PHYSICAL AND LOGICAL TIME

George Porter
Module 3
Fall 2020

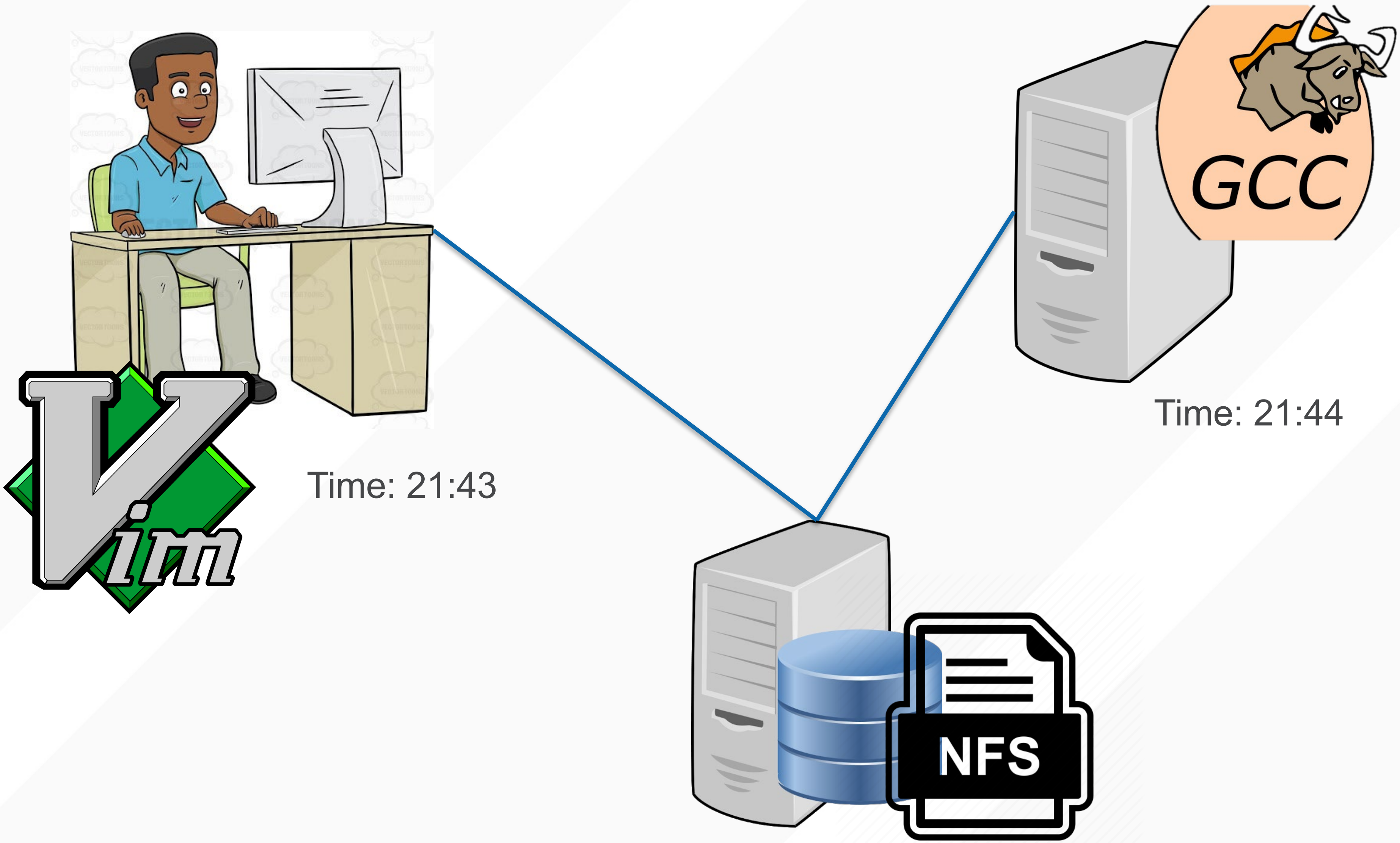
UC San Diego

ATTRIBUTION

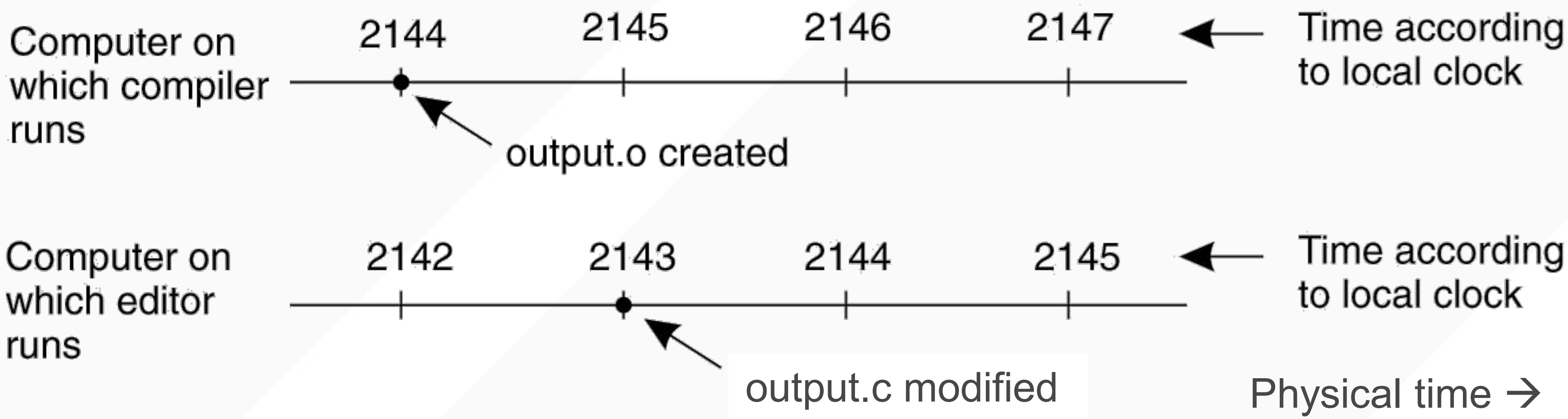
- These slides are released under an Attribution-NonCommercial-ShareAlike 3.0 Unported (CC BY-NC-SA 3.0) Creative Commons license
- These slides incorporate material from:
    - Kyle Jamieson, Princeton University (also under a CC BY-NC-SA 3.0 Creative Commons license)
    - Ghosh, “Distributed Systems”

UC San Diego

MOTIVATION: FIX BUG IN SOURCE CODE; RECOMPILE



A DISTRIBUTED EDIT-COMPILE WORKFLOW



- 2143 < 2144 → *make* **doesn't call compiler**

Lack of time synchronization result – a possible object file mismatch

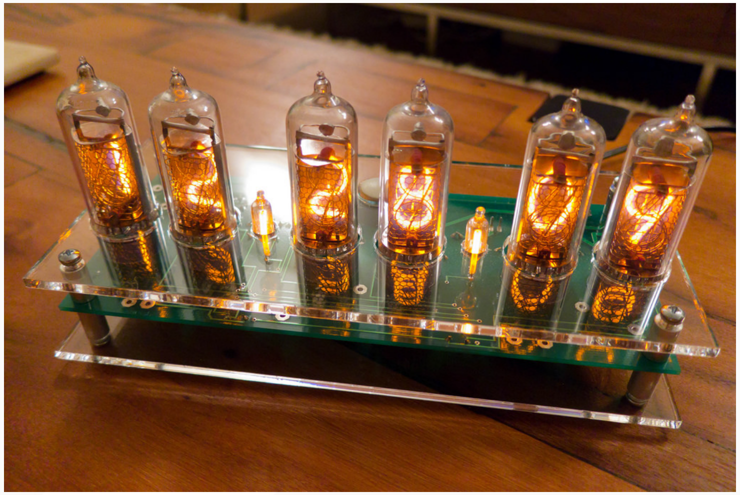
Outline

1. Time sources
2. Physical time
    1. Internal synchronization
    2. External synchronization
    3. NTP
3. Logical time
    1. Lamport clocks
    2. Vector clocks
    3. Totally-ordered multicast

WHAT IS TIME?

- Based on the Sun? 1 second =
  - 1/86,400th of a rotation around the nearest star
- Atomic clocks?
  - 9,192,631,770 orbital transitions of Cesium-133
  - But 3ms off from Solar day: A “leap second”
- International Atomic Time (IAT)
  - Weighted average of 300 atomic clocks (since 1955)
- Universal Coordinated Time (UTC)
  - Aka Greenwich Mean time, or Zulu time (time zones)
  - Broadcast from a radio in Ft. Collins, Colorado
- 2020 specific fact: It’s been 25 years since March

CLOCK SOURCES



CLOCK SOURCES

WHAT MAKES TIME SYNCHRONIZATION HARD?

1. Quartz oscillator **sensitive** to temperature, age, vibration, radiation
   - Accuracy one part per million (**one second** of **clock drift** over **12 days**)

2. Sending time update signals (RF or Internet):
   - ***Asynchronous:*** arbitrary message **delays**
   - ***Best-effort***: messages **don't always arrive**

CLOCK SKEW

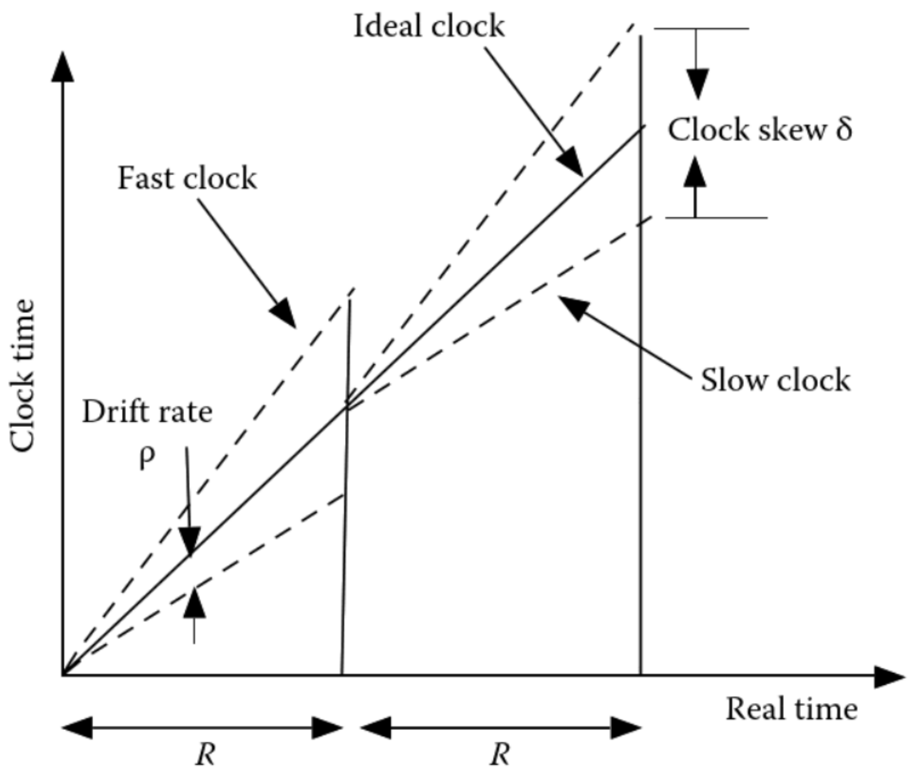


FIGURE 6.5 The cumulative drift between two clocks drifting apart at the rate r is brought closer after every resynchronization interval R.

JUST USE COORDINATED UNIVERSAL TIME?

- UTC is broadcast from radio stations on land and satellite (*e.g.,* the Global Positioning System)
  - Computers with receivers can synchronize their clocks with these timing signals

- Signals from land-based stations are accurate to about 0.1–10 milliseconds

- Signals from GPS are accurate to about one microsecond
  - *Why can't we put GPS receivers on all our computers?*

Outline

1. Time sources
2. Physical time
    1. Internal synchronization
    2. External synchronization
    3. NTP
3. Logical time
    1. Lamport clocks
    2. Vector clocks
    3. Totally-ordered multicast

INTERNAL SYNCHRONIZATION: BERKELEY ALGORITHM

- The ***Berkeley algorithm*** is a distributed algorithm for timekeeping
  - Assumes all machines have **equally-accurate local clocks**
  - Obtains **average** from participating computers and synchronizes clocks to that average
    - Disregard clocks that are severely out of sync

- Non-goal: Synchronizing to the “real” time

BERKELEY ALGORITHM EXAMPLE

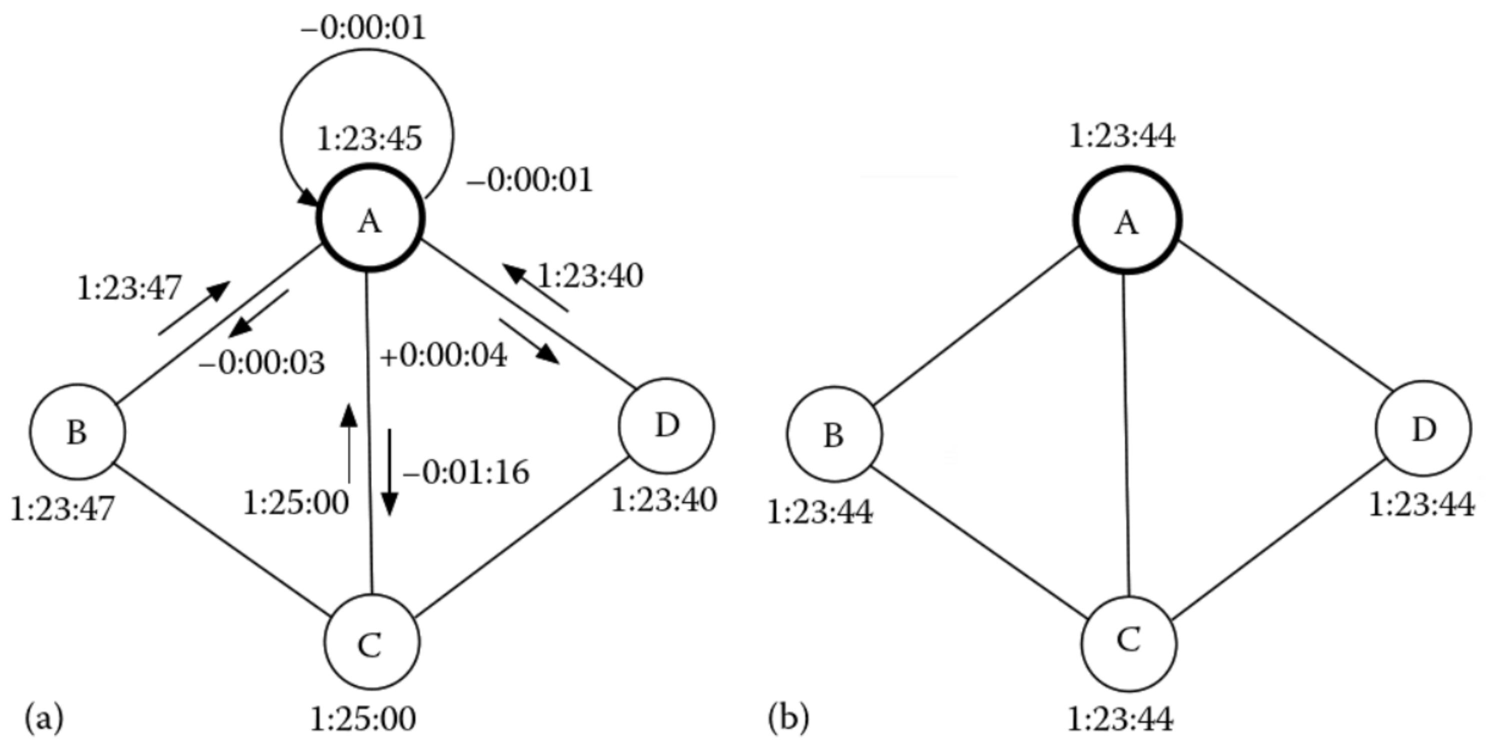


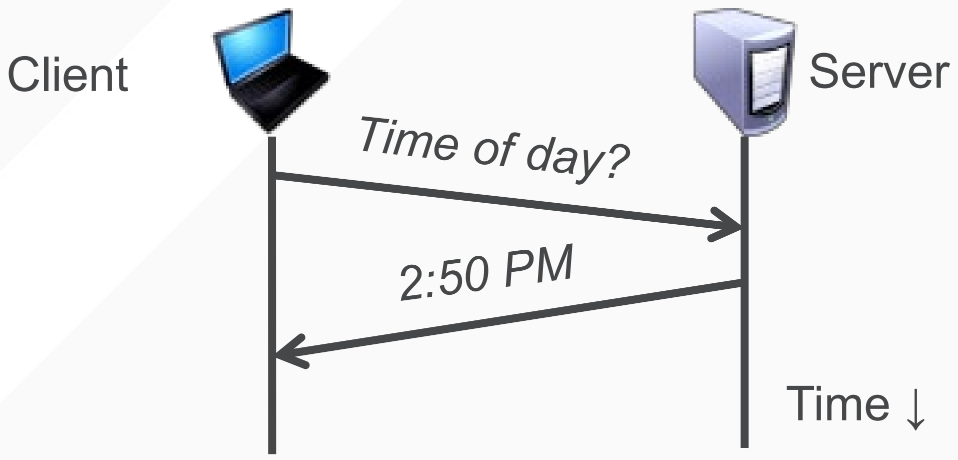
FIGURE 6.6 The readings of the clocks (a) before and (b) after one round of the Berkeley algorithm: A is the leader, and C is an outlier whose value lies outside the permissible limit of 0:00:06 chosen for this system.

Outline

1. Time sources
2. Physical time
    1. Internal synchronization
    2. External synchronization
    3. NTP
3. Logical time
    1. Lamport clocks
    2. Vector clocks
    3. Totally-ordered multicast

SYNCHRONIZATION TO A TIME SERVER

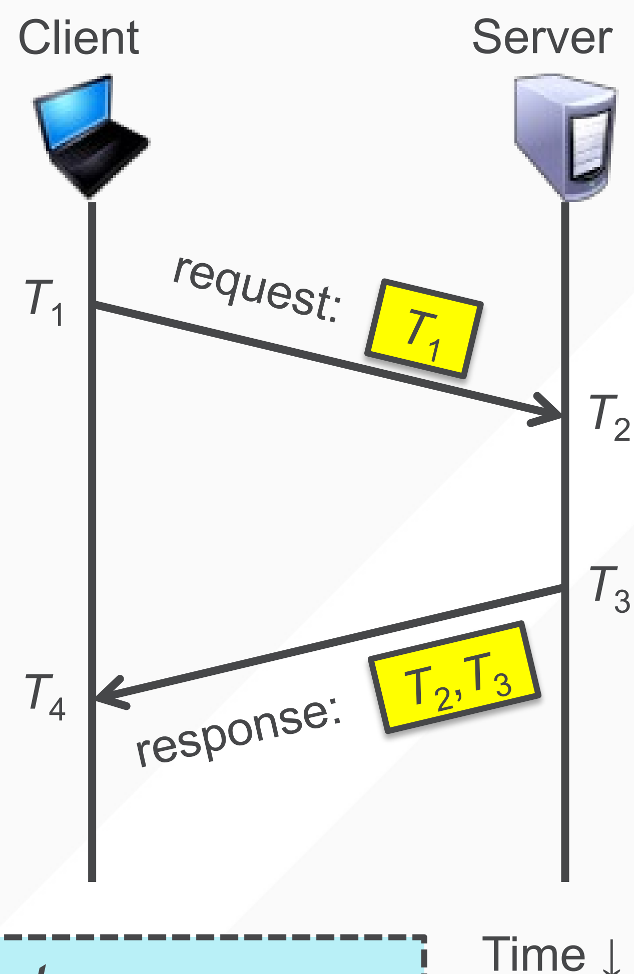
- Suppose a server with an accurate clock (*e.g.*, GPS-disciplined crystal oscillator)
  - Could simply issue an RPC to obtain the time:



- But this doesn't account for network latency
  - **Message delays** will have **outdated** server's answer

CRISTIAN'S ALGORITHM: OUTLINE

1. Client sends a ***request*** packet, timestamped with its local clock T_1
2. Server timestamps its receipt of the request T_2 with its local clock
3. Server sends a ***response*** packet with its local clock T_3 and T_2
4. Client locally timestamps its receipt of the server's response T_4



How the client can use these timestamps to synchronize its local clock to the server's local clock?

CRISTIAN'S ALGORITHM: OFFSET SAMPLE CALCULATION

Goal: Client sets clock ← $T_3 + \delta_{resp}$

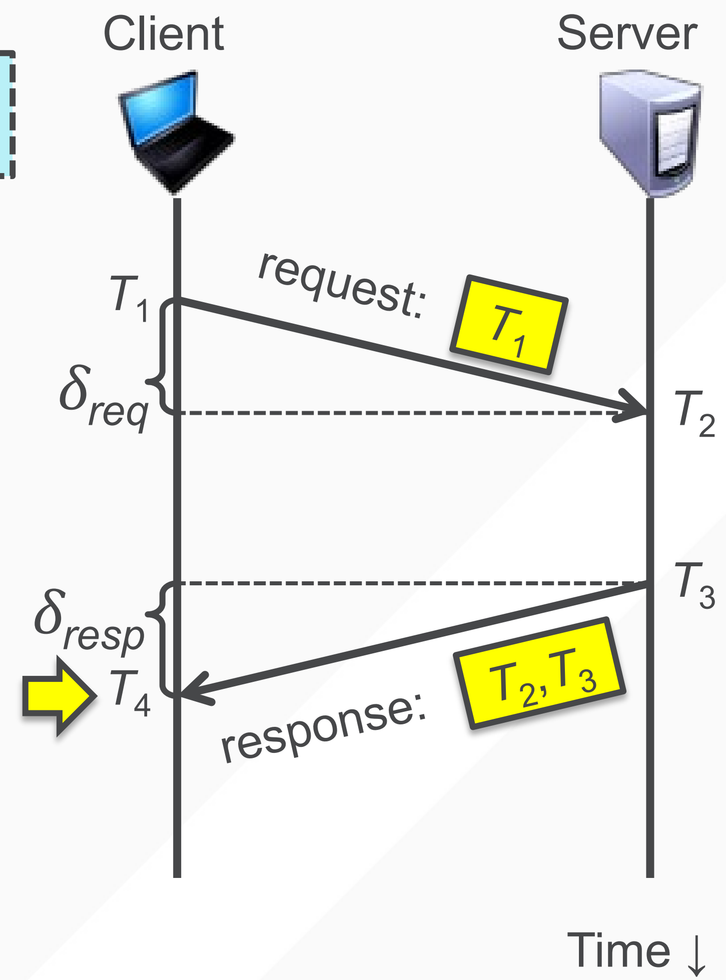
- Client samples ***round trip time***

$$\delta = \delta_{req} + \delta_{resp} = (T_4 - T_1) - (T_3 - T_2)$$

- **But client knows δ, not δ_{resp}**

Assume: $\delta_{req} \approx \delta_{resp}$

Client sets clock ← $T_3 + ½\delta$



Outline

1. Time sources
2. Physical time
    1. Internal synchronization
    2. External synchronization
    3. NTP
3. Logical time
    1. Lamport clocks
    2. Vector clocks
    3. Totally-ordered multicast

THE NETWORK TIME PROTOCOL (NTP)

- Enables clients to be accurately synchronized to UTC despite message delays
- Provides **reliable** service
  - Survives lengthy losses of connectivity
  - Communicates over redundant network paths
- Provides an **accurate** service
  - Unlike the Berkeley algorithm, leverages **heterogeneous** accuracy in clocks

NTP: SYSTEM STRUCTURE

- Servers and time sources are arranged in layers (***strata***)
  - Stratum 0: High-precision time sources themselves
    - *e.g.*, atomic clocks, shortwave radio time receivers
  - Stratum 1: NTP servers **directly connected** to Stratum 0
  - Stratum 2: NTP servers that synchronize with Stratum 1
    - Stratum 2 servers are **clients of** Stratum 1 servers
  - Stratum 3: NTP servers that synchronize with Stratum 2
    - Stratum 3 servers are **clients of** Stratum 2 servers

- Users' computers synchronize with Stratum 3 servers

NTP HIEREARCHY

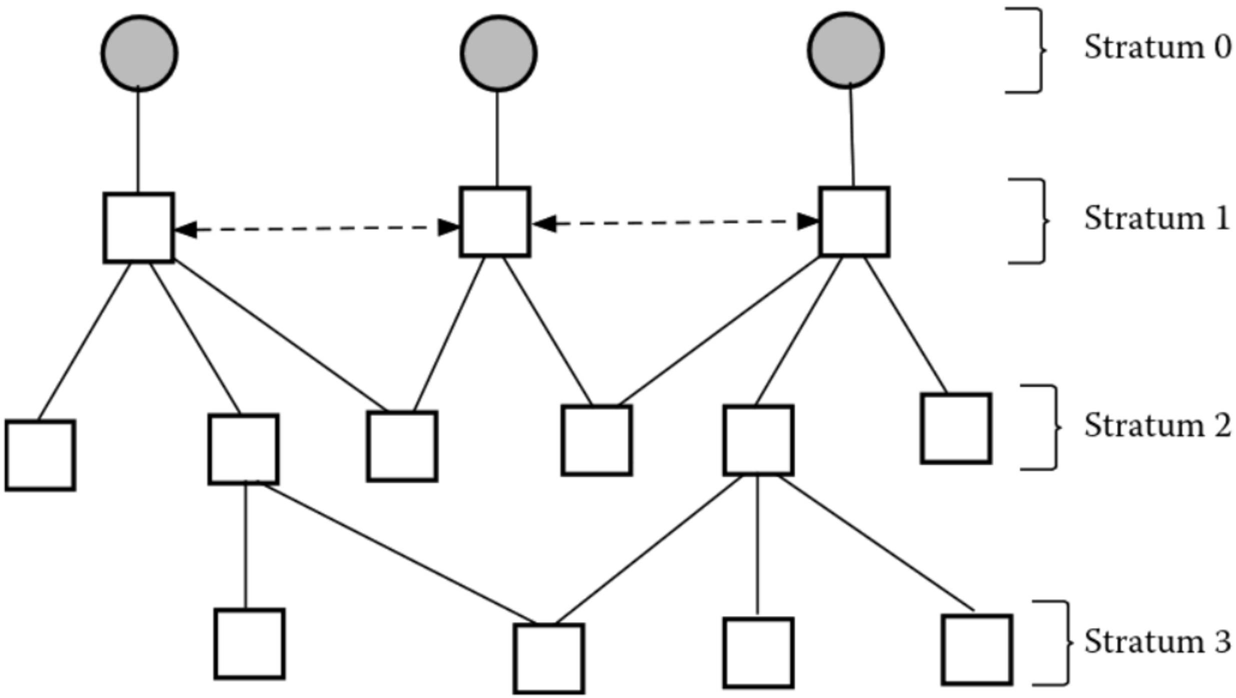


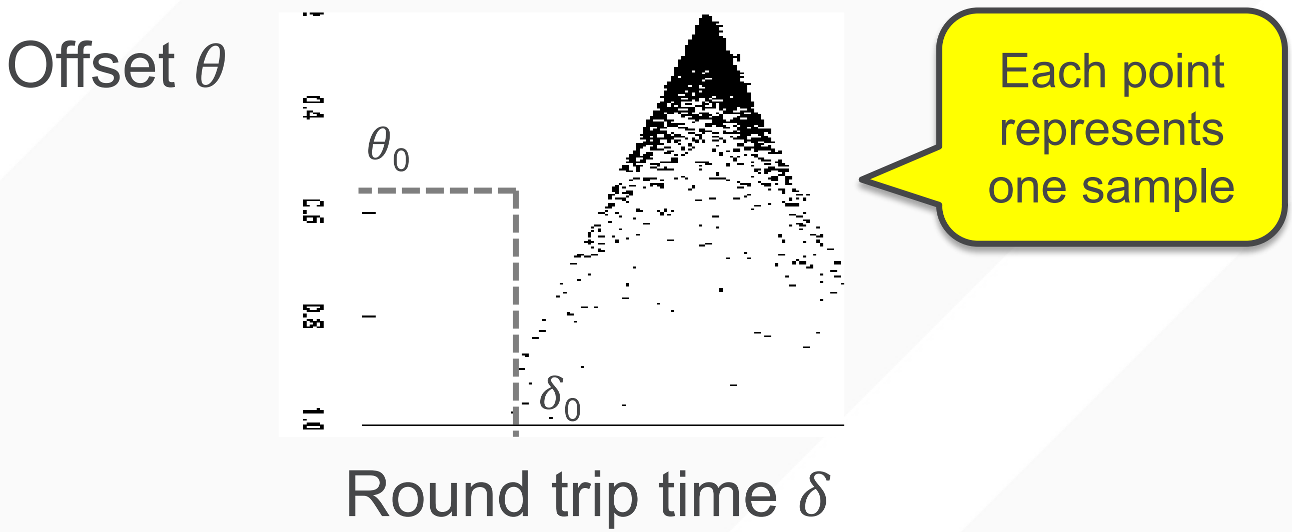
FIGURE 6.9 A network of time servers used in NTP. The top-level devices (stratum 0) have the highest precision.

NTP OPERATION: SERVER SELECTION

- Messages between an NTP client and server are exchanged in pairs: request and response
  - Use Cristian's algorithm
- For i^{th} message exchange with a particular server, calculate:
  1. **Clock offset** θ_i from client to server
  2. **Round trip time** δ_i between client and server
- Over last eight exchanges with server k, the client computes its **dispersion** $\sigma_k = \max_i \delta_i - \min_i \delta_i$
  - Client uses the server with **minimum dispersion**
  - Outliers are discarded

NTP OPERATION : CLOCK OFFSET CALCULATION

- Client tracks minimum round trip time and associated offset over the last eight message exchanges (δ_0, θ_0)
  - θ_0 is the best estimate of offset: client adjusts its clock by θ_0 to **synchronize to server**



NTP OPERATION: HOW TO CHANGE TIME

- Can’t just change time: Don’t want time to **run backwards**
  - Recall the make example
- Instead, change the **update rate** for the clock
  - Changes time in a more gradual fashion
  - Prevents inconsistent local timestamps

LOGICAL TIME

UC San Diego

TIME SYNC CASE STUDY: GOOGLE SPANNER

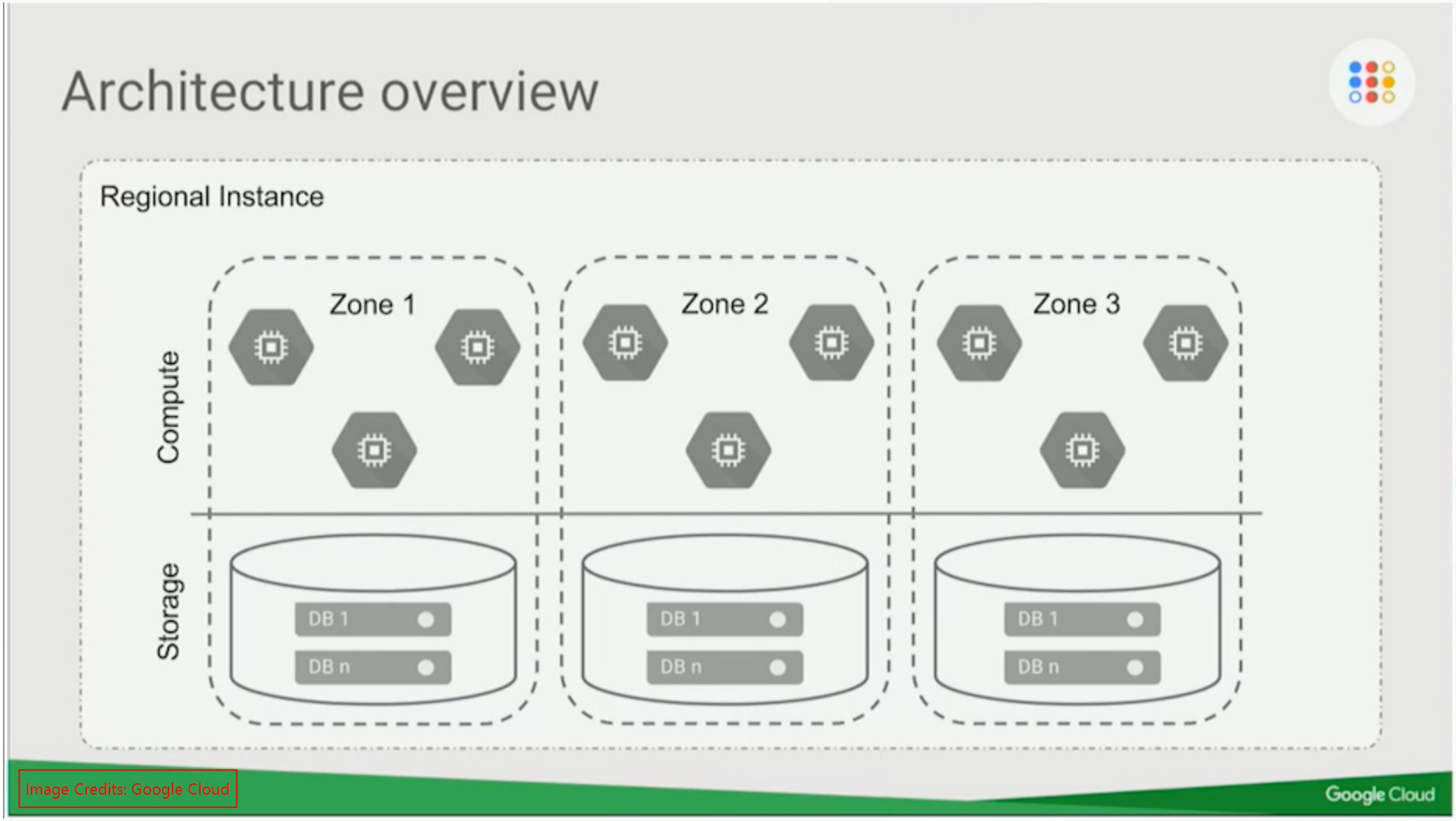
Cloud Spanner

- Google’s distributed database
- Spans regions, continents, etc.
- Consistent results within a certain time period

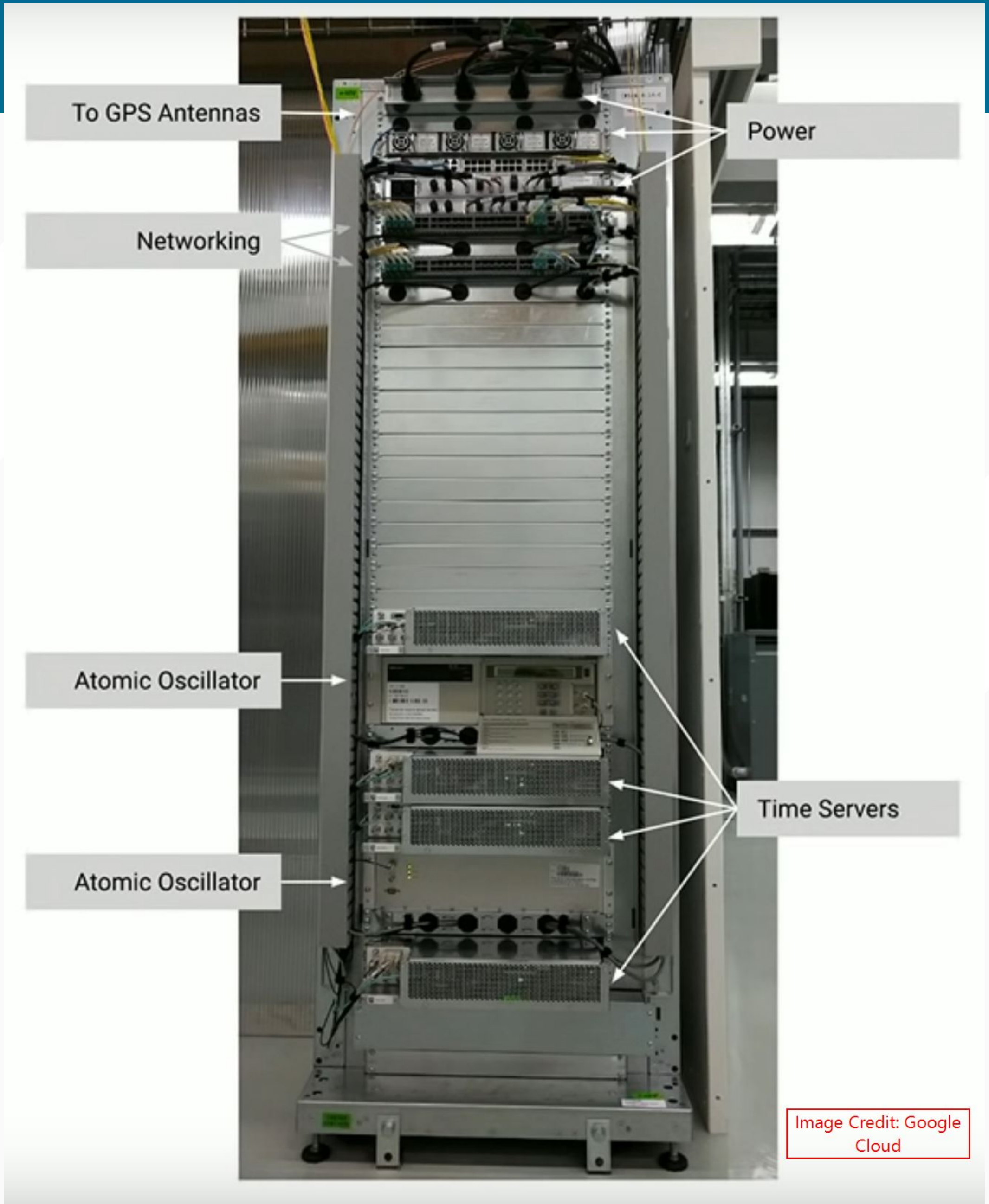
REMEMBER READ/WRITE LEASES?

- Need to block reads while a client has a write lease, lowering performance
- But what if we kept every version of a file/variable, associated with when the updates occurred?
  - The read requests could be timestamped, and could read the "old" values
  - Called External Consistency
- If you're interested in this topic, check out CSE 223b
- Take-away: Synchronized clocks are super useful!

SPANNER ARCHITECTURE

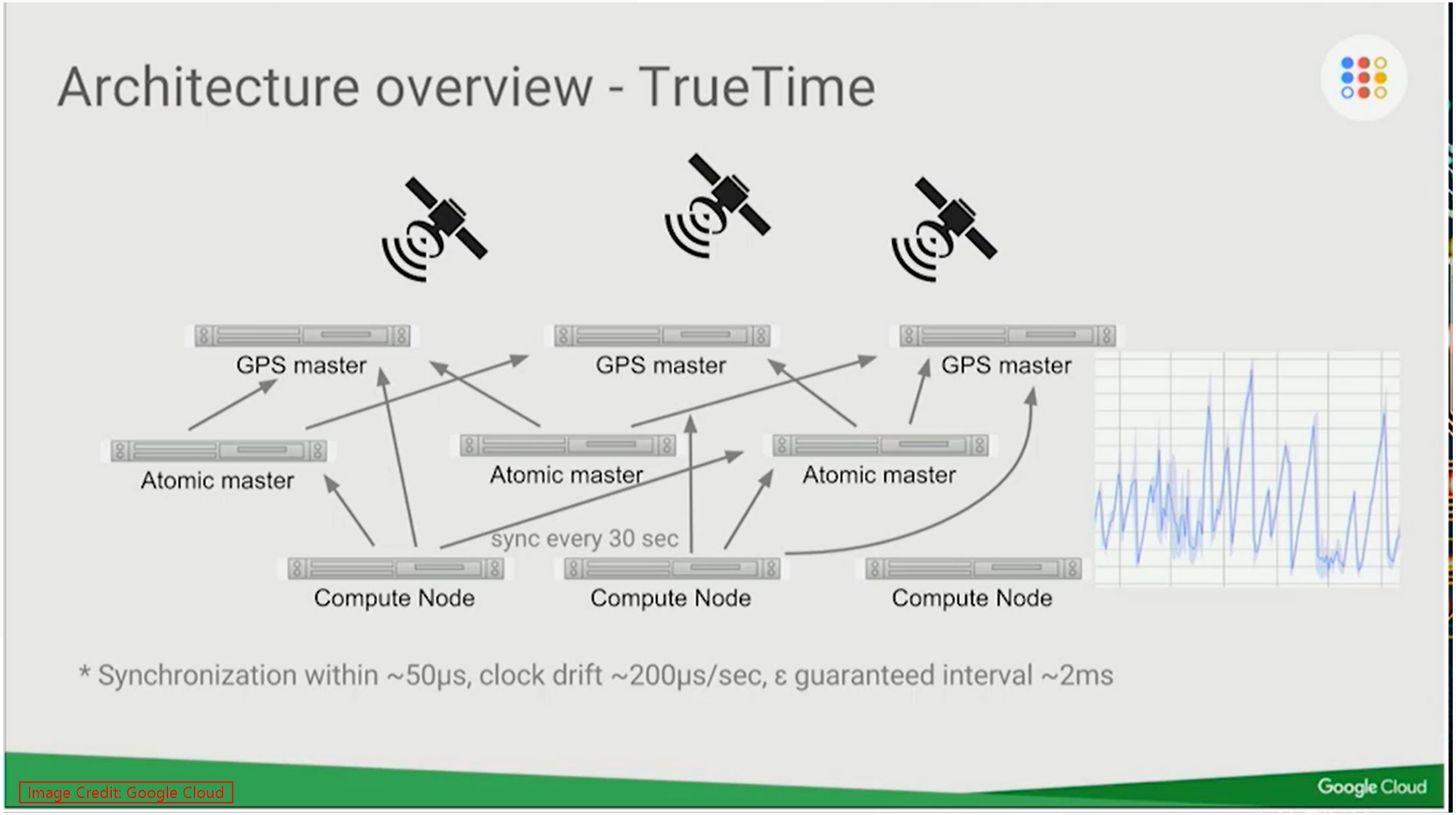


TRUETIME™



To GPS Antennas
Power
Networking
Atomic Oscillator
Time Servers
Atomic Oscillator
Image Credit: Google Cloud

TRUETIME ARCHITECTURE



Outline

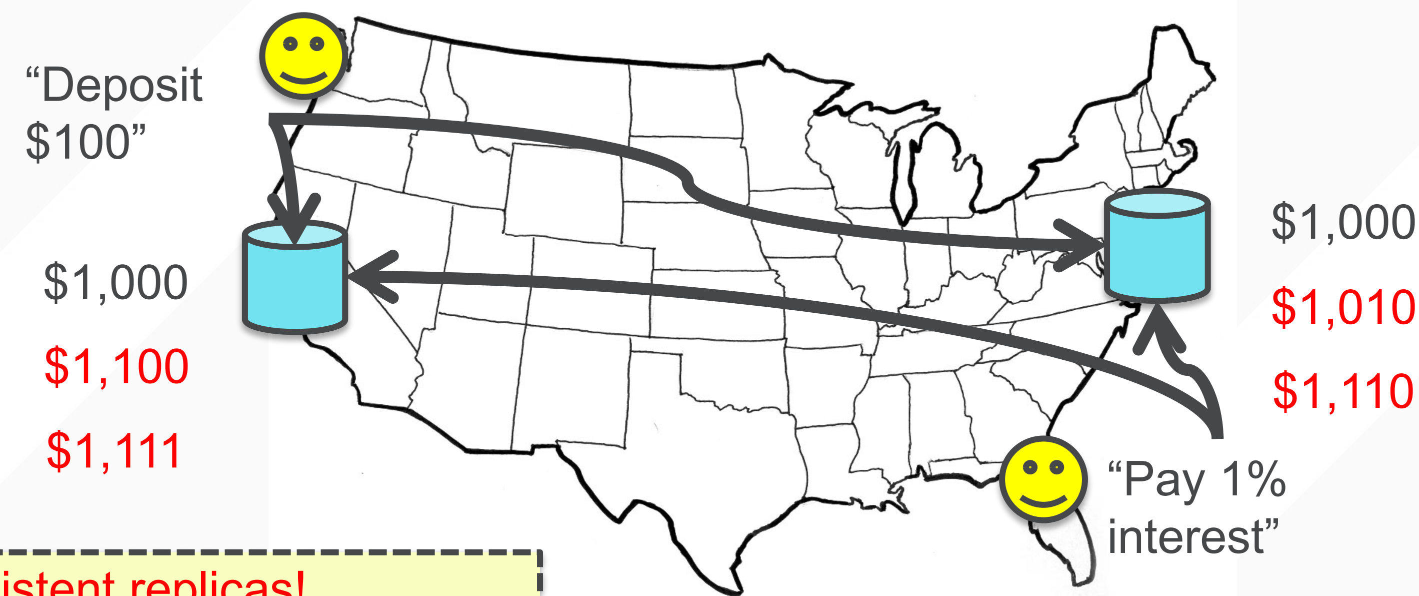
1. Time sources
2. Physical time
    1. Internal synchronization
    2. External synchronization
    3. NTP
3. Logical time
    1. Lamport clocks
    2. Vector clocks
    3. Totally-ordered multicast

MOTIVATION: MULTI-SITE DATABASE REPLICATION

- A New York-based bank wants to make its transaction ledger database **resilient** to **whole-site failures**
- **Replicate** the database, keep one copy in sf, one in nyc

MOTIVATION: MULTI-SITE DATABASE REPLICATION

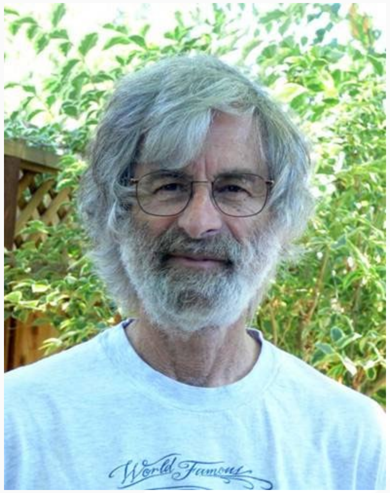
- **Replicate** the database, keep one copy in sf, one in nyc
  - Client sends **query** to the **nearest** copy
  - Client sends **update to both** copies



Inconsistent replicas!
Updates should have been performed **in the same order** at each copy

IDEA: *LOGICAL* CLOCKS

- Landmark 1978 paper by Leslie Lamport
- **Insight:** only the **events themselves** matter

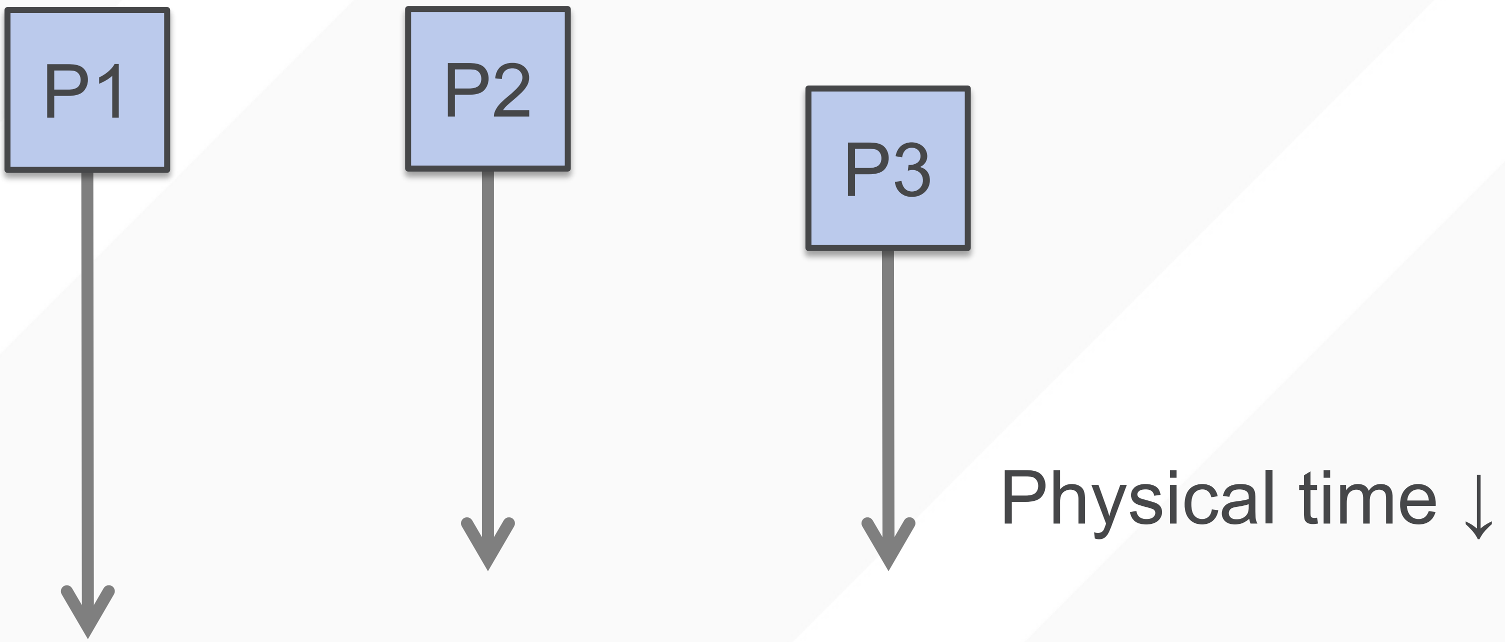


Idea: Disregard the precise clock time

Instead, capture just a "happens before" relationship between a pair of events

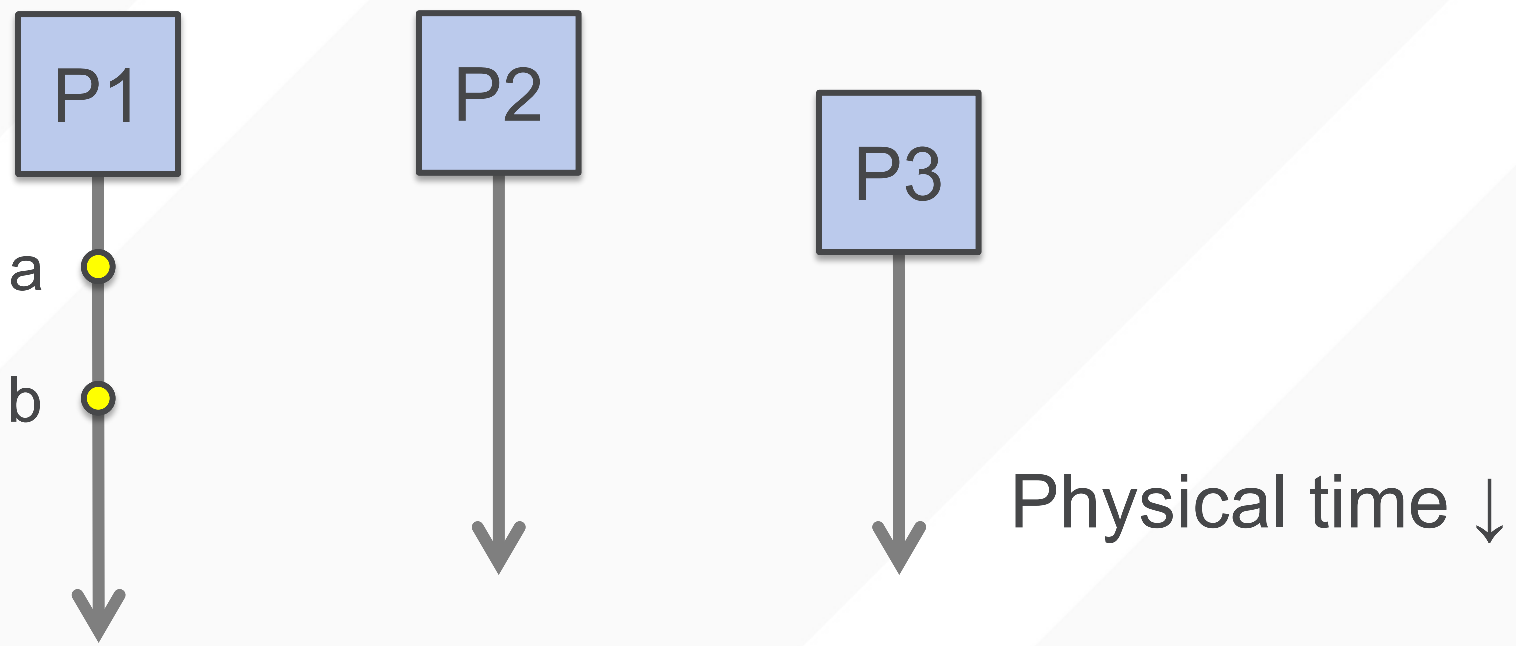
DEFINING "HAPPENS-BEFORE"

- Consider three processes: **P1**, **P2**, and **P3**
- **Notation:** Event **a** ***happens before*** event **b (a → b)**



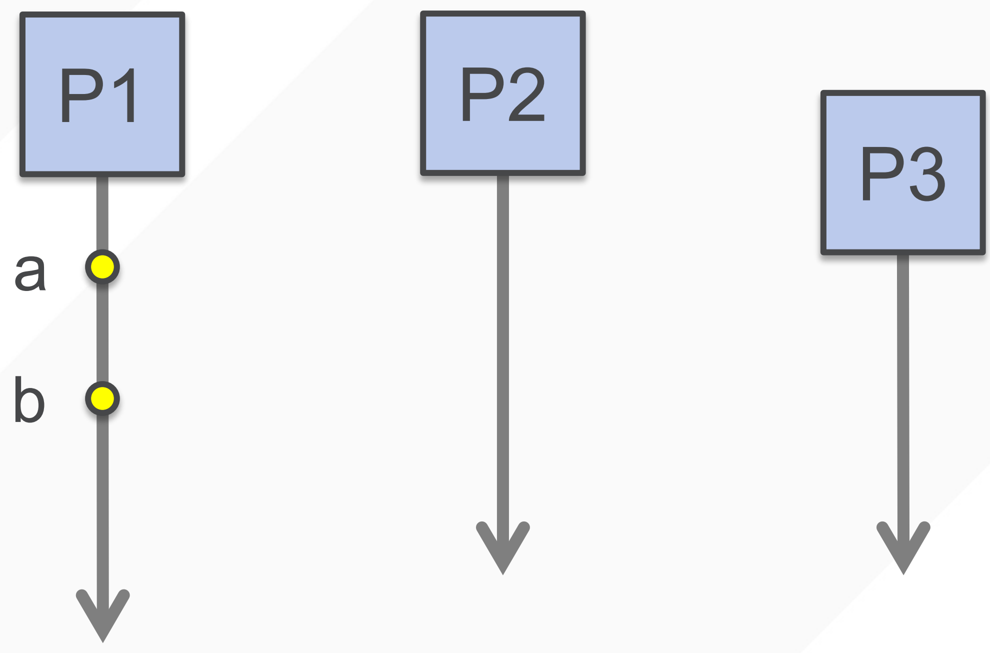
DEFINING "HAPPENS-BEFORE"

1. Can observe event order at a single process



DEFINING "HAPPENS-BEFORE"

1. If **same process** and **a** occurs before **b**, then **a** → **b**

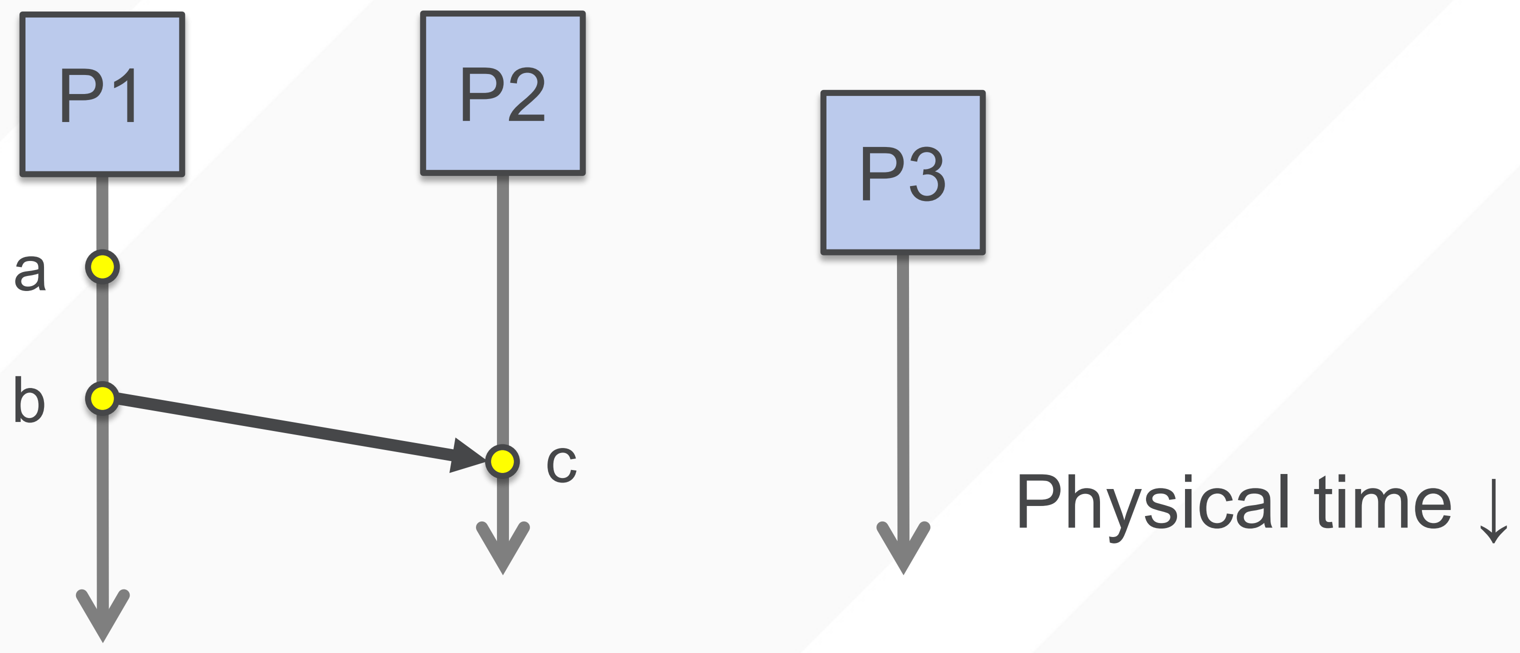


Physical time ↓

DEFINING "HAPPENS-BEFORE"

1. If **same process** and **a** occurs before **b**, then **a** → **b**

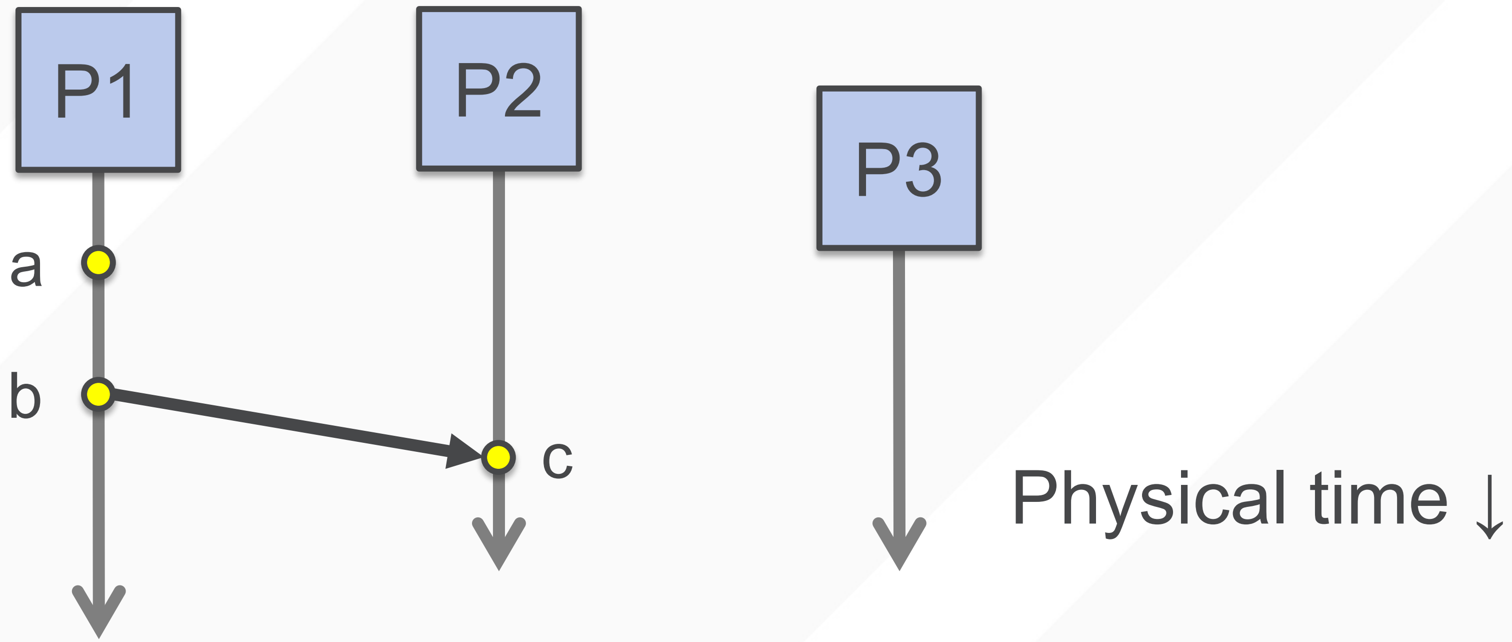
2. Can observe ordering when processes communicate



DEFINING "HAPPENS-BEFORE"

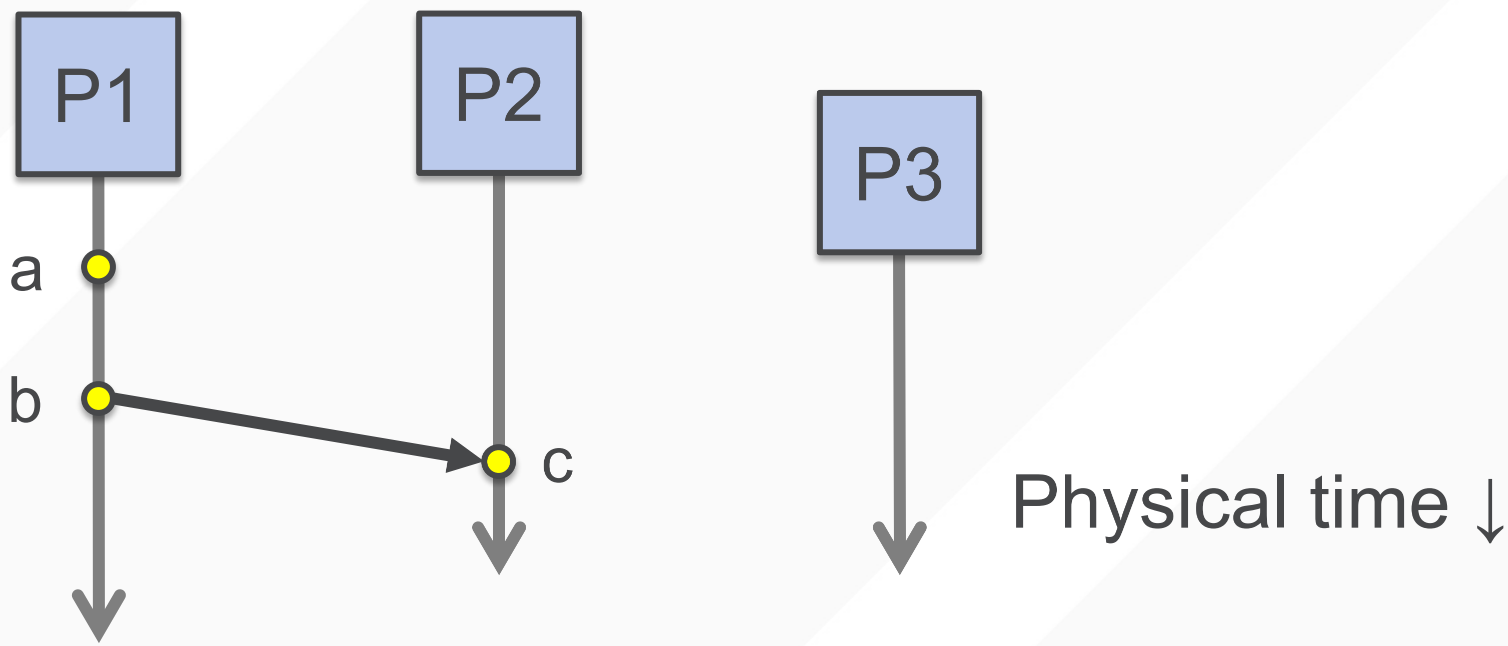
1. If **same process** and **a** occurs before **b**, then **a** → **b**

2. If **c** is a message receipt of **b**, then **b** → **c**



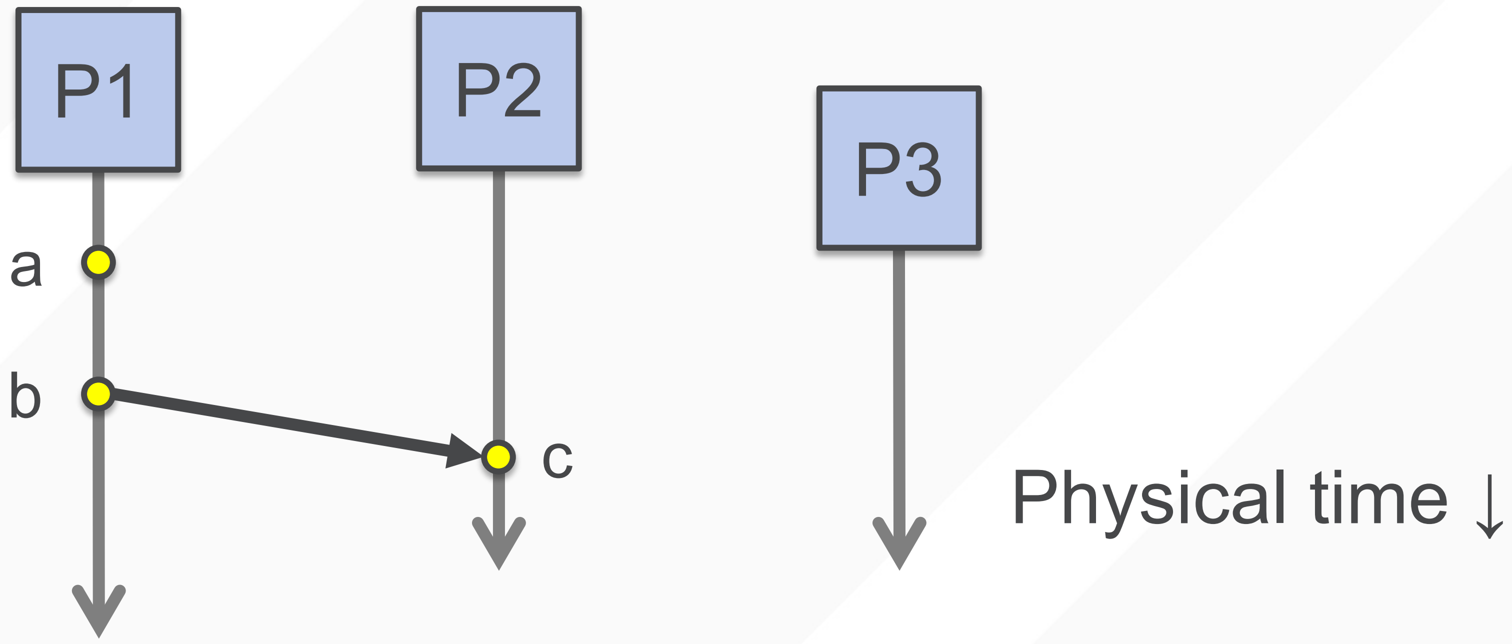
DEFINING "HAPPENS-BEFORE"

1. If **same process** and **a** occurs before **b**, then **a** → **b**
2. If **c** is a message receipt of **b**, then **b** → **c**
3. Can observe ordering transitively



TRANSITIVE "HAPPENS-BEFORE"

1. If **same process** and **a** occurs before **b**, then **a** → **b**
2. If **c** is a message receipt of **b**, then **b** → **c**
3. If **a** → **b** and **b** → **c**, then **a** → **c**



CONCURRENT EVENTS

- We seek a ***clock time C*(a)** for every event **a**

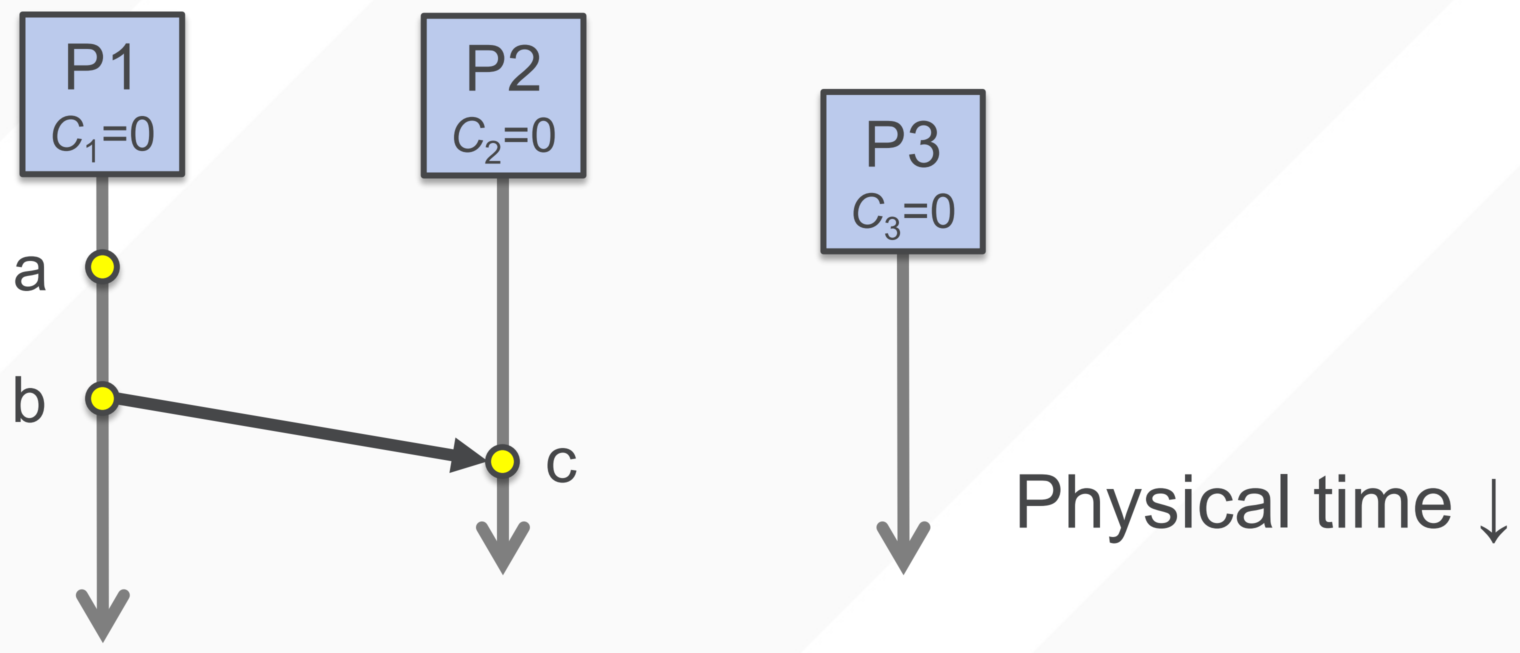
Plan: Tag events with clock times; use clock times to make distributed system correct

- Clock condition: If **a** → **b**, then *C*(**a**) < *C*(**b**)

THE LAMPORT CLOCK ALGORITHM

- Each process **P**$_i$ maintains a local clock C_i

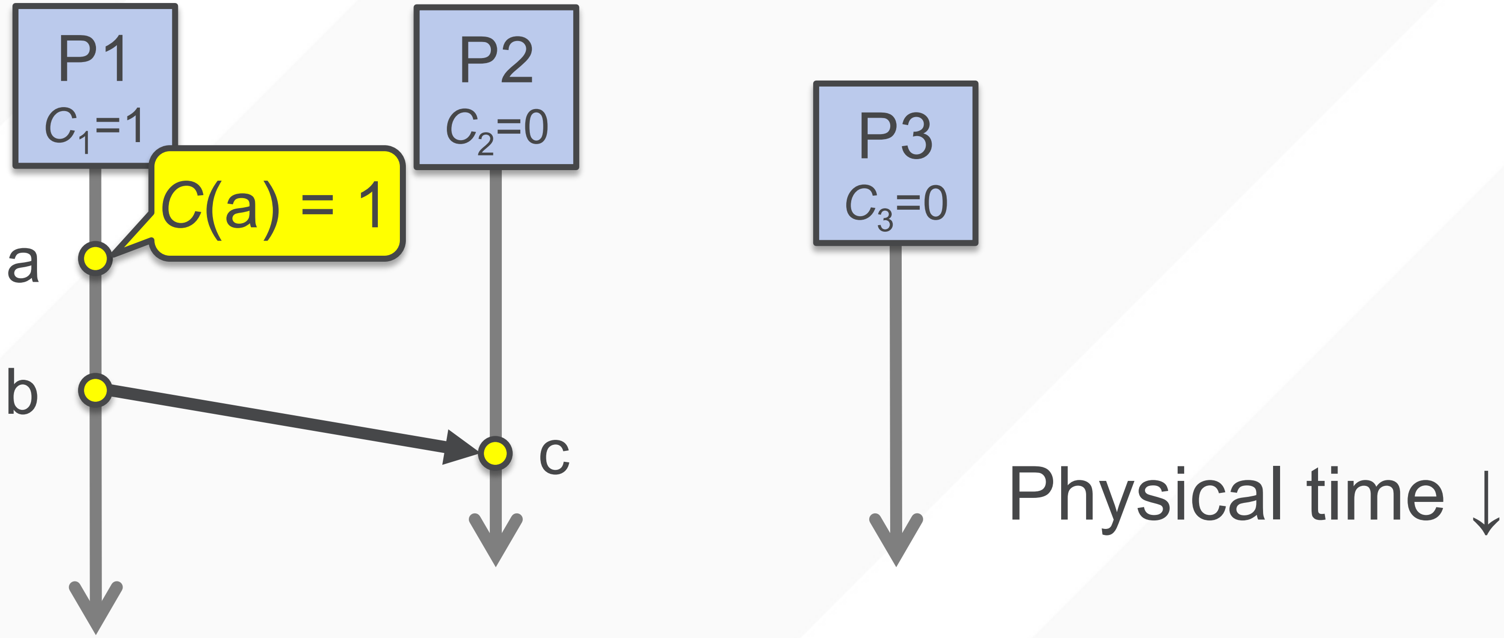
1. Before executing an event, $C_i \leftarrow C_i + 1$



THE LAMPORT CLOCK ALGORITHM

1. Before executing an event **a**, $C_i \leftarrow C_i + 1$:

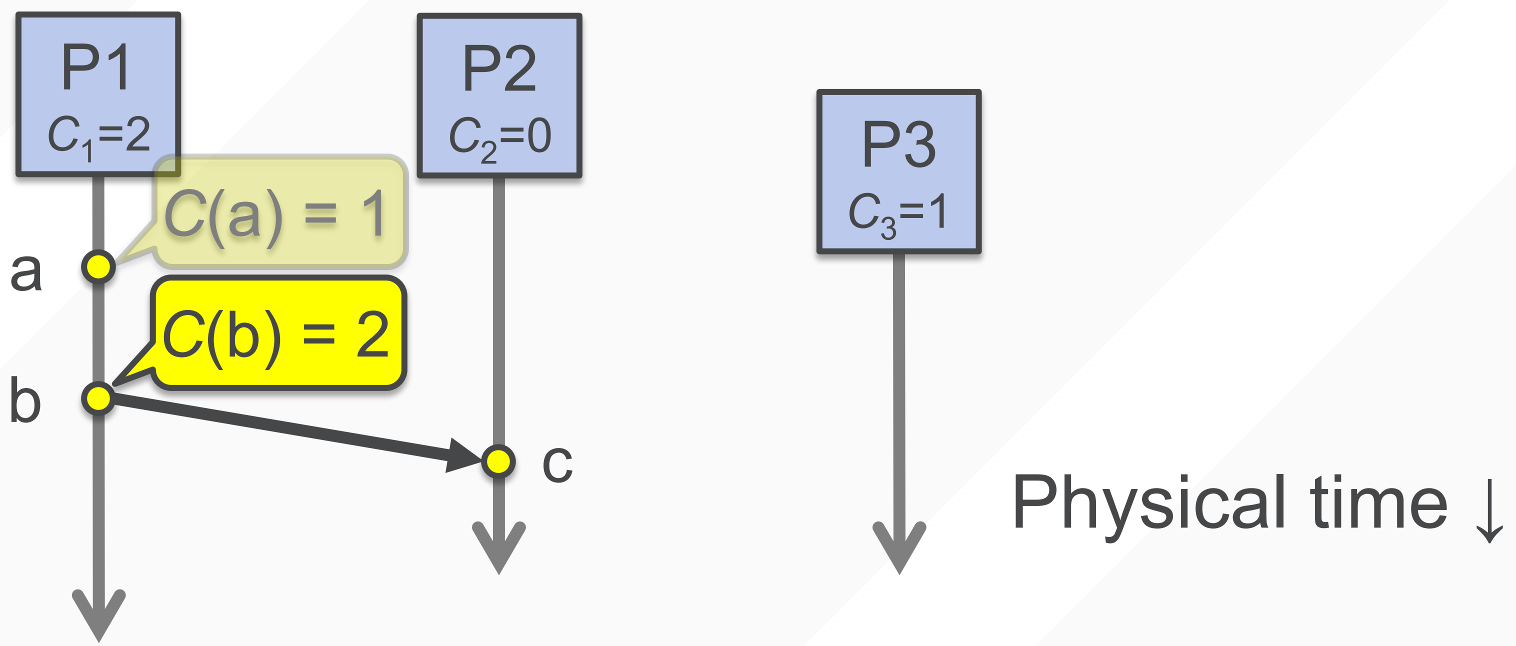
    - Set event time $C(\mathbf{a}) \leftarrow C_i$



THE LAMPORT CLOCK ALGORITHM

1. Before executing an event **b**, $\boldsymbol{C_i} \leftarrow \boldsymbol{C_i} + 1$:

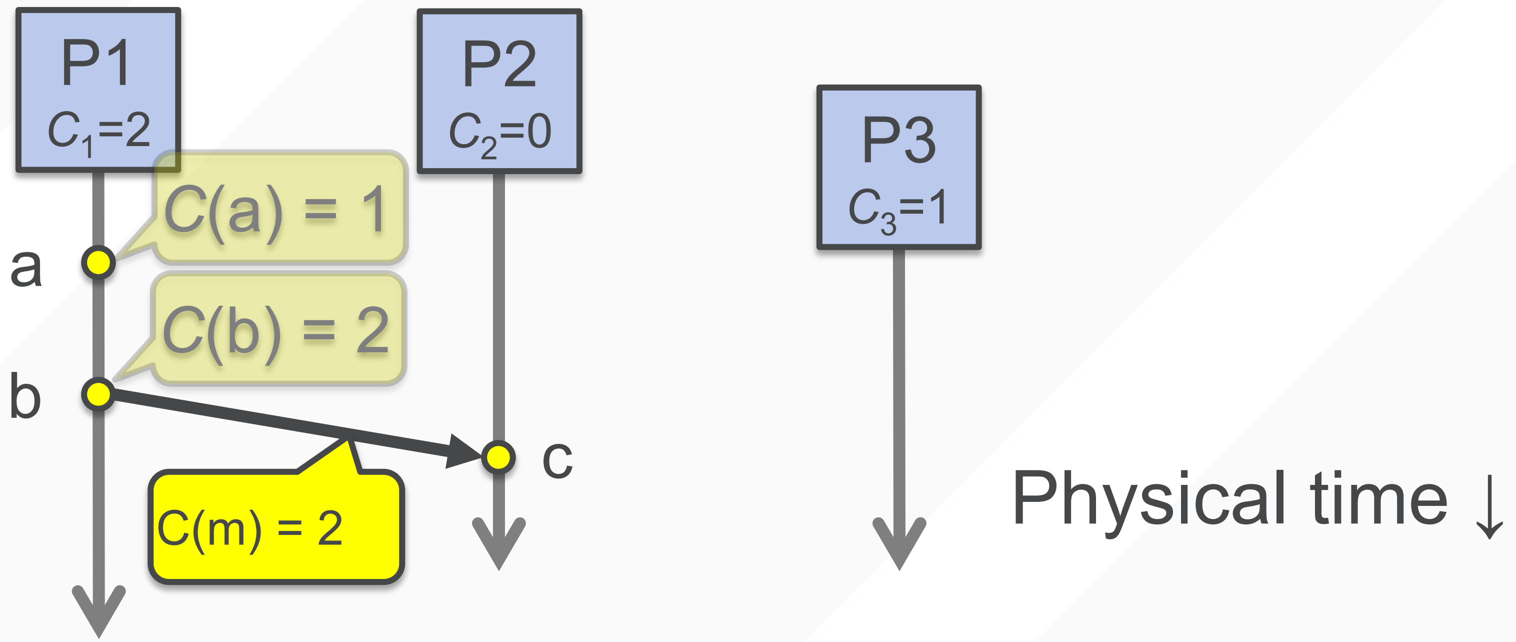
   - Set event time C(**b**) $\leftarrow C_i$



THE LAMPORT CLOCK ALGORITHM

1. Before executing an event **b**, $C_i \leftarrow C_i + 1$

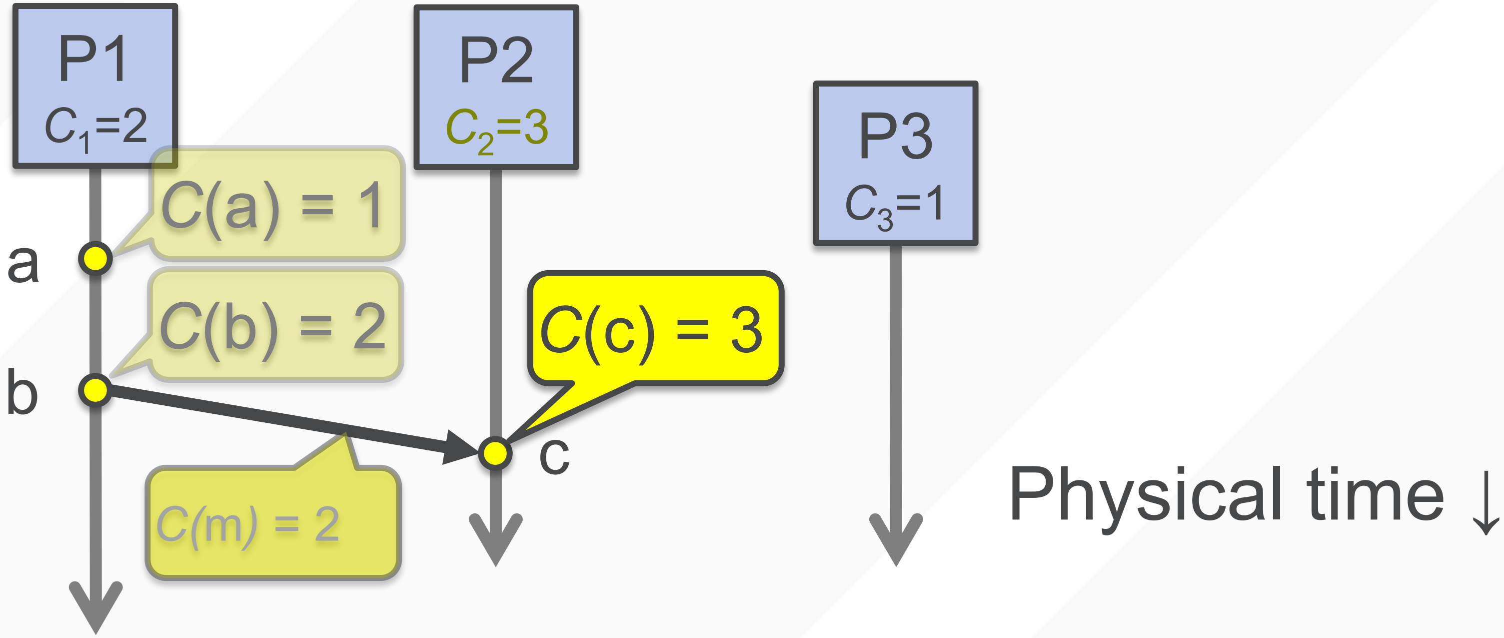
2. Send the local clock in the message **m**



THE LAMPORT CLOCK ALGORITHM

3. On process $\mathbf{P}_j$ receiving a message **m**:

- Set C_j **and** receive event time $C(\mathbf{c}) \leftarrow 1 + \mathbf{max}\{\, C_j, C(\mathbf{m}) \,\}$



ORDERING ALL EVENTS

- **Break ties** by **appending the process number** to each event:
  1. Process $\mathbf{P}_i$ timestamps event **e** with $C_i(\mathbf{e}).i$
  2. $C(\mathbf{a}).i < C(\mathbf{b}).j$ when:
     - $C(\mathbf{a}) < C(\mathbf{b})$, **or** $C(\mathbf{a}) = C(\mathbf{b})$ and $i < j$

- Now, for any two events **a** and **b**:
  - $C(\mathbf{a}) < C(\mathbf{b})$ or $C(\mathbf{b}) < C(\mathbf{a})$

Outline

1. Time sources
2. Physical time
    1. Internal synchronization
    2. External synchronization
    3. NTP
3. Logical time
    1. Lamport clocks
    2. Vector clocks
    3. Totally-ordered multicast

VECTOR CLOCK (VC)

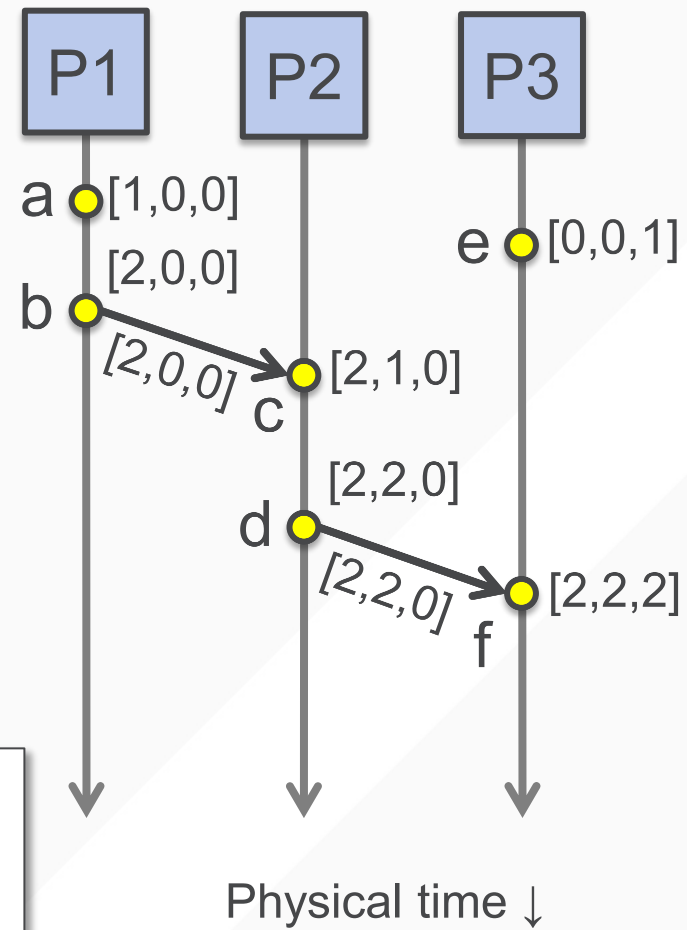
- Label each event **e** with a vector $V(\mathbf{e}) = [c_1, c_2 \ldots, c_n]$
  - c_i is a count of events in process i that causally precede **e**
- Initially, all vectors are [0, 0, …, 0]

- Two **update rules:**

1. For each **local event** on process i, increment local entry c_i
2. If process j **receives** message with vector $[d_1, d_2, \ldots, d_n]$:
   - Set each local entry $c_k = \max\{c_k, d_k\}$
   - Increment local entry c_j

VECTOR CLOCK: EXAMPLE

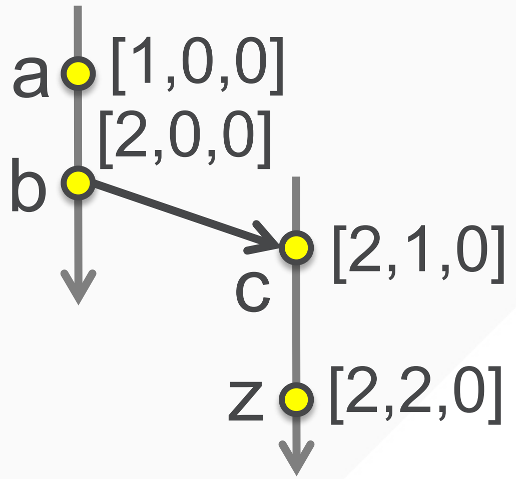
- All counters start at [0, 0, 0]
- Applying local update rule
- Applying message rule
  - Local vector clock piggybacks on inter-process messages



[2,2,2]: Remember we have event e at P3 with timestamp [0,0,1]. D's message gets timestamp [2,2,0], we take max to get [2,2,1] then increment the local entry to get [2,2,2].

VECTOR CLOCKS CAN ESTABLISH CAUSALITY

- Rule for comparing vector clocks:
  - V(**a**) = V(**b**) when $\mathbf{a}_k = \mathbf{b}_k$ for all k
  - V(**a**) < V(**b**) when $\mathbf{a}_k \leq \mathbf{b}_k$ for all k and V(**a**) ≠ V(**b**)
- **Concurrency:** ***a* // *b*** if $\mathbf{a}_i < \mathbf{b}_i$ and $\mathbf{a}_j > \mathbf{b}_j$, some i, j
- V(a) < V(z) **when** there is a chain of events linked by → between a and z



PHYSICAL AND LOGICAL TIME

Two events **a, z**

Lamport clocks: C(a) < C(z)

Conclusion: None

Vector clocks: V(a) < V(z)

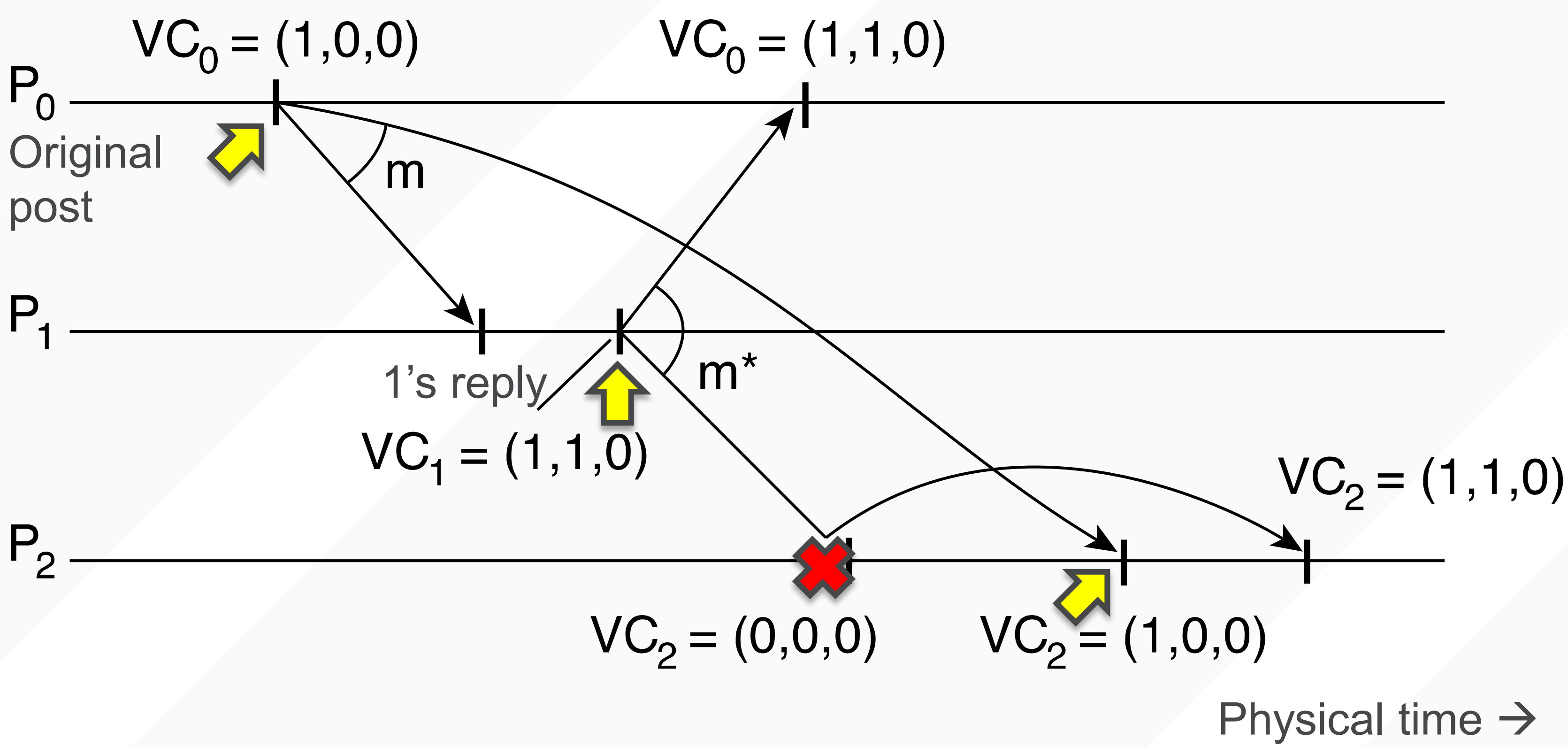
Conclusion: a → … → z

Vector clock timestamps tell us about causal event relationships

VC APPLICATION: CAUSALLY-ORDERED BULLETIN BOARD SYSTEM

- Distributed bulletin board application
  - Each post → multicast of the post to all other users
- **Want:** No user to see a reply before the corresponding original message post
- Deliver message only **after** all messages that **causally precede** it have been delivered
  - Otherwise, the user would see a reply to a message they **could not find**

VC APPLICATION: CAUSALLY-ORDERED BULLETIN BOARD SYSTEM

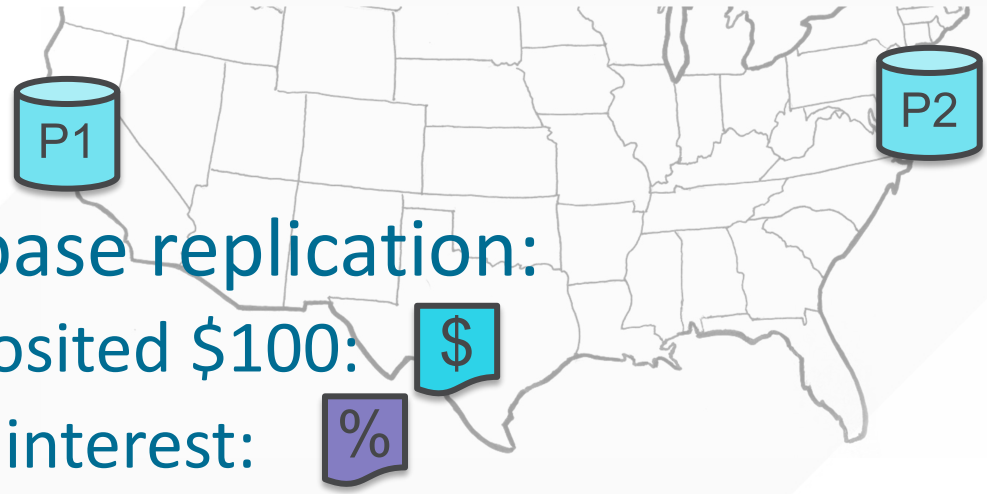


- User 0 posts, user 1 replies to 0's post; user 2 observes

Outline

1. Time sources
2. Physical time
    1. Internal synchronization
    2. External synchronization
    3. NTP
3. Logical time
    1. Lamport clocks
    2. Vector clocks
    3. Totally-ordered multicast

MAKING CONCURRENT UPDATES CONSISTENT



- Recall multi-site database replication:
  - San Francisco (**P1**) deposited \$100:
  - New York (**P2**) paid 1% interest:

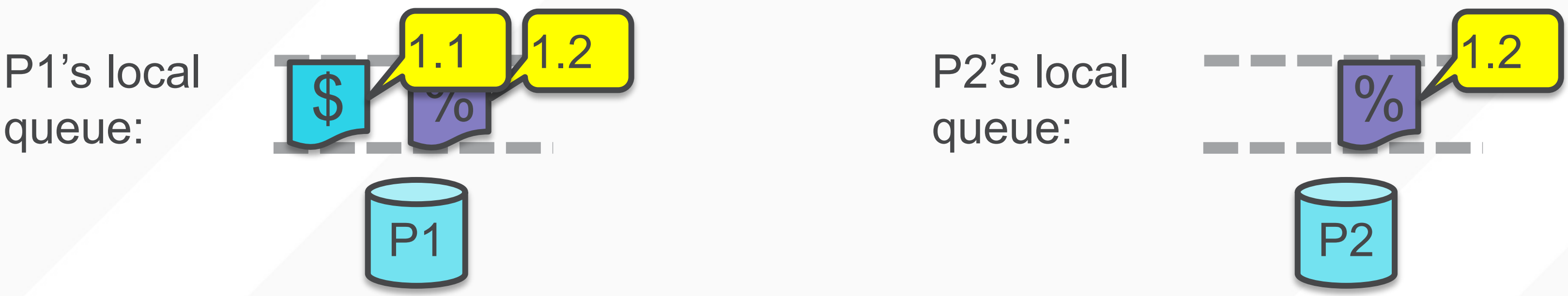
We reached an inconsistent state

Could we design a system that uses Lamport Clock total order to make multi-site updates consistent?

TOTALLY-ORDERED MULTICAST

- Client sends update to **one replica** → **Lamport timestamp C(*x*)**

- **Key idea:** Place events into a **local queue**
  - **Sorted** by increasing C(x)



Goal: All sites apply the updates in (the same) Lamport clock order

TOTALLY-ORDERED MULTICAST

1. On **receiving** an event from **client**, broadcast to others (including yourself)

2. On **receiving or processing** an **event:**

   a) Add it to your local queue

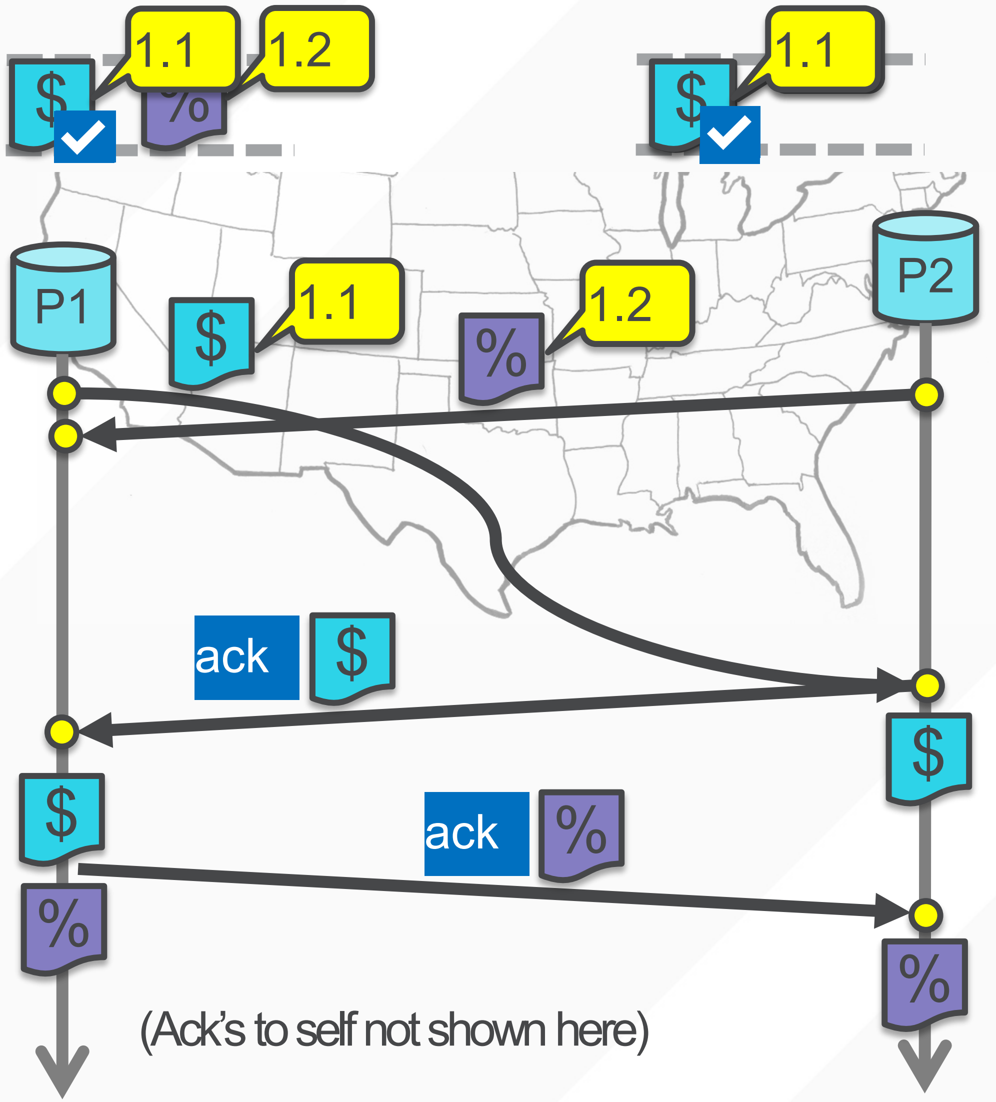
   b) Broadcast an ***acknowledgement message*** to every process (including yourself) **only from head of queue**

3. When you **receive** an **acknowledgement:**

   - Mark corresponding event ***acknowledged*** in your queue

4. **Remove and process** events <u>**everyone**</u> has ack'ed from <u>**head**</u> of queue

TOTALLY-ORDERED MULTICAST



SO, ARE WE DONE?

- *Does totally-ordered multicast solve the problem of multi-site replication in general?*
- Not by a long shot!

1. Our protocol **assumed:**
    - No **node failures**
    - No **message loss**
    - No **message corruption**
2. All to all communication **does not scale**
3. **Waits forever** for message delays **(performance?)**

UC San Diego