Clock class
  - Clock accuracy
  - Clock variance: estimate of stability based on past syncs
  - Priority 2 (admin-defined hint #2)
  - Unique ID (tie-breaker)

CLOCK SYNCHRONIZATION

Precision Time Protocol (PTP): IEEE 1588 Precision Time Protocol



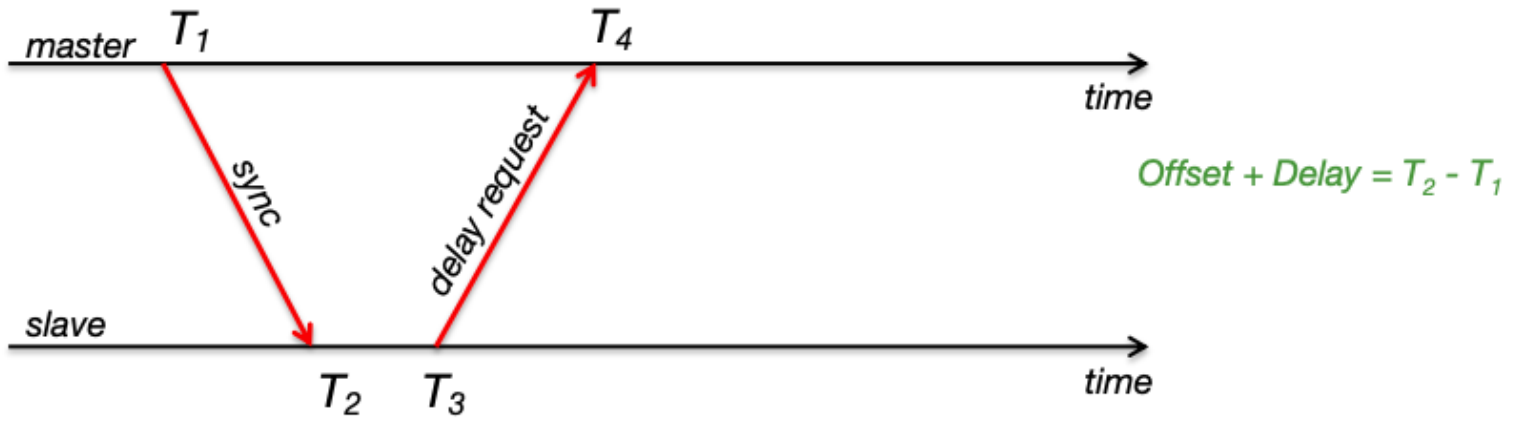
Master initiates the protocol by sending a *sync* message containing a timestamp

Slave timestamps arrival with a timestamp from its local clock

Offset + Delay $= T_2 - T_1$

CLOCK SYNCHRONIZATION

Precision Time Protocol (PTP): IEEE 1588 Precision Time Protocol

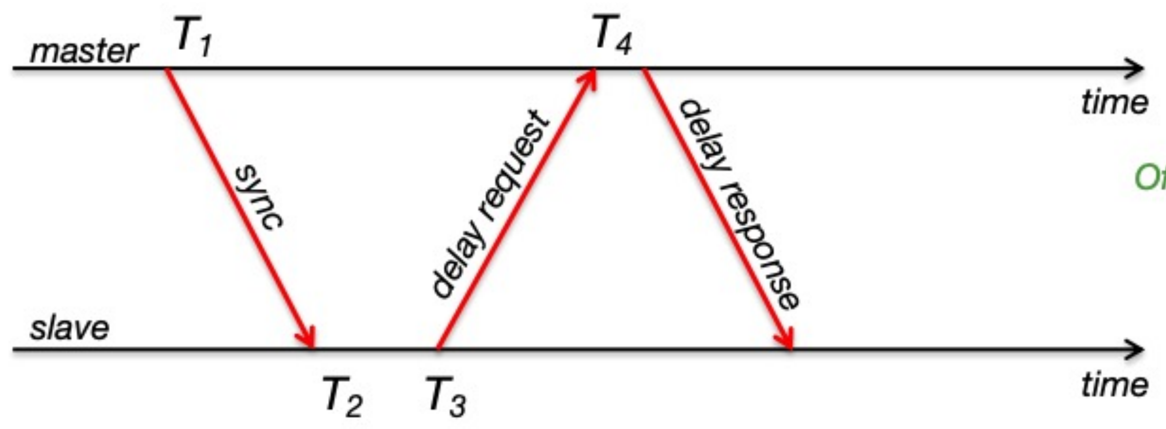


Slave needs to figure out the network delay. Send a *delay request*

Note the time it was sent

CLOCK SYNCHRONIZATION

Precision Time Protocol (PTP): IEEE 1588 Precision Time Protocol



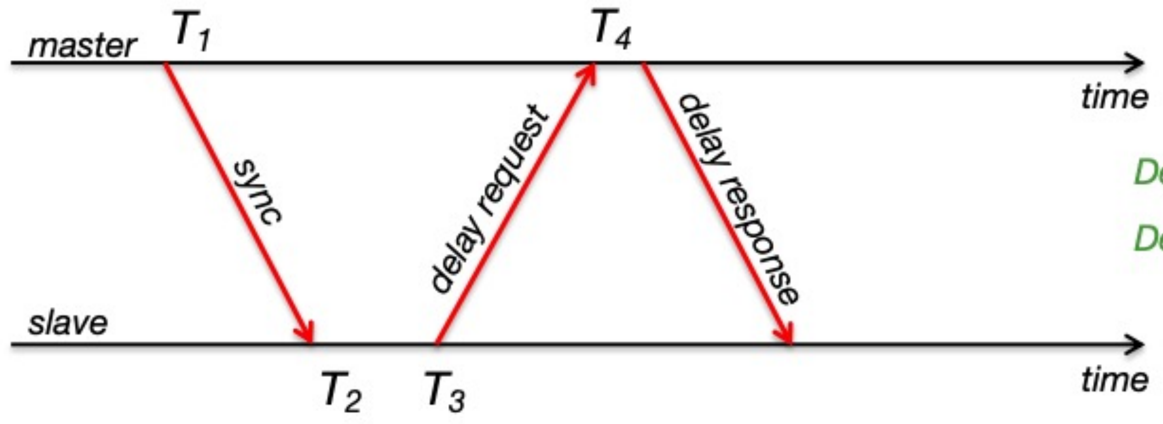
Offset + Delay $= T_2 - T_1$

Master marks the time of arrival and returns it in a *delay response*

Delay response = Delay - Offset $= T_4 - T_3$

CLOCK SYNCHRONIZATION

Precision Time Protocol (PTP): IEEE 1588 Precision Time Protocol



$Delay + Offset = T_2 - T_1$

$Delay - Offset = T_4 - T_3$

$$
\begin{array}{rl}
 & master\_slave\_difference = T_2 - T_1 = delay + offset \\
- & slave\_master\_difference = T_4 - T_3 = delay - offset \\
\hline
\end{array}
$$

$master\_slave\_difference - slave\_master\_difference = 2(offset)$

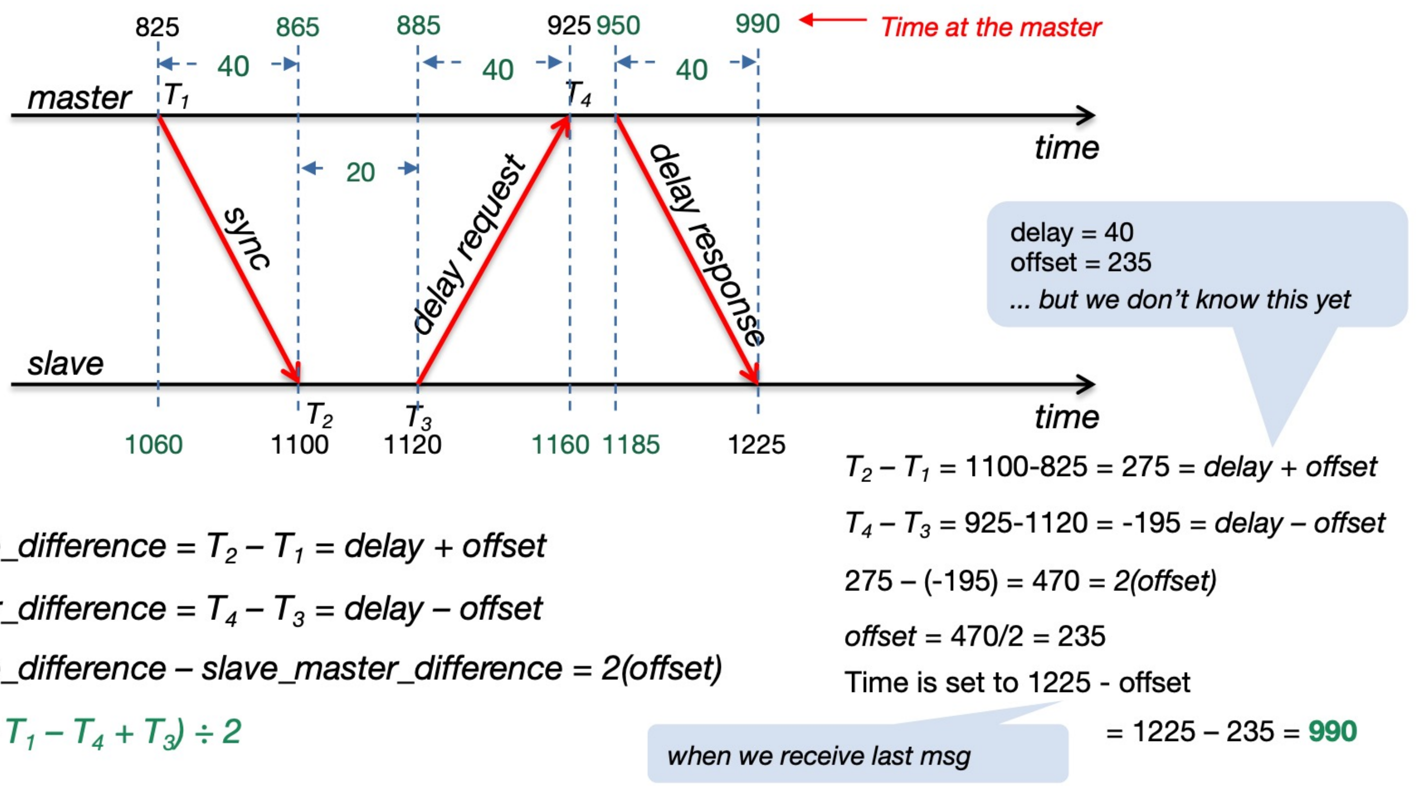
$(T_2 - T_1) - (T_4 - T_3) = T_2 - T_1 - T_4 + T_3 = 2(offset)$

$offset = (T_2 - T_1 - T_4 + T_3) \div 2$

CLOCK SYNCHRONIZATION

Precision Time Protocol (PTP): IEEE 1588 Precision Time Protocol

Example



$T_2 - T_1$ = 1100-825 = 275 = *delay* + *offset*

$T_4 - T_3$ = 925-1120 = -195 = *delay* – *offset*

275 – (-195) = 470 = 2(*offset*)

offset = 470/2 = 235

Time is set to 1225 - offset

= 1225 – 235 = **990**

master_slave_difference = $T_2 - T_1$ = *delay* + *offset*

slave_master_difference = $T_4 - T_3$ = *delay* – *offset*

master_slave_difference – *slave_master_difference* = 2(*offset*)

offset = $(T_2 - T_1 - T_4 + T_3) \div 2$

CLOCK SYNCHRONIZATION

NTP vs PTP

Range:

- NTP: nodes widely spread out on the Internet
- PTP: LAN -> Usually implemented at the physical layer to eliminate OS & scheduling overhead

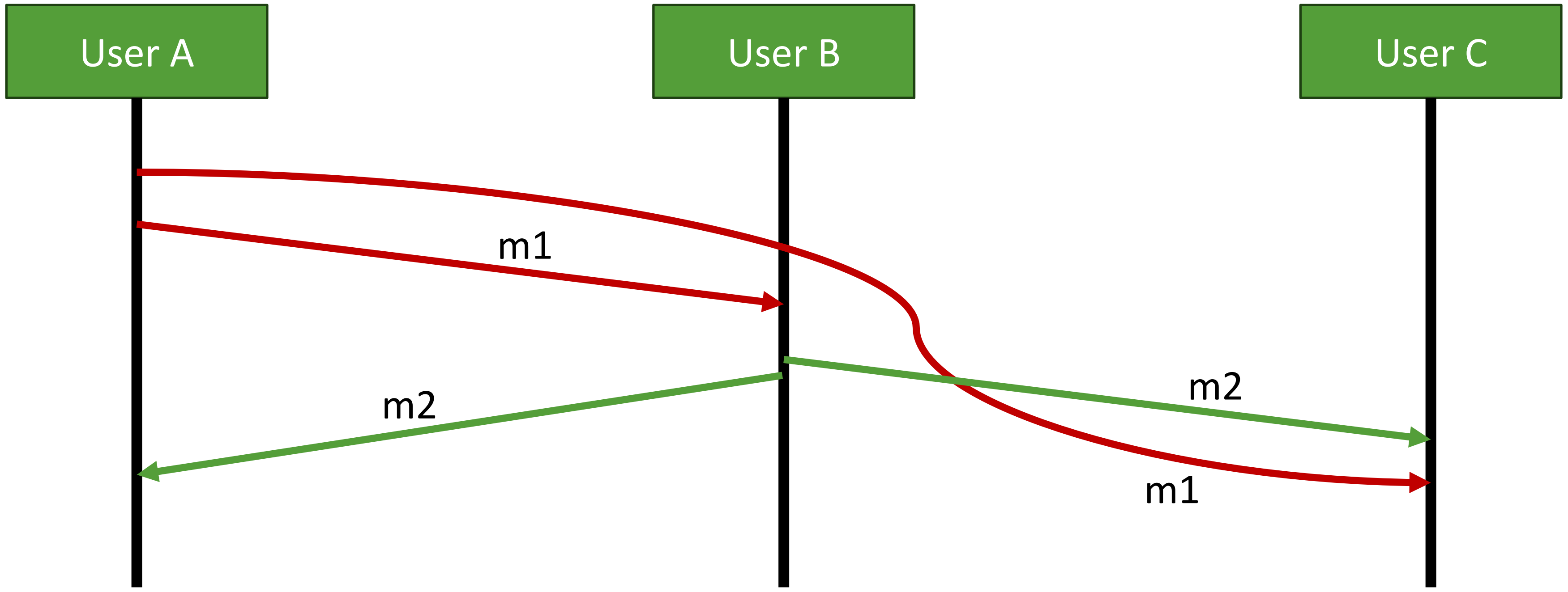
Accuracy

- NTP usually several milliseconds on WAN
- PTP usually sub-microsecond on LAN (around 1 µs)
- PTP can be 10,000x more precise than NTP!

ORDERING OF MESSAGES IN DIST. SYS.

ORDERING OF MESSAGES

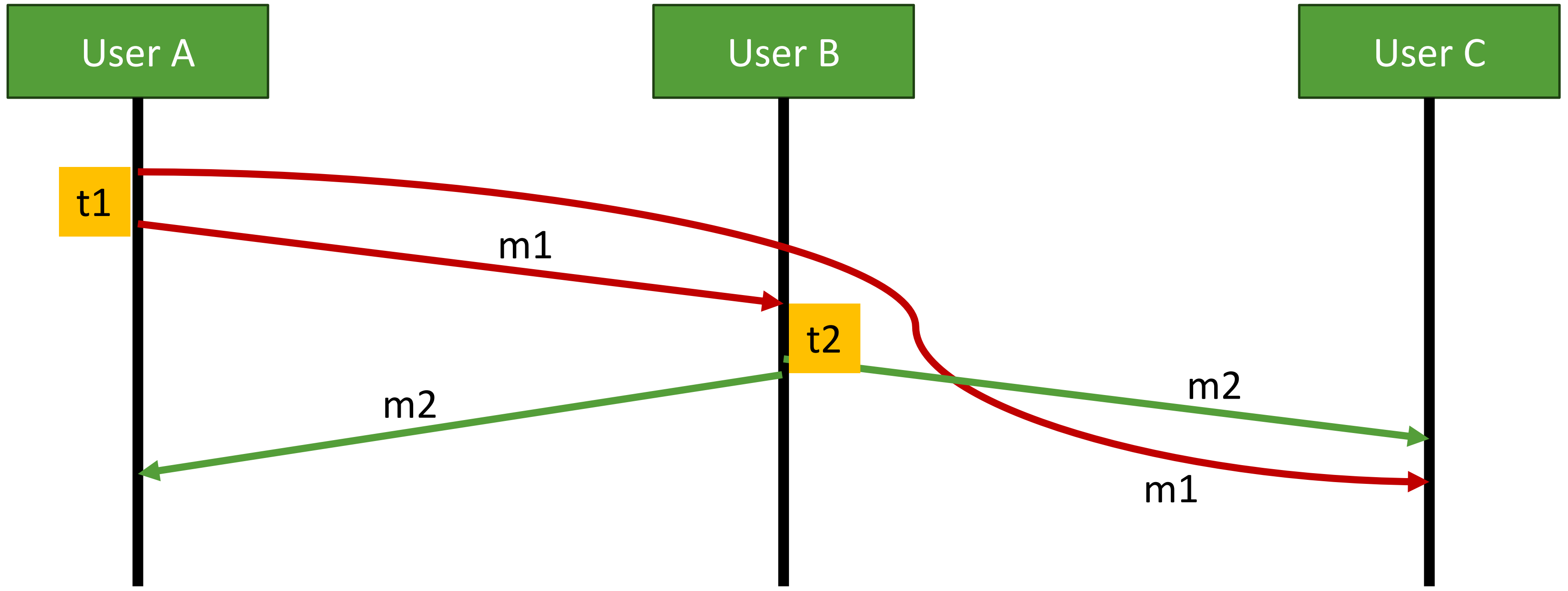
- Consider a scenario with 3 nodes A, B and C group communicating
  - A makes a statement m1 and multi-casts it to B and C
  - m1 is received at B immediately, C receives it late due to latency
  - On receiving m1, B reacts by sending m2 and copies m2 to C.
  - Now C receives m2 and then m1
- Real-life example:
  - Database transactions
  - State variables update etc.



m1 = “User A says: Coffee is hot”

m2 = “User B says: No its cold!”

- User C see m2 first and then m1, even though logically m1 **happened before** m2
- How can C determine the correct order in which it should put the messages?
- A monotonic clock won’t work since its timestamps are not comparable across nodes.
- Solution: Send timestamps



m1 = t1, "User A says: Coffee is hot"

m2 = t2, "User B says: No its cold!"

- A sends m1 with timestamp t1 according to A's clock.
- When B receives m1, the timestamp according to B's clock is t2, where t2 < t1, because A's clock is slightly ahead of B's clock.
- So, if we order messages based on their timestamps from time-of-day clocks, we might again end up with the wrong order.

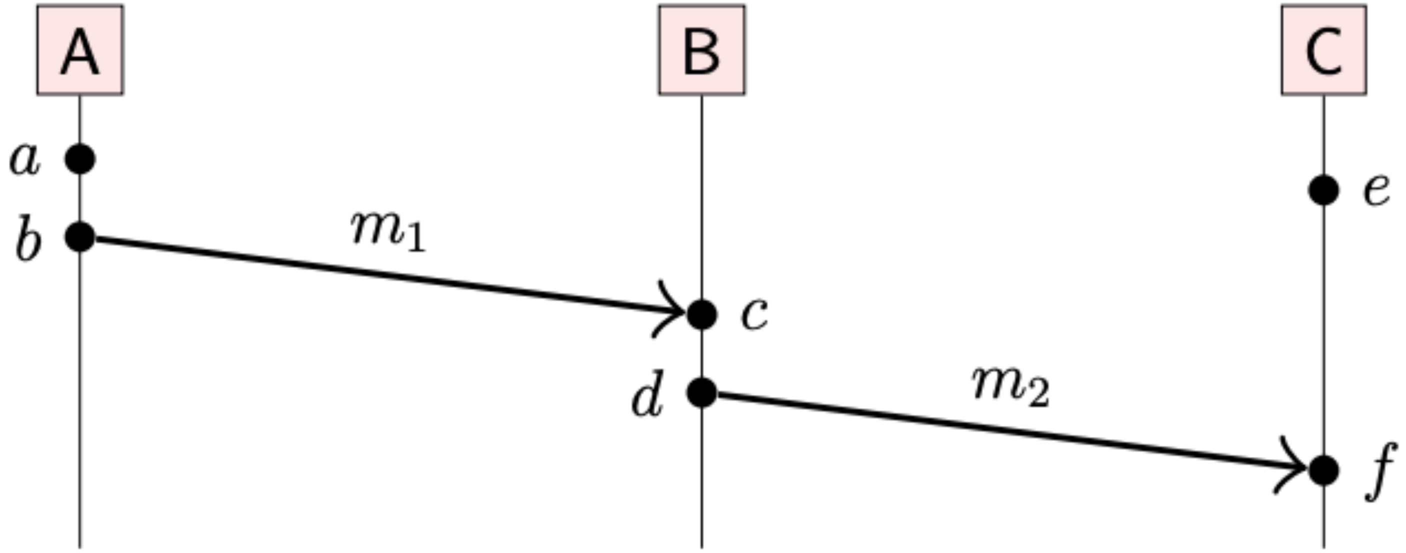
ORDERING OF MESSAGES

- To get “correct order”, define a **happens-before** rule:
  - Event *a* happens before event *b* (written *a* → *b*) iff:
  - *a* and *b* occurred at the same node, *a* occurred before *b* in the local node OR
  - *a* is sending of a message m, *b* is receiving of the message m
  - There exists a event *c* such that *a* → *c and c* → *b*

The happens-before relation is a partial order, i.e. it is possible that neither

a* → *b nor ***b* → *a***.

In that case, a and b are concurrent (written *a* || *b*).



- $a \rightarrow b$, $c \rightarrow d$, and $e \rightarrow f$ due to node execution order
- $b \rightarrow c$ and $d \rightarrow f$ due to messages m_1 and m_2
- $a \rightarrow c$, $a \rightarrow d$, $a \rightarrow f$, $b \rightarrow d$, $b \rightarrow f$, and $c \rightarrow f$ due to transitivity
- $a \parallel e$, $b \parallel e$, $c \parallel e$, and $d \parallel e$

ORDERING OF MESSAGES

Causality:

- The **happens-before** relation is a way of reasoning about **causality** in distributed systems.
- Causality considers whether information could have flowed from one event to another, and thus whether one event may have influenced another.

m1 = t1, "User A says: Coffee is hot"

m2 = t2, "User B says: No its cold!"

In the previous example: message m1 "caused" the message m2

ORDERING OF MESSAGES

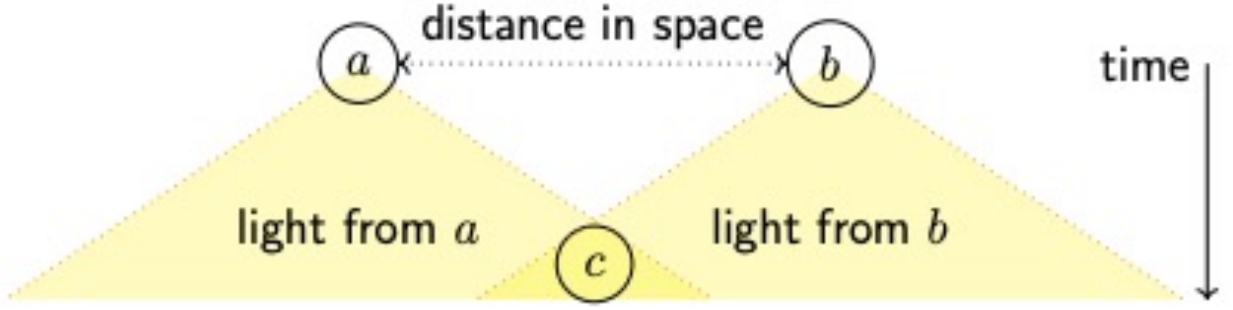
Causality:

Taken from physics (relativity).

When *a* → *b*, then *a* might have caused *b*.

When *a* || *b*, we know that *a* cannot have caused *b*.

Concept taken from Physics: it is not possible for information to travel faster than the speed of light



It is impossible for a signal sent from *a* to arrive at *b*'s location before event *b*, and vice versa. Therefore, *a* and *b* must be **causally unrelated**

It is possible for a signal from *a* to reach c, and therefore *a* might influence *c*

In distributed systems, we usually work with messages on a network rather than beams of light, but the principle is very similar.

CLOCK ALGORITHMS

CLOCK ALGORITHMS

- Distributed systems often broadcast messages (multi-cast)
- Several different broadcast protocols are used in practice, and their main difference is the *order* in which they deliver messages.
- Important to understand how clocks are needed for synchronization

- **Physical clock**: count number of seconds elapsed
- **Logical clock**: count number of events occurred
- **Physical timestamps**: useful for many things, but may be ***inconsistent with causality***.
- **Logical clocks**: designed to capture causal dependencies.

$$(a \rightarrow b) \Rightarrow T(a) < T(b)$$

CLOCK ALGORITHMS

Lamport clock algorithm

```
on initialisation do
    $t := 0$                ▷ each node has its own local variable $t$
end on

on any event occurring at the local node do
    $t := t + 1$
end on

on request to send message $m$ do
    $t := t + 1$; send $(t, m)$ via the underlying network link
end on

on receiving $(t', m)$ via the underlying network link do
    $t := \max(t, t') + 1$
    deliver $m$ to the application
end on
```

CLOCK ALGORITHMS

Lamport clock algorithm

- Each node maintains a counter *t*, incremented on every local event *e*
- Let L(*e*) be the value of *t* after that increment
- Attach current *t* to messages sent over network
- Recipient moves its clock forward to timestamp in the message (if greater than local counter), then increments

Properties of this scheme:

- If $a \rightarrow b$ then $L(a) < L(b)$
- However, $L(a) < L(b)$ does not imply $a \rightarrow b$
- Possible that L(a) = L(b) for (a != b)

CLOCK ALGORITHMS

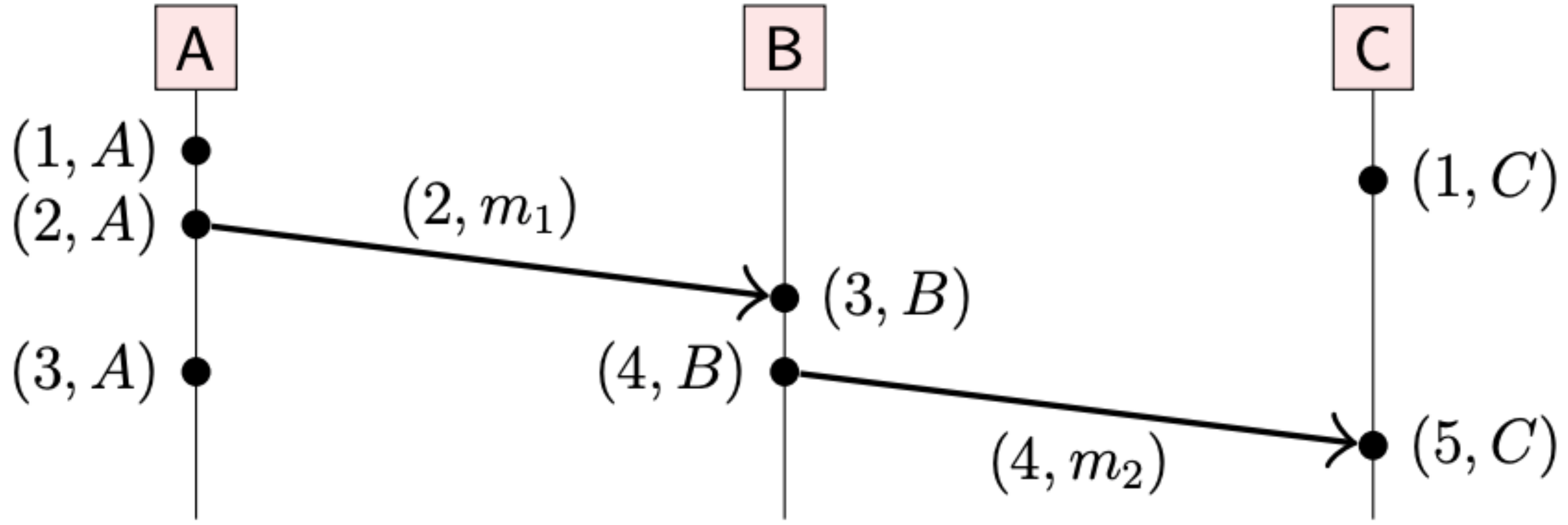
Lamport clock algorithm

- A Lamport timestamp is essentially an integer that counts the number of events that have occurred.
- As such, it has no direct relationship to physical time.
- On each node, time increases because the integer is incremented on every event.
- The algorithm assumes a crash-stop model (or a crash-recovery model if the timestamp is maintained in stable storage, i.e. on disk).

CLOCK ALGORITHMS

Lamport clock algorithm

- When a message is sent over the network, the sender attaches its current Lamport timestamp to that message



- t = 2 is attached to m1 and t = 4 is attached to m2.
- When the C receives a message, it moves its local Lamport clock forward to the timestamp in the message plus one

CLOCK ALGORITHMS

Lamport clock algorithm

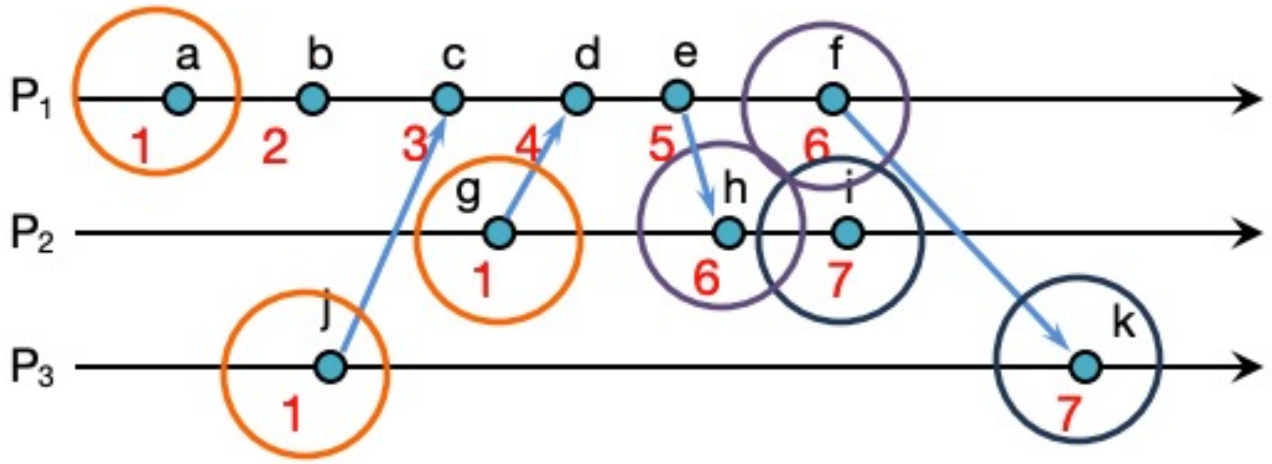
- Lamport timestamps have the property that if *a* happened before *b*, then *b* always has a greater timestamp than *a*; in other words, **the timestamps are consistent with causality.**
- It is also possible for two different events to have the same timestamp
- If we need a unique timestamp for every event, each timestamp can be extended with the name or identifier of the node on which that event occurred.

Given the Lamport timestamps of two events, it is in general not possible to tell whether those events are concurrent or whether one happened before the other. If we do want to detect when events are concurrent, we need a different type of logical time: **a vector clock.**

CLOCK ALGORITHMS

Lamport clock algorithm

- It is also possible for two different events to have the same timestamp



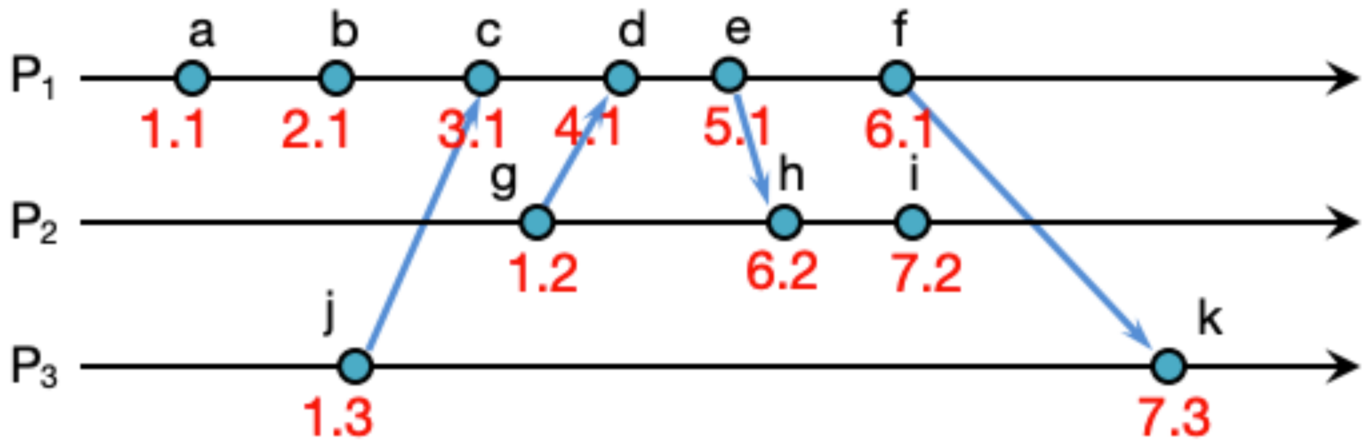
$a \rightarrow b$, $b \rightarrow c$, … : local events sequenced

$i \rightarrow c$, $f \rightarrow d$, $d \rightarrow g$, … : Lamport imposes a *send*→*receive* relationship

CLOCK ALGORITHMS

Lamport clock algorithm

- If we need a unique timestamp for every event, each timestamp can be extended with the name or identifier of the node on which that event occurred.



If $L(e) < L(e')$

- We cannot conclude that $e \rightarrow e'$

By looking at Lamport timestamps

- We cannot conclude which events are causally related

Solution: use a **vector clock**

CLOCK ALGORITHMS

Vector clock algorithm

- Lamport timestamps are just a single integer, **vector timestamps are a list of integers**, one for each node in the system.
  - Assume *n* nodes in the system, $N = \{N1, N2, \ldots, Nn\}$
  - Vector timestamp of event *a* is V (*a*) = $\{t1, t2, \ldots, tn\}$
  - t_i is number of events observed by node N_i
  - Each node has a current vector timestamp *T*
  - On event at node *Ni*, increment vector element T[i]
  - Attach current vector timestamp to each message
  - Recipient merges message vector into its local vector

Apart from the difference between a scalar and a vector, the vector clock algorithm is very similar to a Lamport clock

CLOCK ALGORITHMS

Vector clock algorithm

```
on initialisation at node $N_i$ do
    $T := \langle 0, 0, \ldots, 0 \rangle$          ▷ local variable at node $N_i$
end on

on any event occurring at node $N_i$ do
    $T[i] := T[i] + 1$
end on

on request to send message $m$ at node $N_i$ do
    $T[i] := T[i] + 1$; send $(T, m)$ via network
end on

on receiving $(T', m)$ at node $N_i$ via the network do
    $T[j] := \max(T[j], T'[j])$ for every $j \in \{1, \ldots, n\}$
    $T[i] := T[i] + 1$; deliver $m$ to the application
end on
```

CLOCK ALGORITHMS

Vector clock algorithm

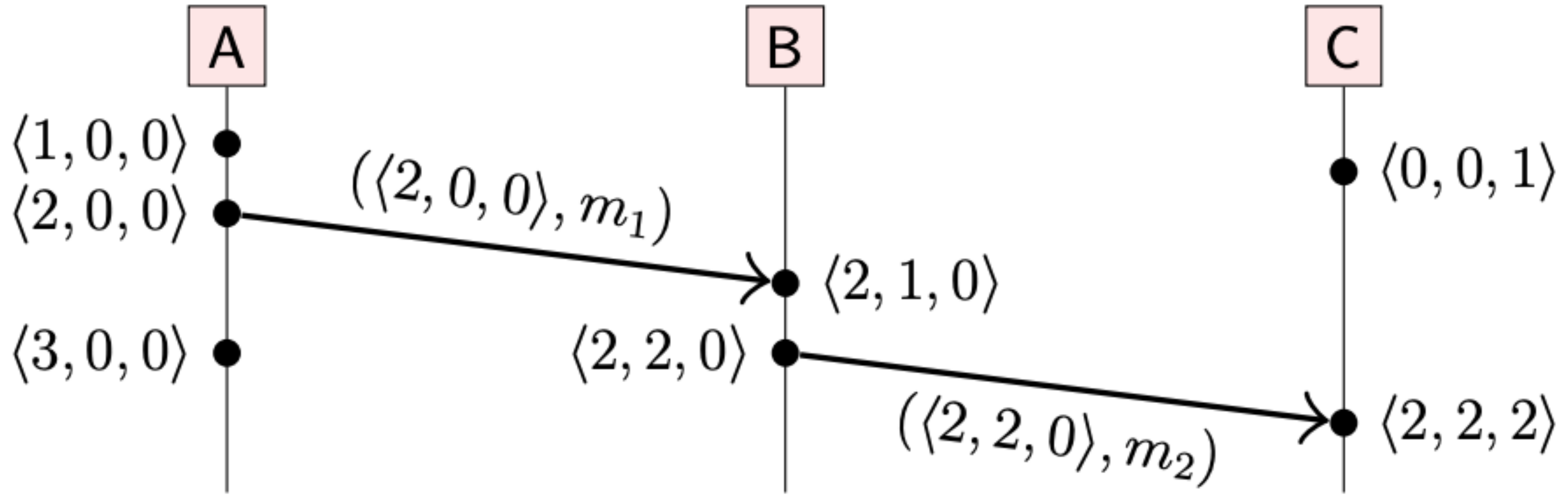
- A node initializes its vector clock to contain a zero for each node in the system.
- Whenever an event occurs at node Ni , it increments the ith entry (its own entry) in its vector clock.
- When a message is sent over the network, the sender's current vector timestamp is attached to the message.
- Finally, when a message is received, the recipient merges the vector timestamp in the message with its local timestamp by taking the element-wise maximum of the two vectors, and then the recipient increments its own entry.

CLOCK ALGORITHMS

Vector clock algorithm

Assuming the vector of nodes is N = {A, B, C}

The vector timestamp of an event *e* represents a set of events, *e* and its causal dependencies

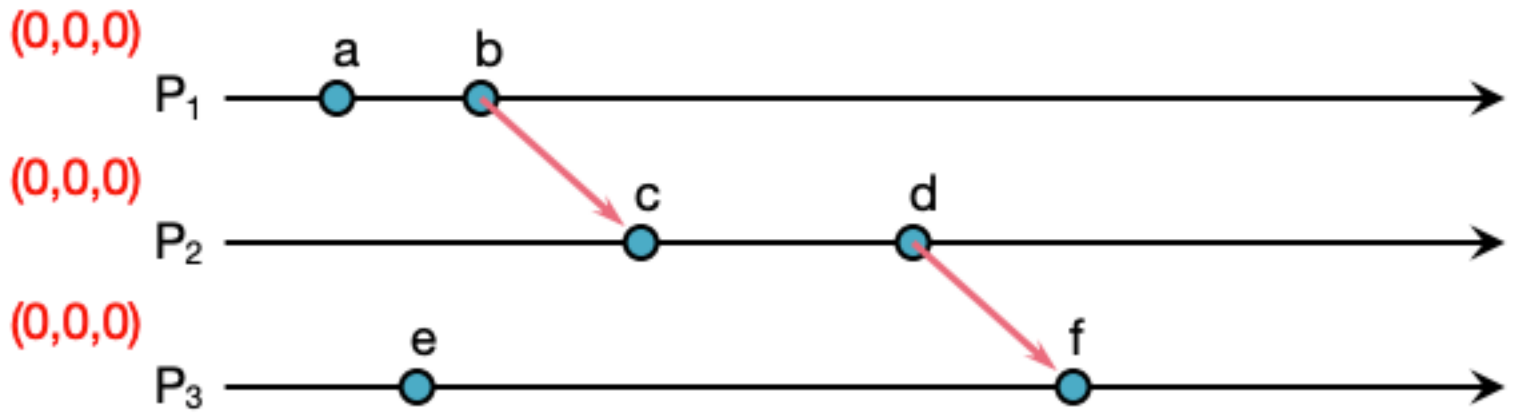


For example, (2, 2, 0) represents the first two events from A, the first two events from B, and no events from C

CLOCK ALGORITHMS

Vector clock algorithm

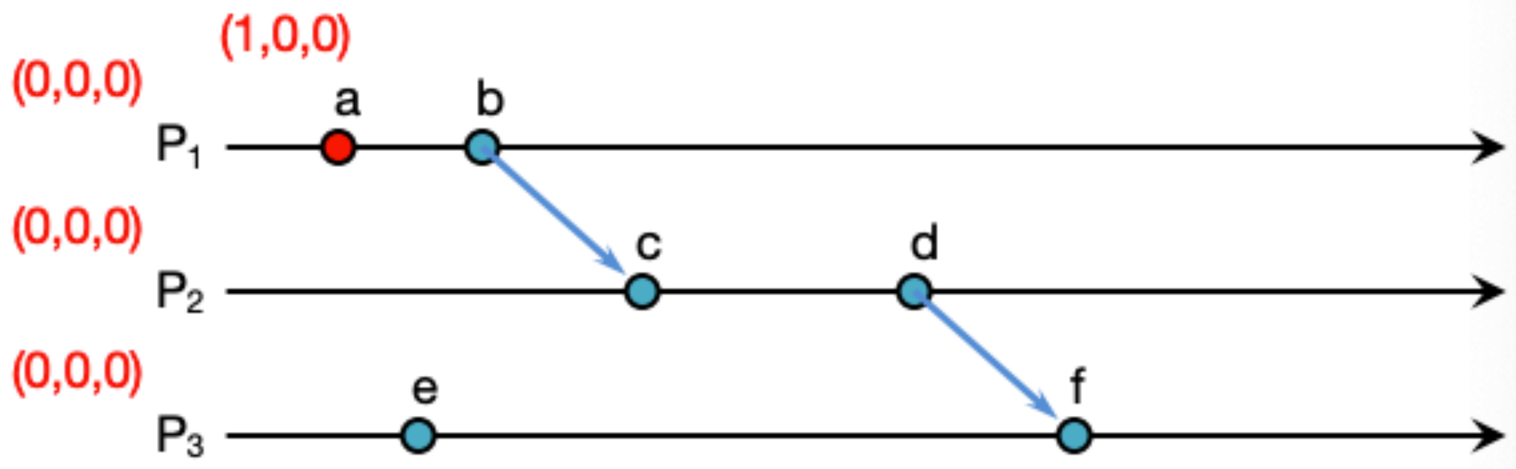
Example



CLOCK ALGORITHMS

Vector clock algorithm

Example

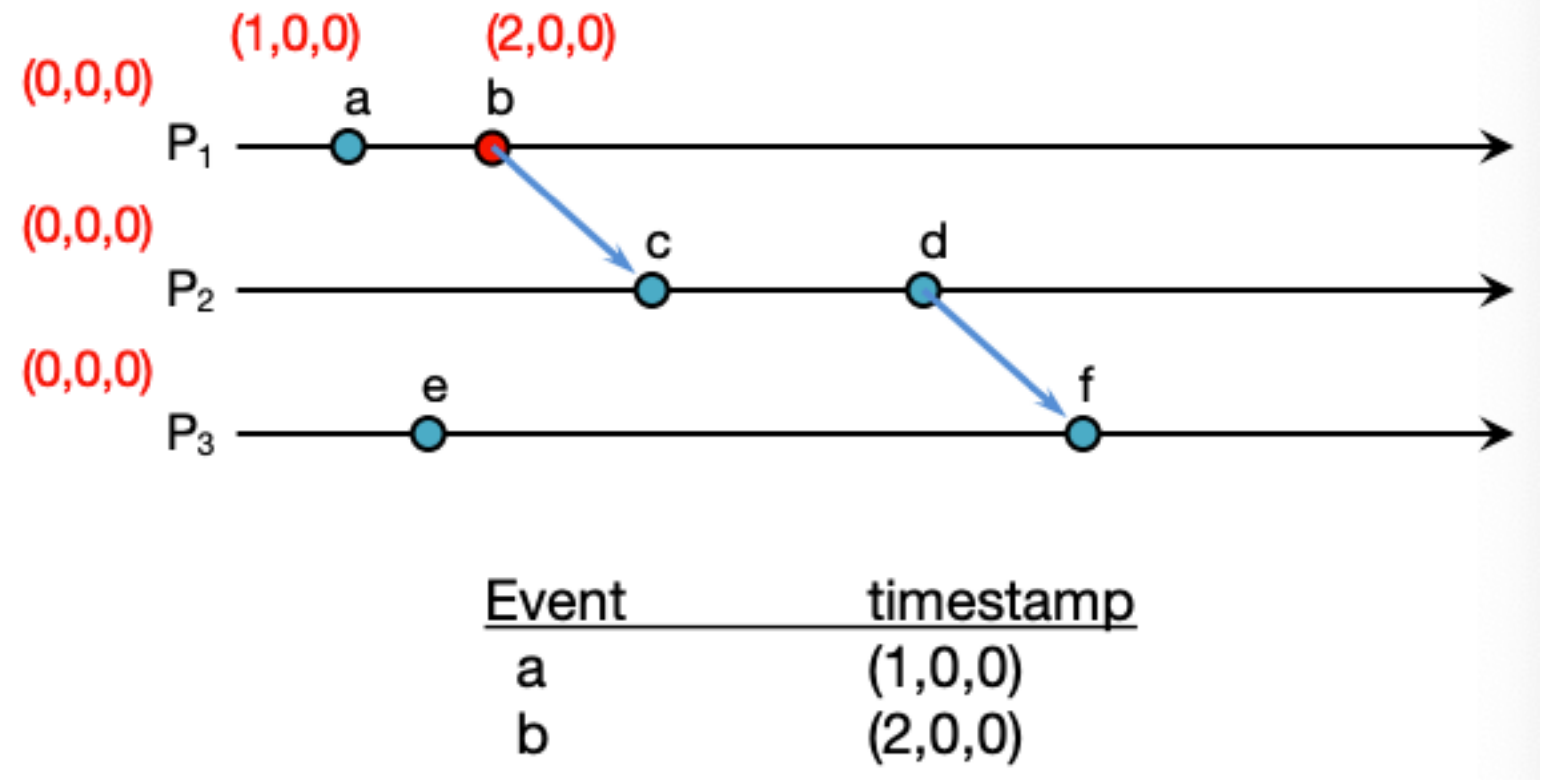


Event	timestamp
a	(1,0,0)

CLOCK ALGORITHMS

Vector clock algorithm

Example

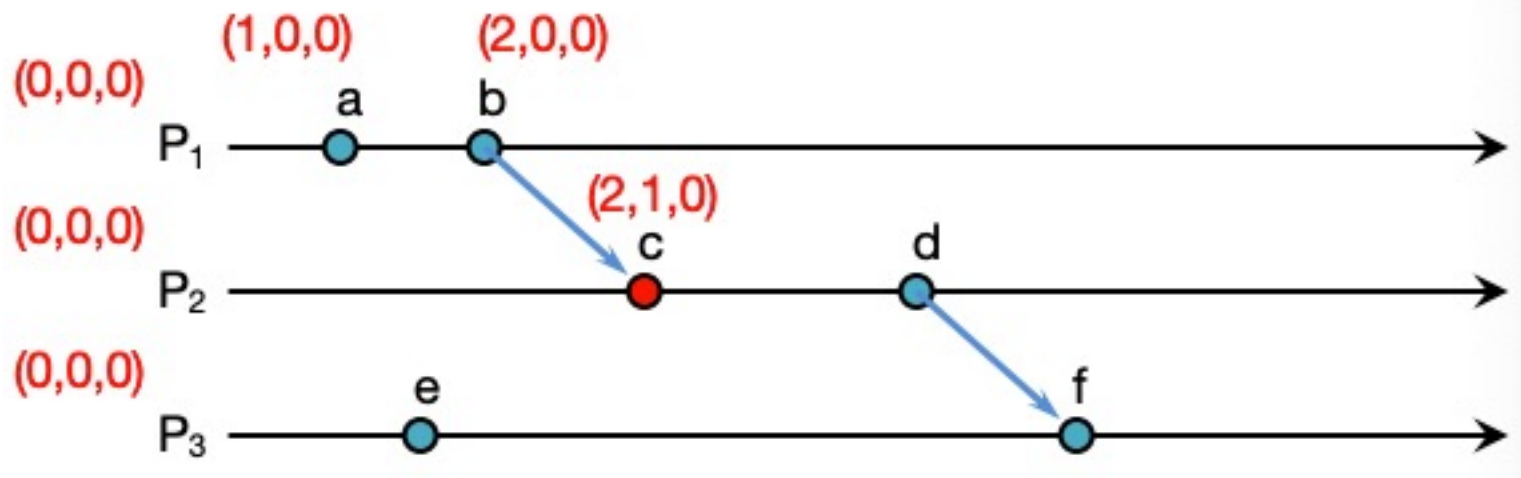


Event	timestamp
a	(1,0,0)
b	(2,0,0)

CLOCK ALGORITHMS

Vector clock algorithm

Example

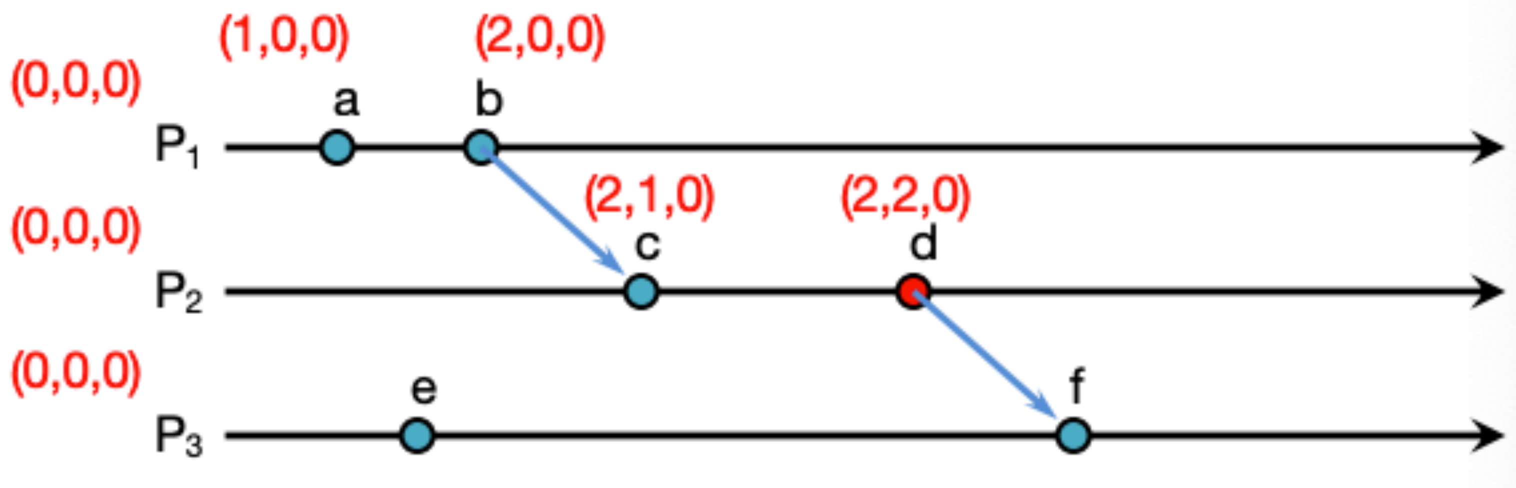


Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)

CLOCK ALGORITHMS

Vector clock algorithm

Example

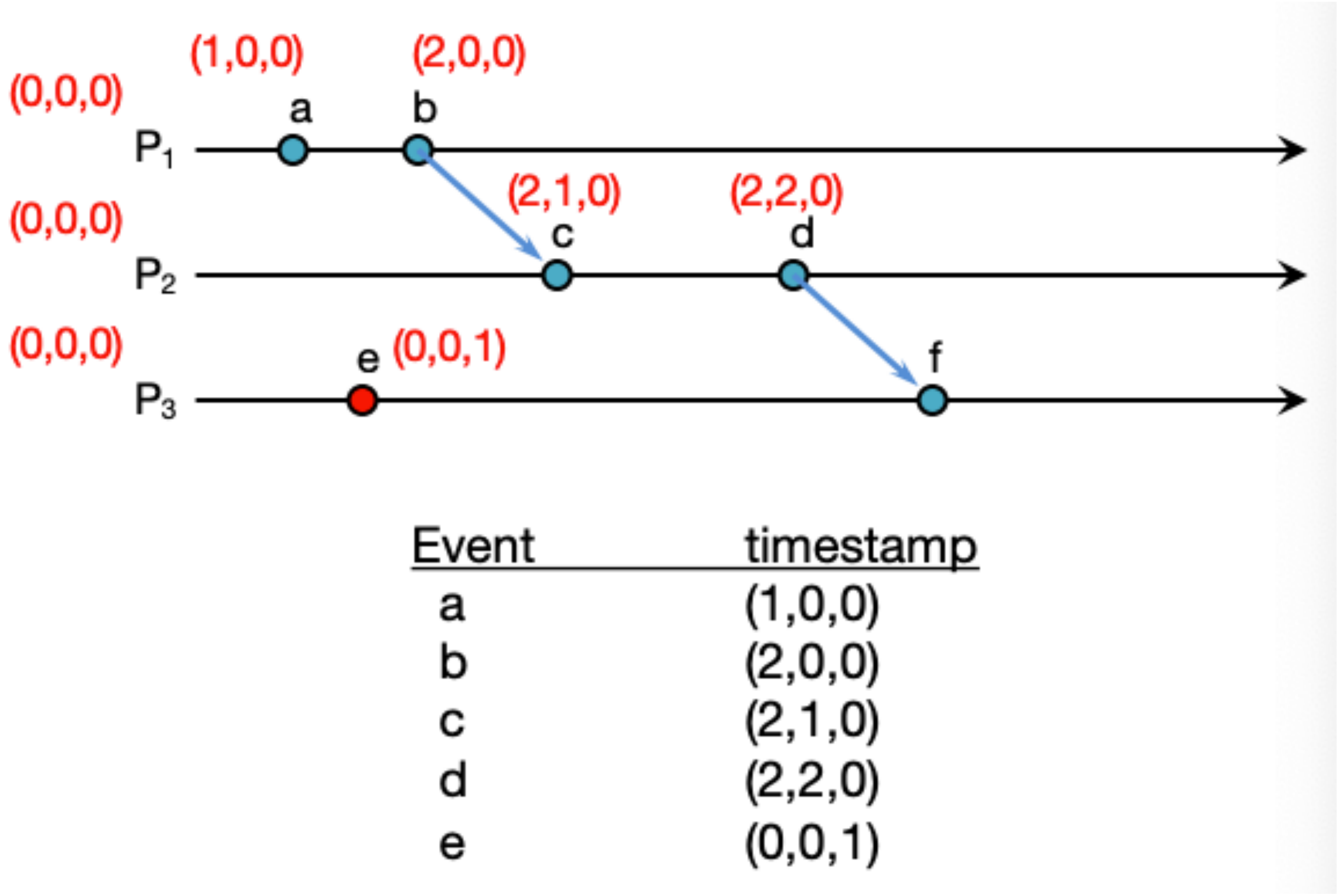


Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)

CLOCK ALGORITHMS

Vector clock algorithm

Example

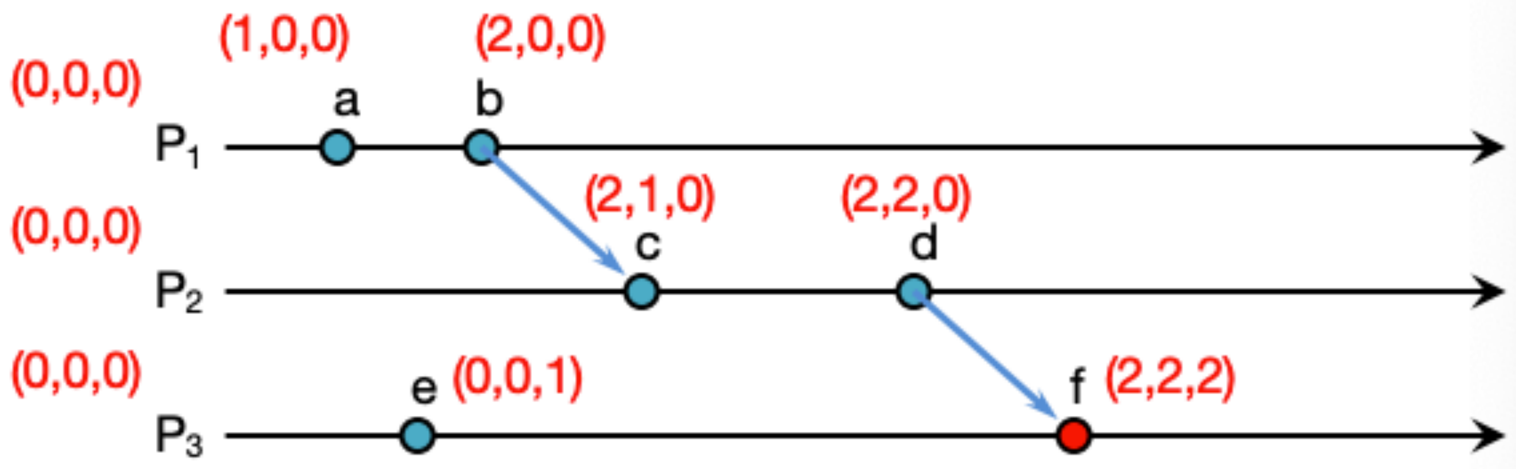


Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)

CLOCK ALGORITHMS

Vector clock algorithm

Example

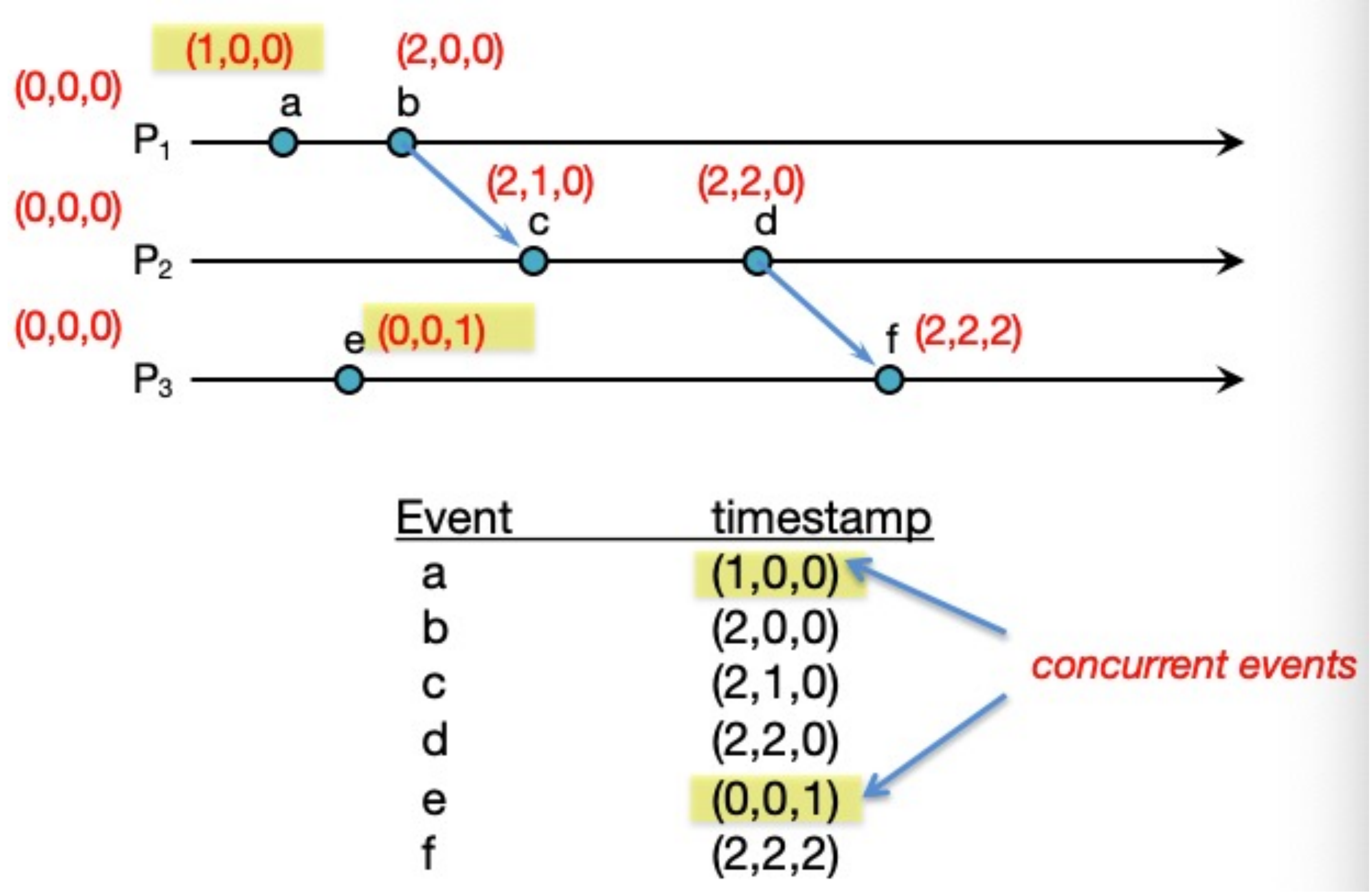


Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)

CLOCK ALGORITHMS

Vector clock algorithm

Example

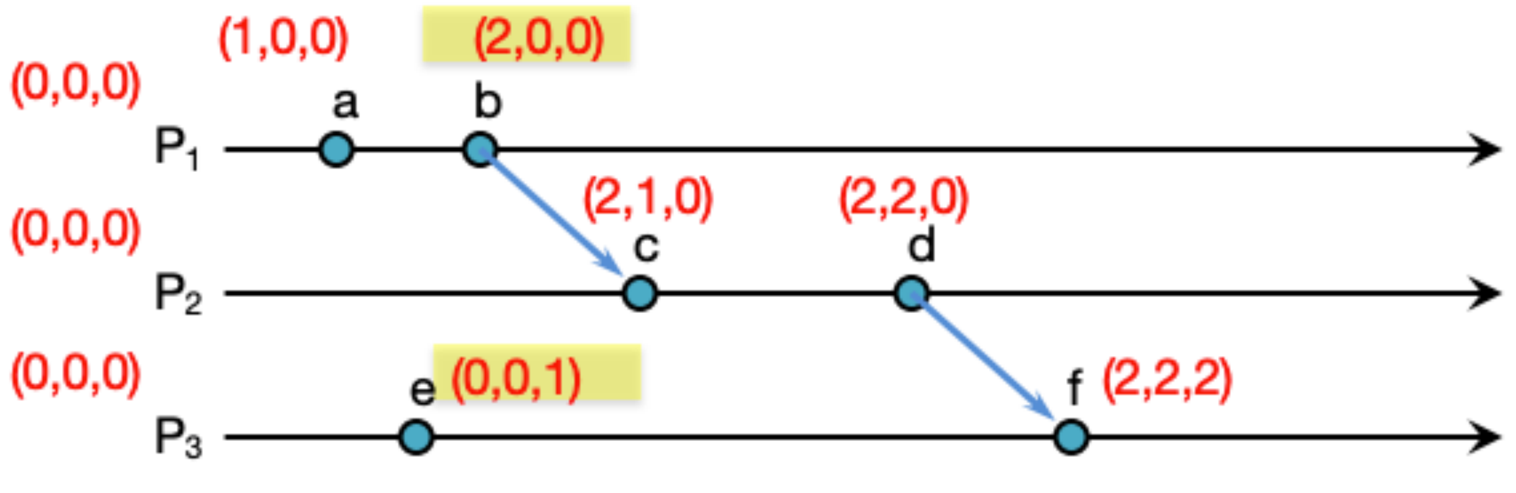


Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)

CLOCK ALGORITHMS

Vector clock algorithm

Example



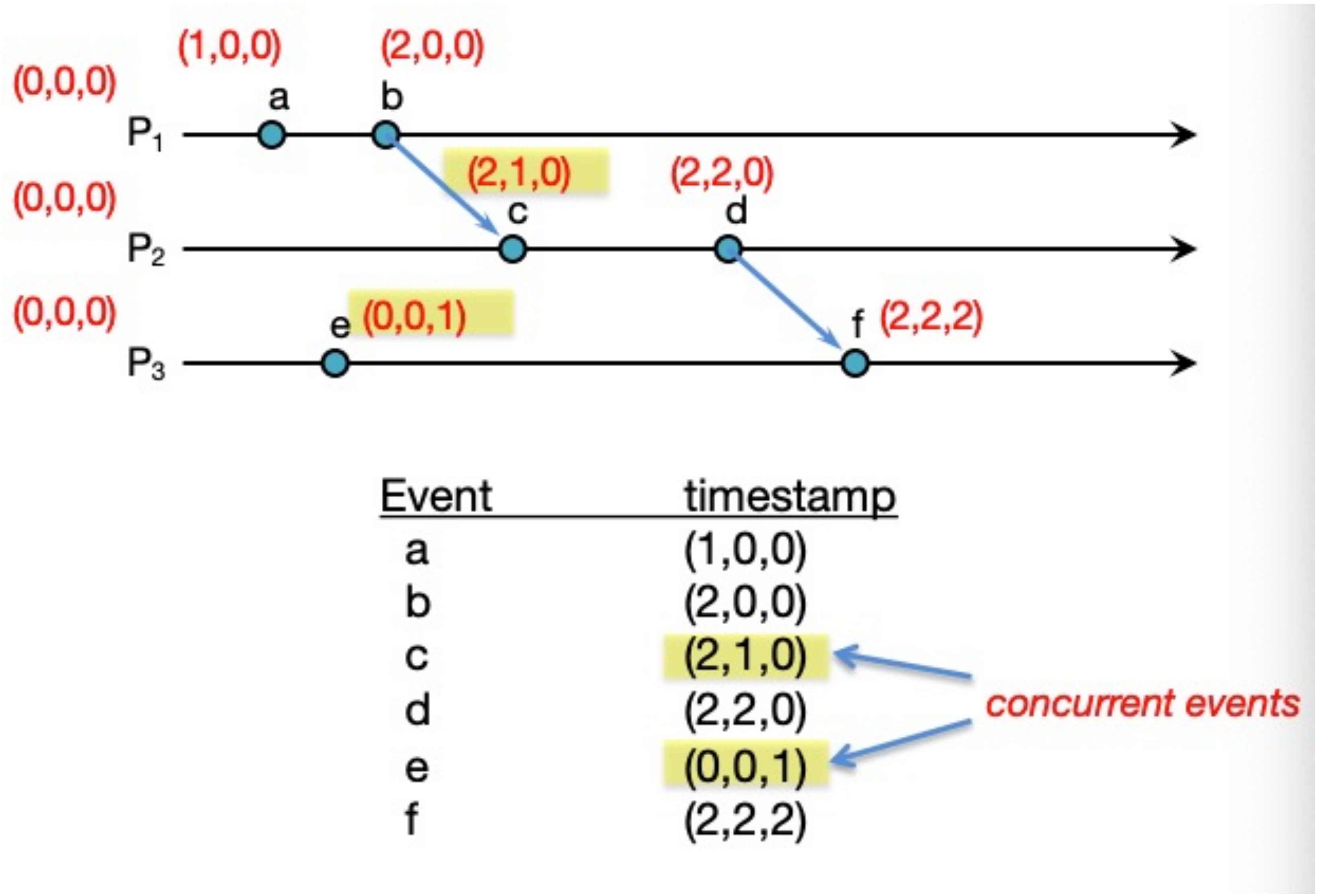
Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)

concurrent events

CLOCK ALGORITHMS

Vector clock algorithm

Example

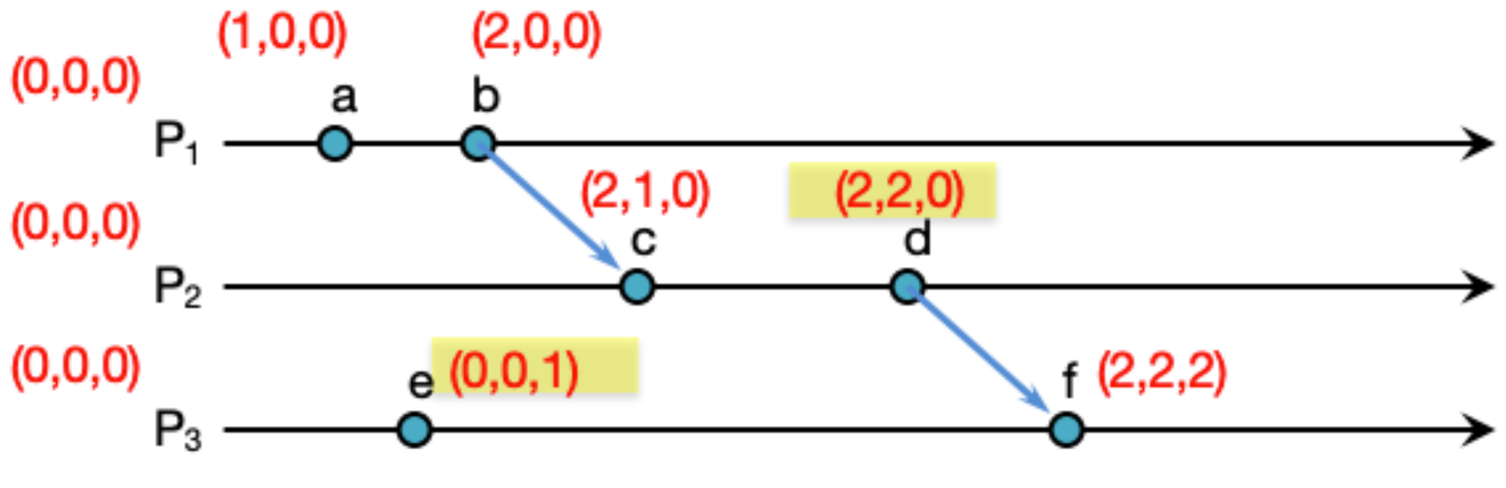


Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)

CLOCK ALGORITHMS

Vector clock algorithm

Example



Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)

concurrent events

SUMMARY

- Vector clocks give us a way of identifying which events are causally related
  - We are guaranteed to get the sequencing correct
  - Problems: Vector size + Larger vector need more comparison time (Space & Time)
- Causality
  - If a -> b then event *a* can affect event *b*
- Concurrency
  - If neither a -> b nor b -> a then one event cannot affect the other
- Partial Ordering
  - Causal events are sequenced
- Total Ordering
  - All events are sequenced