Time Synchronization and Logical Clocks

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

CS 240: Computing Systems and Concurrency
Lecture 5

Mootaz Elnozahy

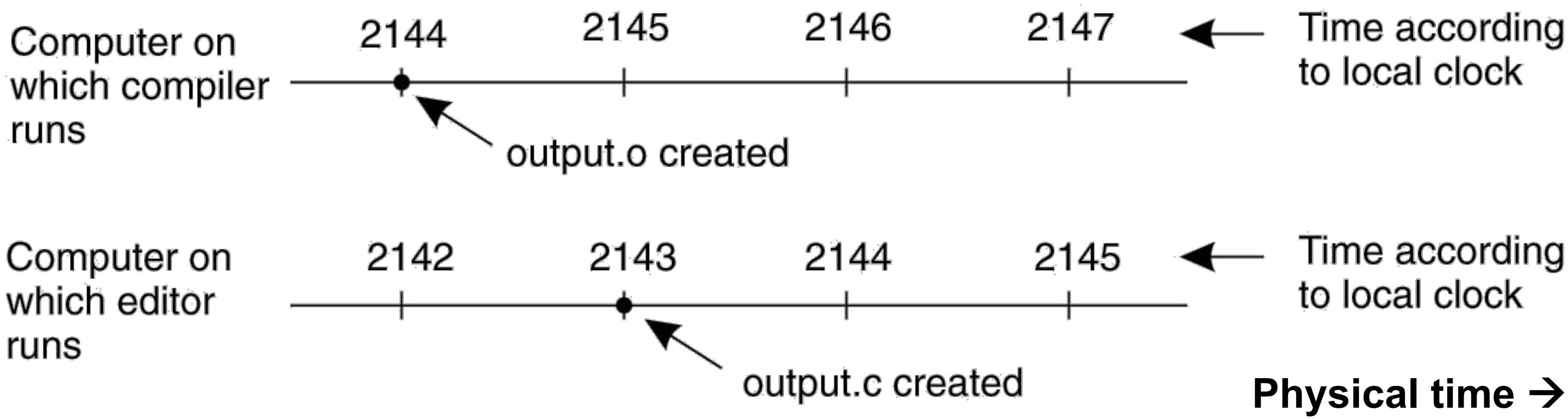
Today

1. **The need for time synchronization**

2. “Wall clock time” synchronization

3. Logical Time

A distributed edit-compile workflow



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

Lack of time synchronization result – a **possible object file mismatch**

What makes time synchronization hard?

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

2. The internet is:
   - ***Asynchronous:*** arbitrary message **delays**
   - ***Best-effort***: messages **don't always arrive**

Today

1. The need for time synchronization

2. **"Wall clock time" synchronization**
   – Cristian's algorithm, Berkeley algorithm, NTP

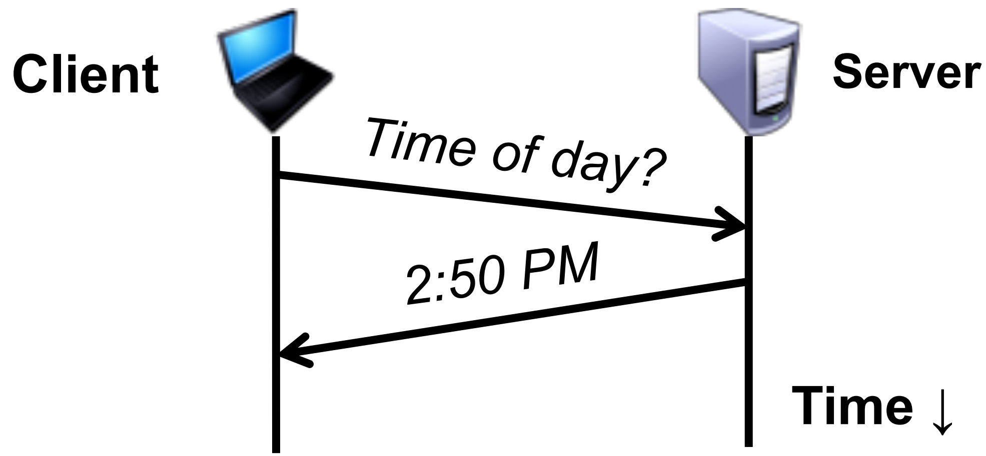
3. Logical Time
   – Lamport clocks
   – Vector clocks

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?*

Synchronization to a time server

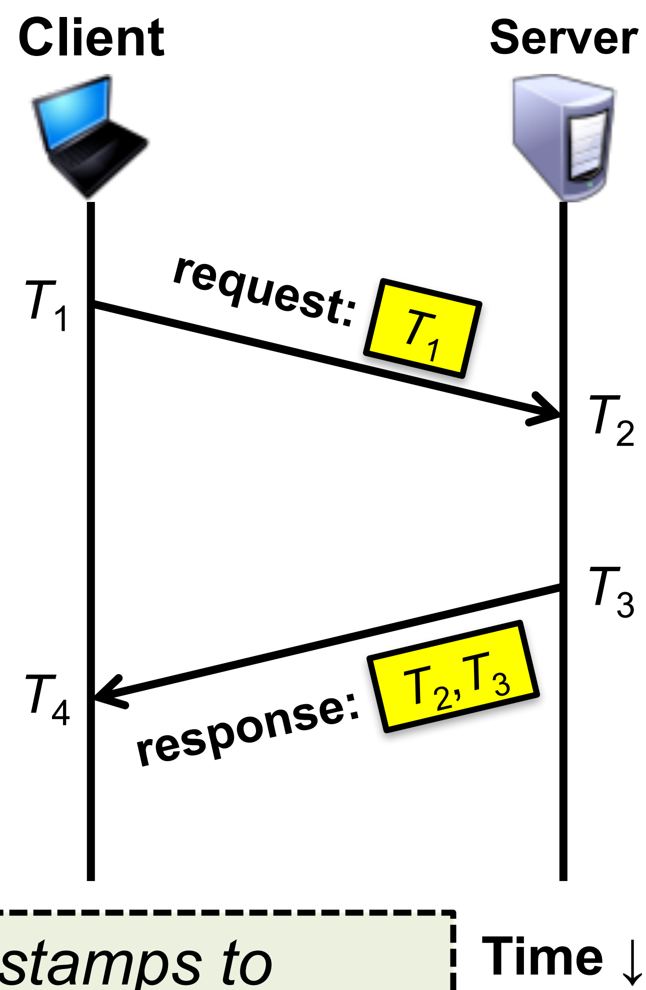
- Suppose a server with an accurate clock (*e.g.*, GPS-disciplined crystal oscillator)
  - Could simply issue an RPC to obtain the time:



- 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  - **Message delays** will have **outdated** server's answer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?

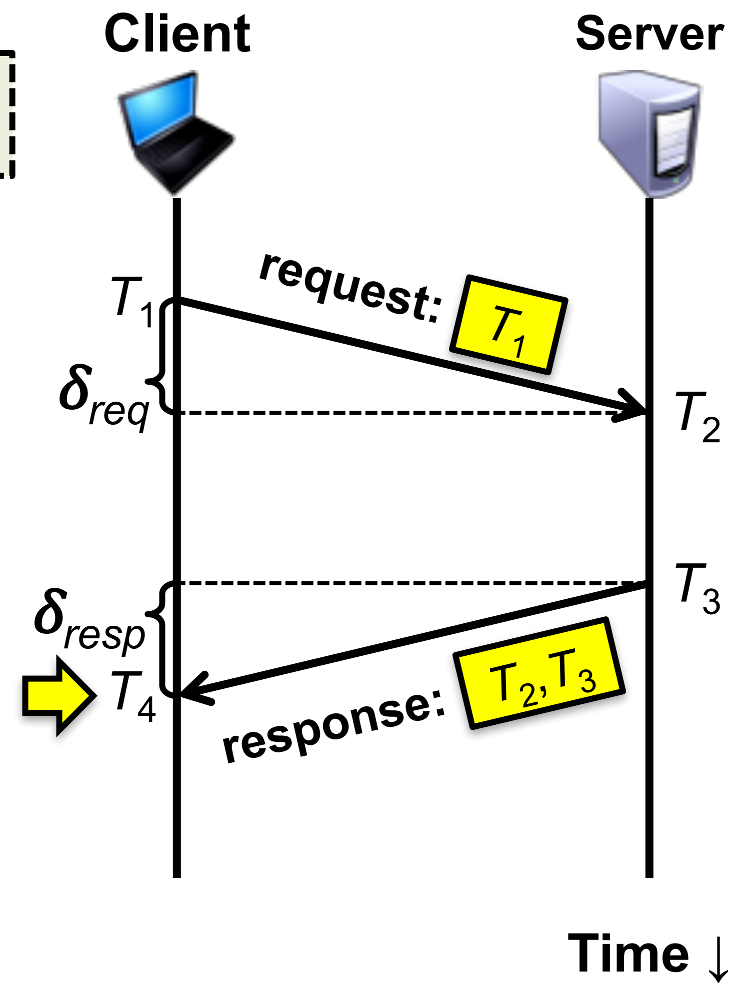
Cristian's algorithm: Offset sample calculation

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

- Client samples ***round trip time*** δ = $\delta_{req} + \delta_{resp} = (T_4 - T_1) - (T_3 - T_2)$
- **But client knows δ, not δ_{resp}**

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

Client sets clock ← $T_3 + ½\delta$



Today

1. The need for time synchronization

2. **"Wall clock time" synchronization**
   - Cristian's algorithm, **Berkeley algorithm,** NTP

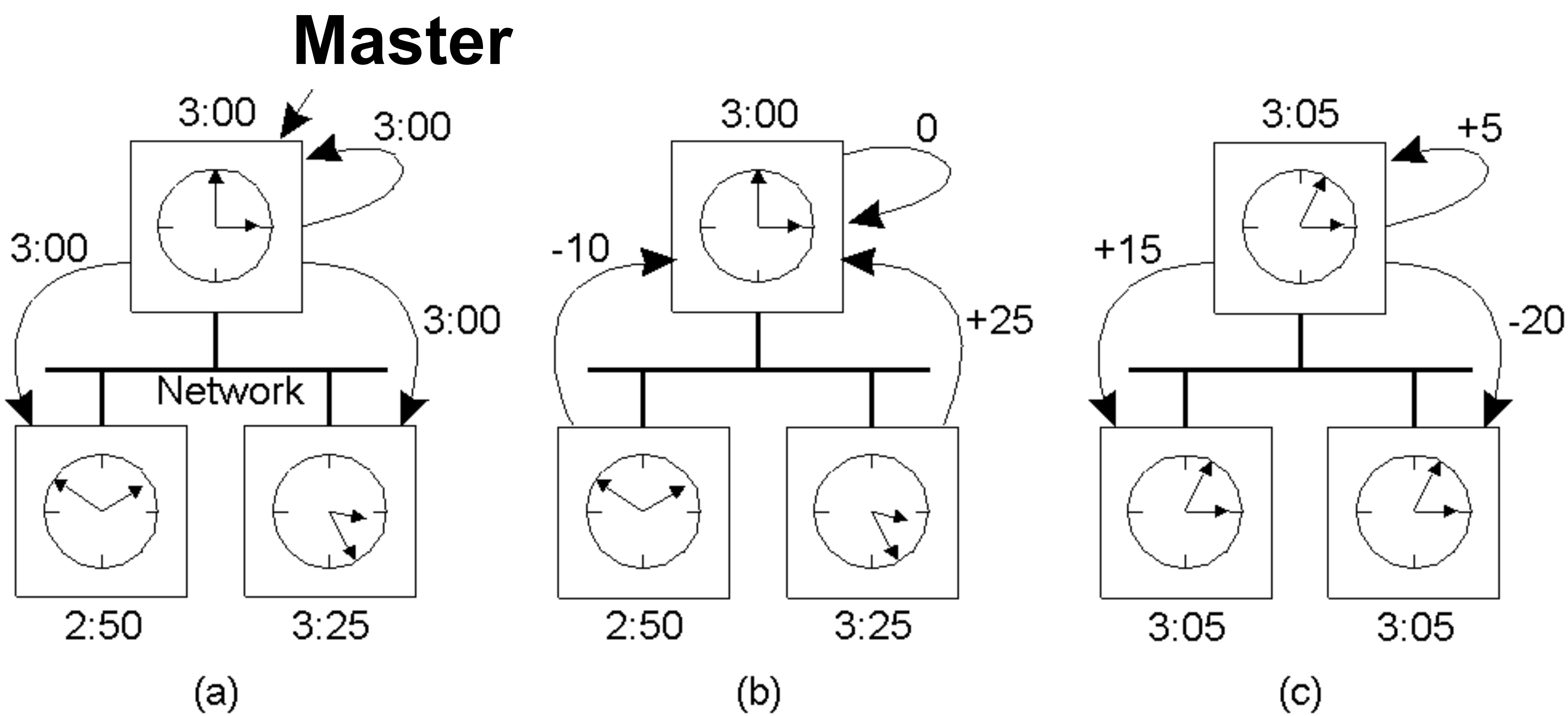
3. Logical Time
   - Lamport clocks
   - Vector clocks

Berkeley algorithm

- A **single time server** can **fail**, blocking timekeeping
- 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

Berkeley algorithm

- ***Master machine***: polls ***L*** other machines using Cristian's algorithm → { θ_i } (i = 1…L)



Today

1. The need for time synchronization

2. **"Wall clock time" synchronization**
   – Cristian's algorithm, Berkeley algorithm, **NTP**

3. Logical Time
   – Lamport clocks
   – Vector clocks

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 operation: Server selection

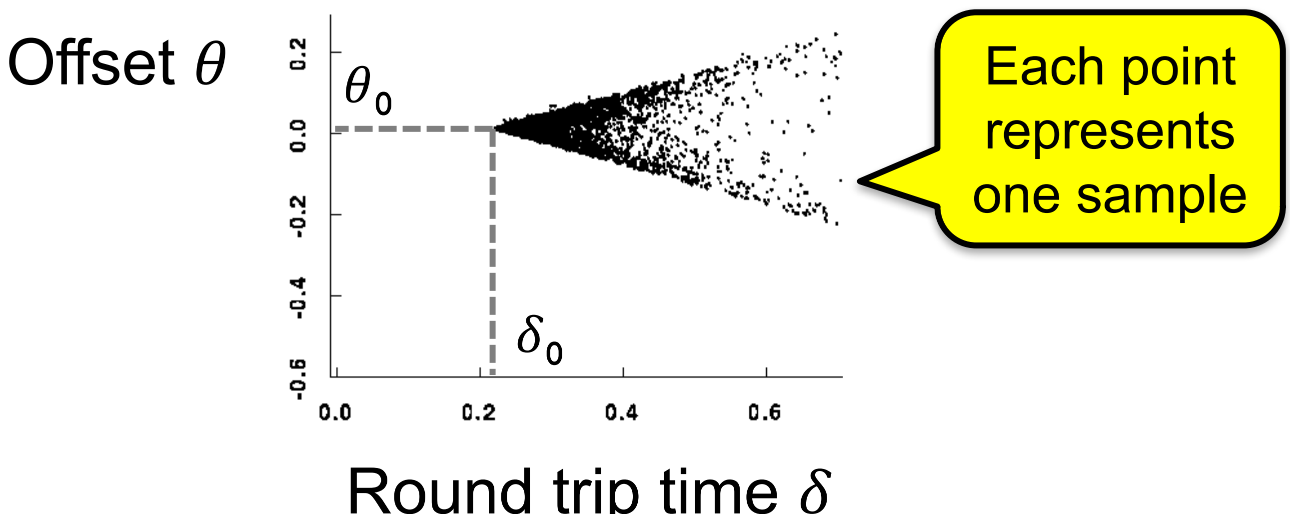
- Messages between an NTP client and server are exchanged in pairs: request and response
  - Use Cristian's algorithm

- 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**

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

Clock synchronization: Take-away points

- Clocks on different systems will always behave differently
  - Disagreement between machines can result in undesirable behavior
- NTP, Berkeley clock synchronization
  - Rely on timestamps to estimate network delays
  - **100s μs−ms accuracy**
  - Clocks never exactly synchronized
- Often **inadequate** for distributed systems
  - Often need to reason about the **order of events**
  - Might need precision on the order of **ns**

Today

1. The need for time synchronization

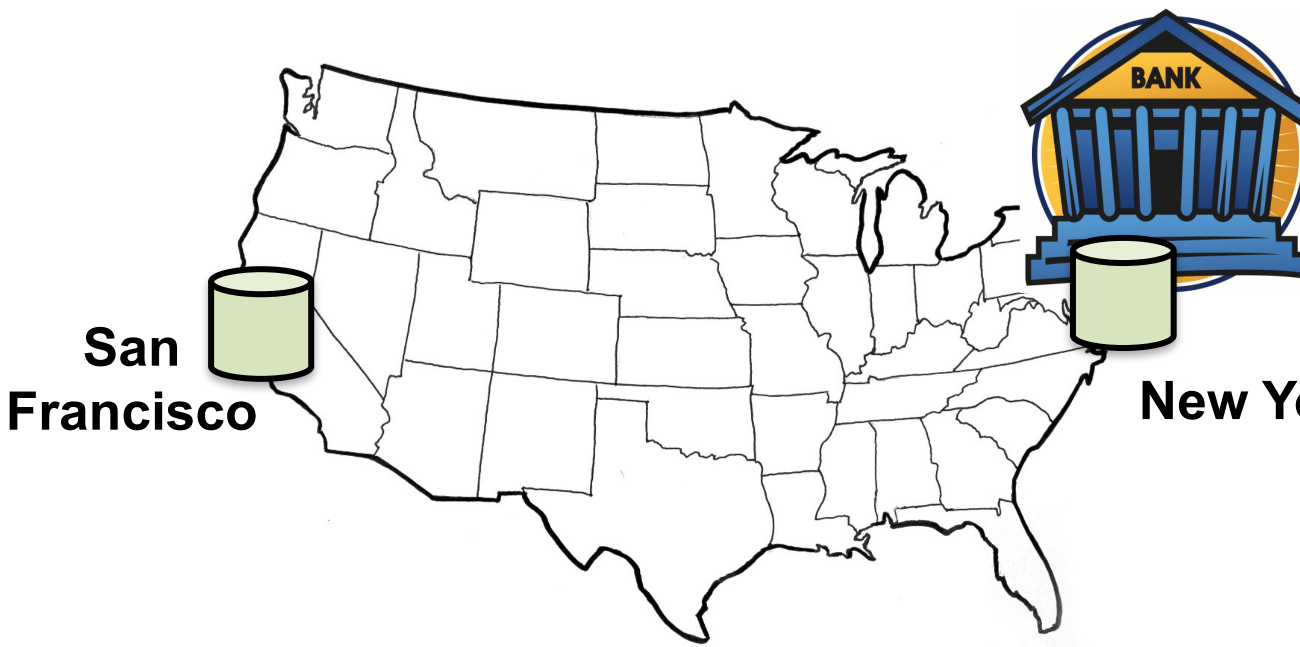
2. "Wall clock time" synchronization
   – Cristian's algorithm, Berkeley algorithm, NTP

3. **Logical Time**
   – **Lamport clocks**
   – Vector clocks

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



The consequences of concurrent updates

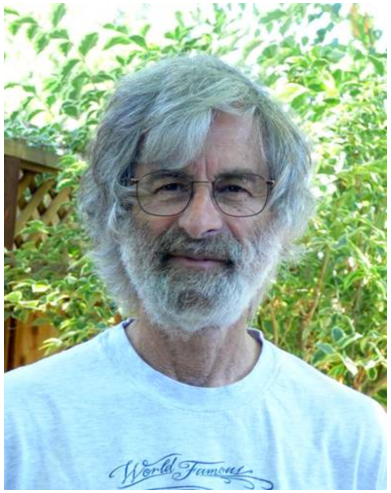
- **Replicate** the database, keep one copy in sf, one in nyc
  - Client sends **query** to the **nearest** copy
  - Client sends **update to both** copies

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

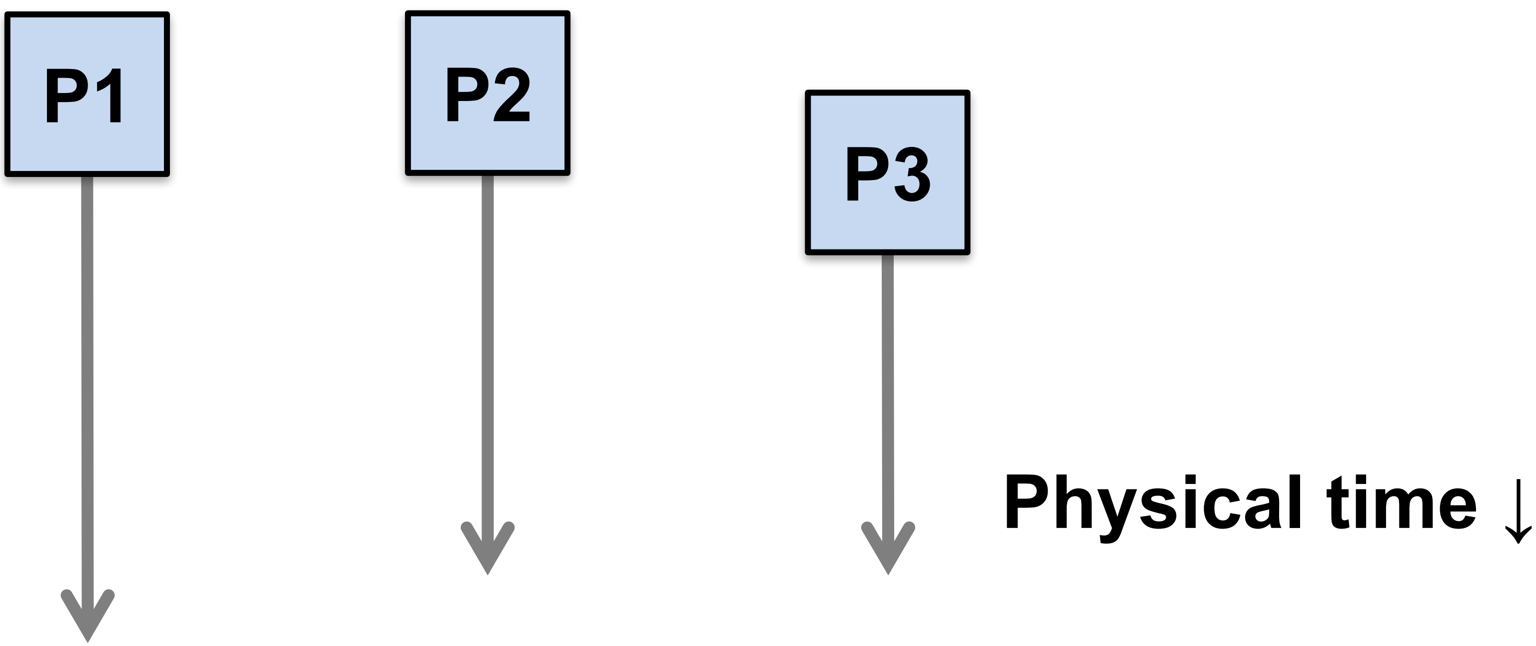


Idea: Disregard the precise clock time

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

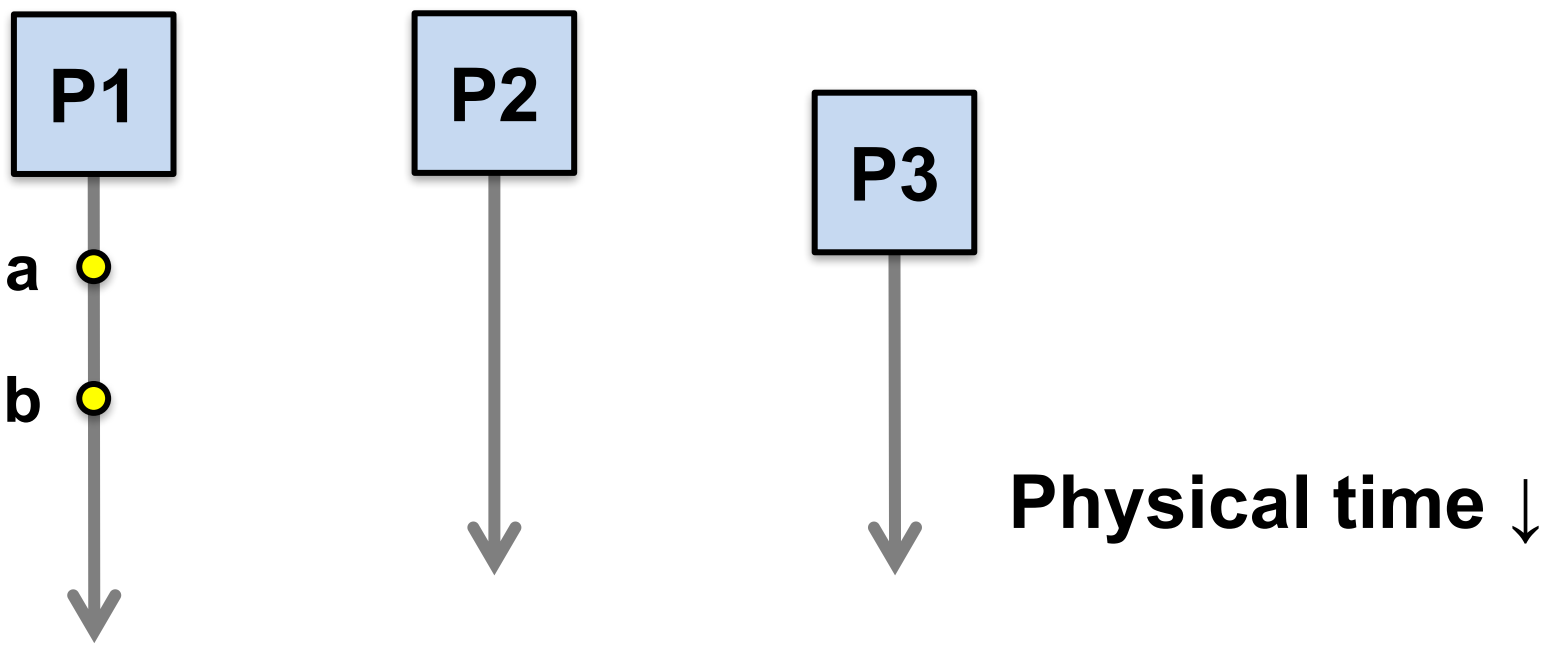
Defining "happens-before"

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



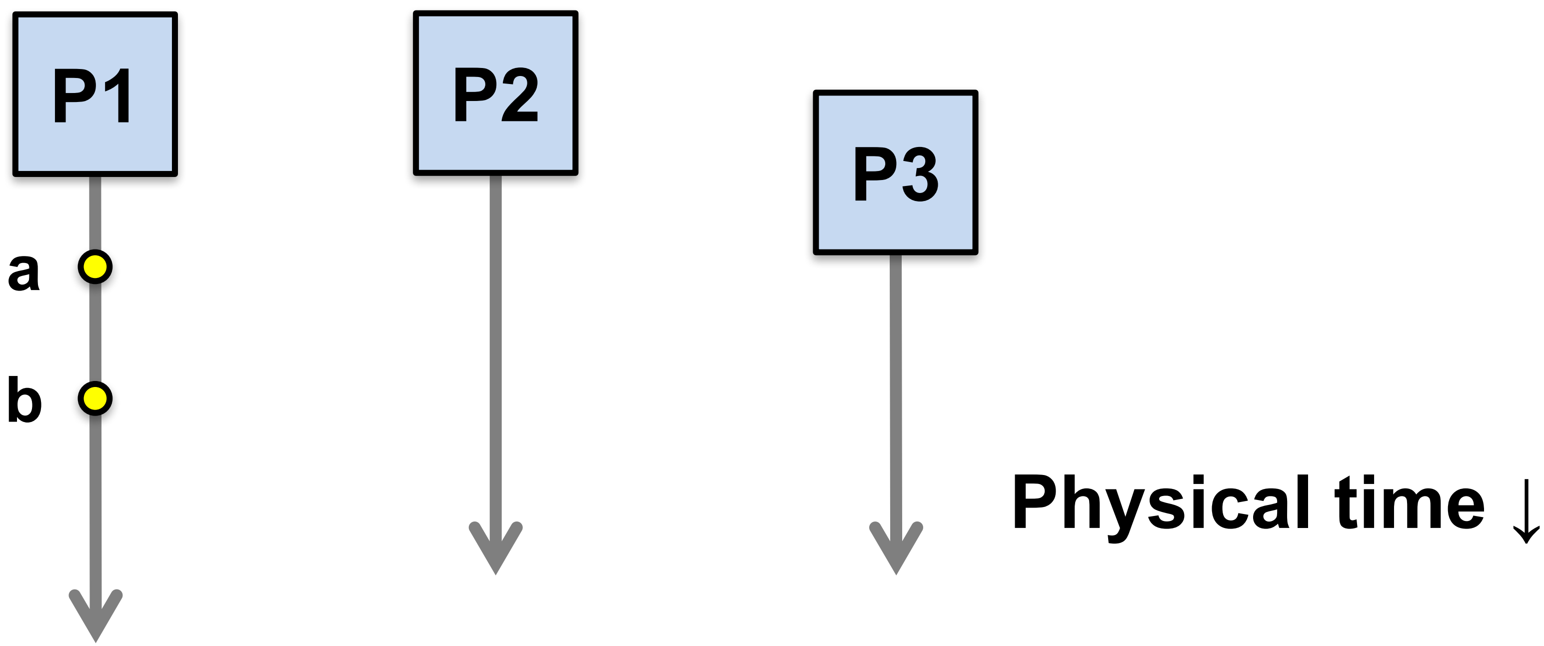
Defining "happens-before"

1. Can observe event order at a single process



Defining “happens-before”

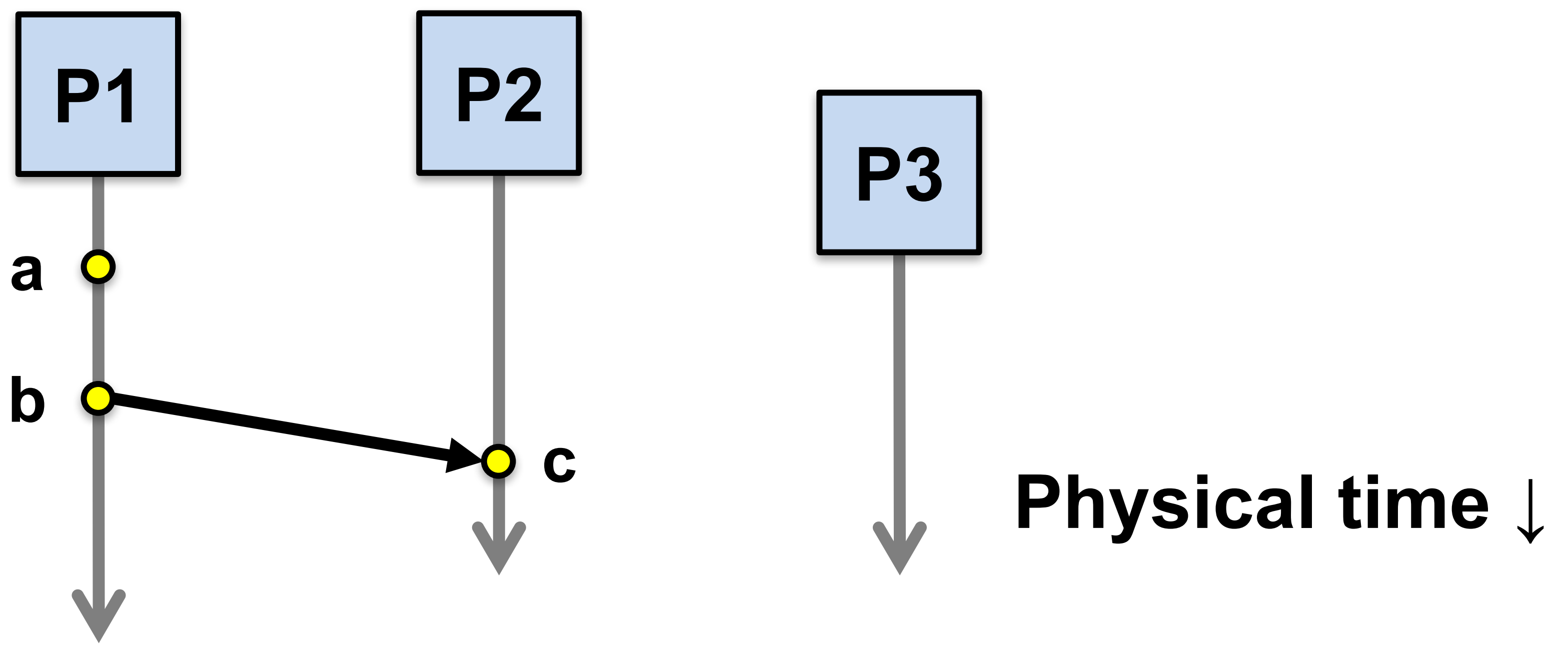
1. If **same process** and **a** occurs before **b**, then **a** → **b**



Defining "happens-before"

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

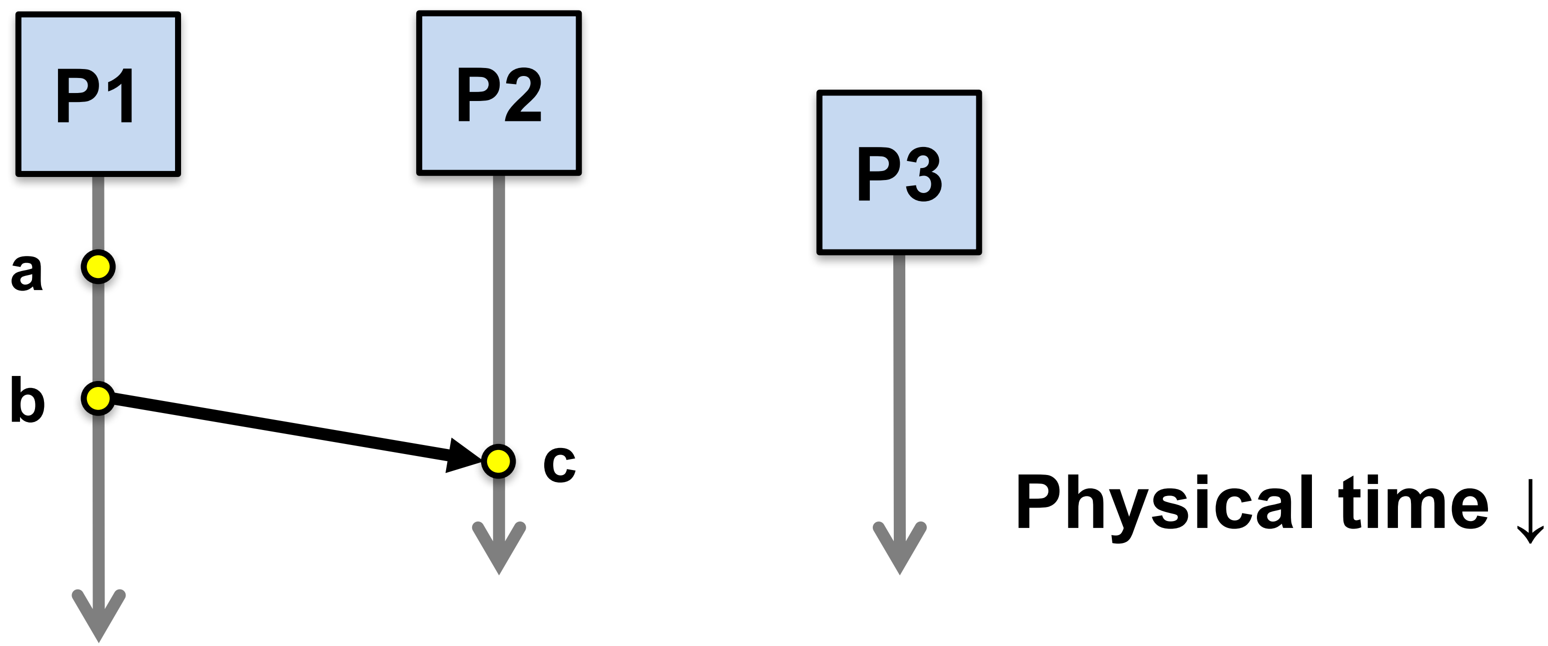
2. Can observe ordering when processes communicate



Defining "happens-before"

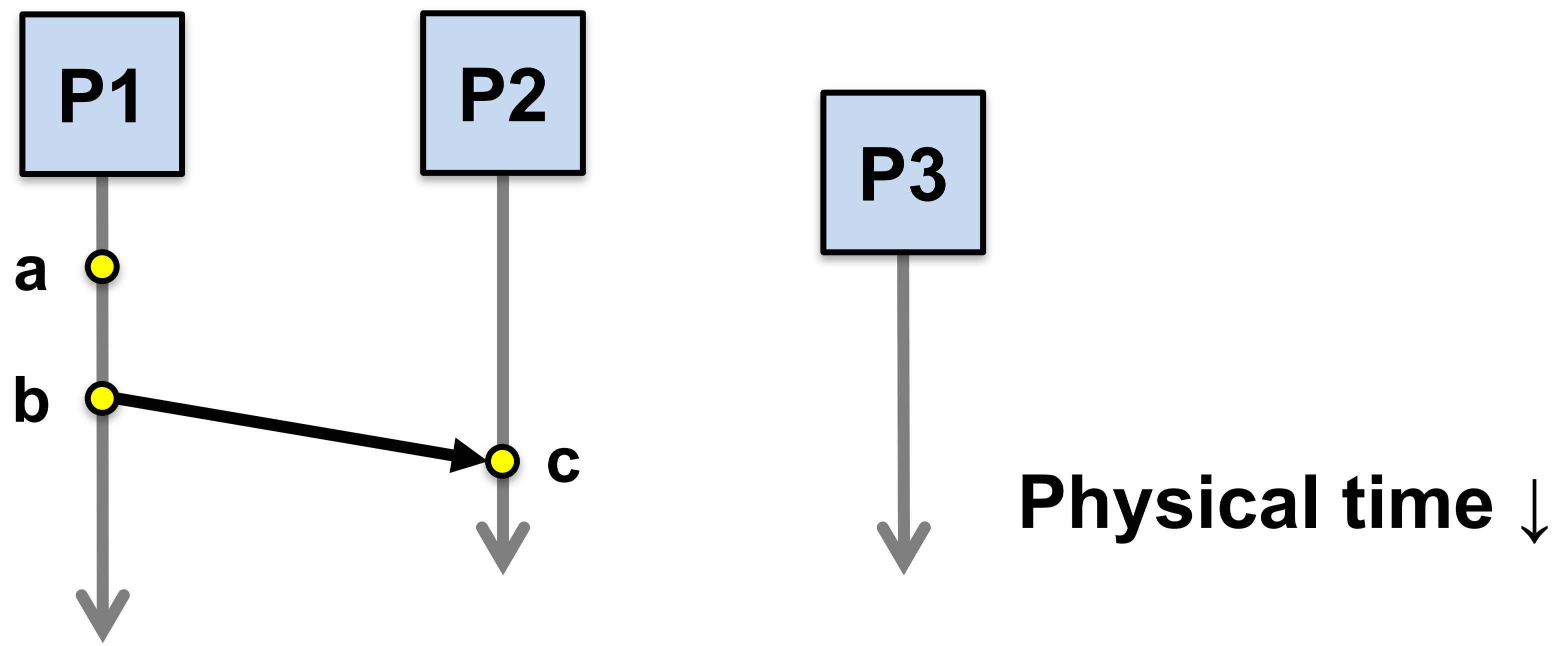
1. If **same process** and **a** occurs before **b**, then **a** → **b**

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

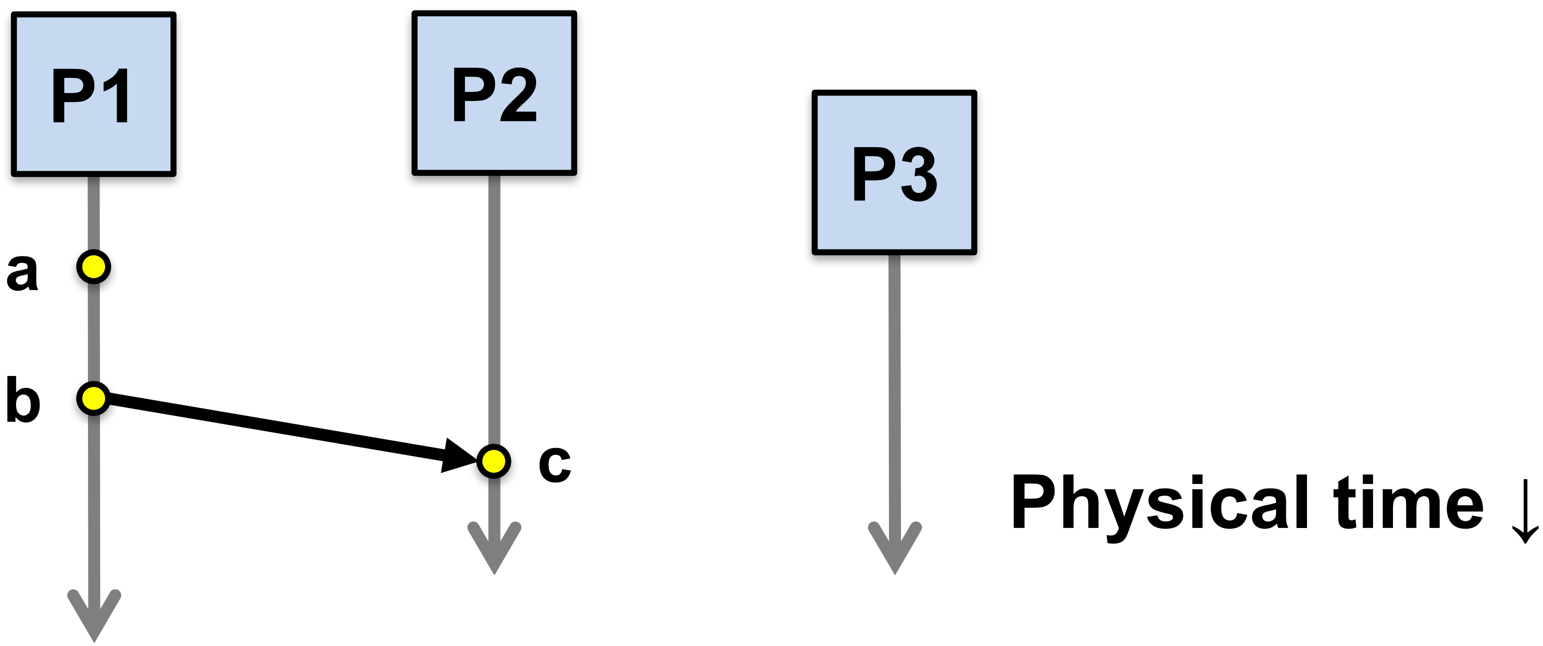


Defining "happens-before"

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

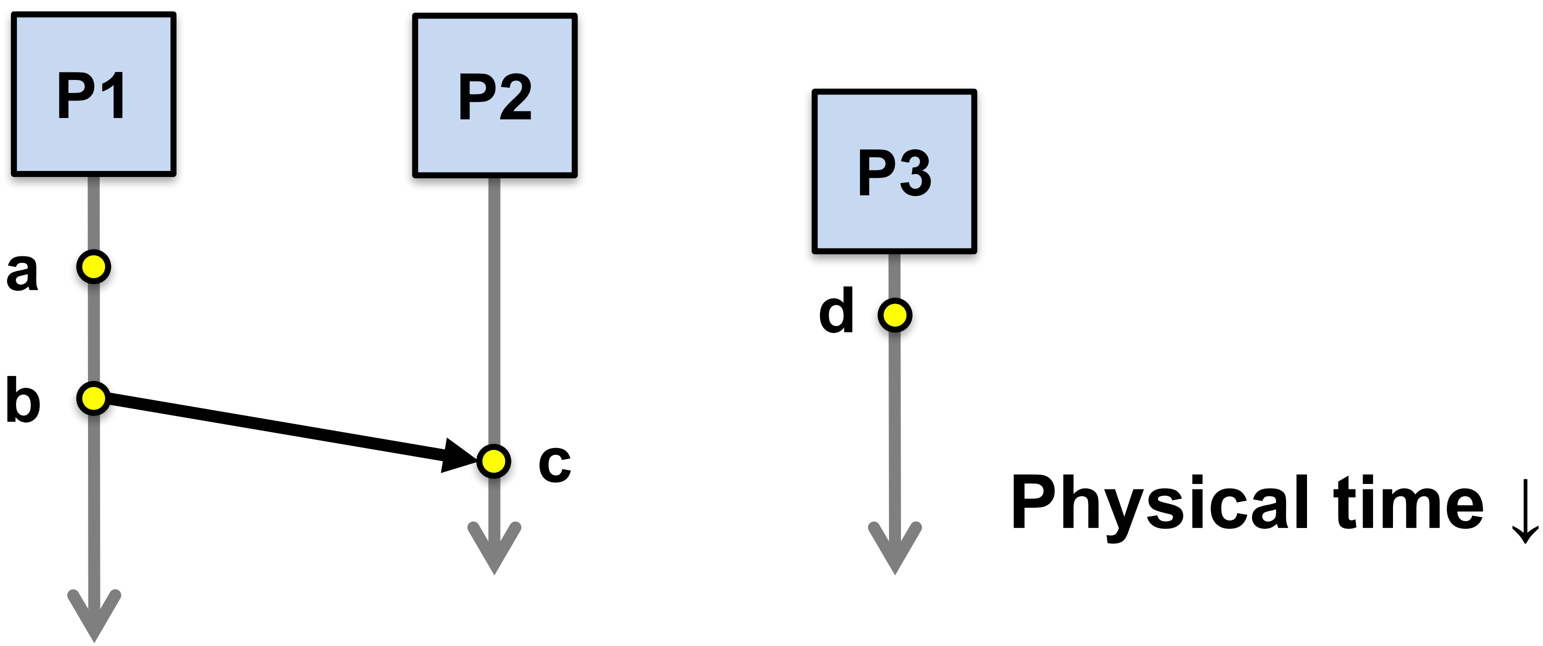
2. If **c** is a message receipt of **b**, then **b** → **c**

3. If **a** → **b** and **b** → **c**, then **a** → **c**



Concurrent events

- Not all events are related by →
- **a, d** not related by → so ***concurrent,*** written as **a** || **d**



Lamport clocks: Objective

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

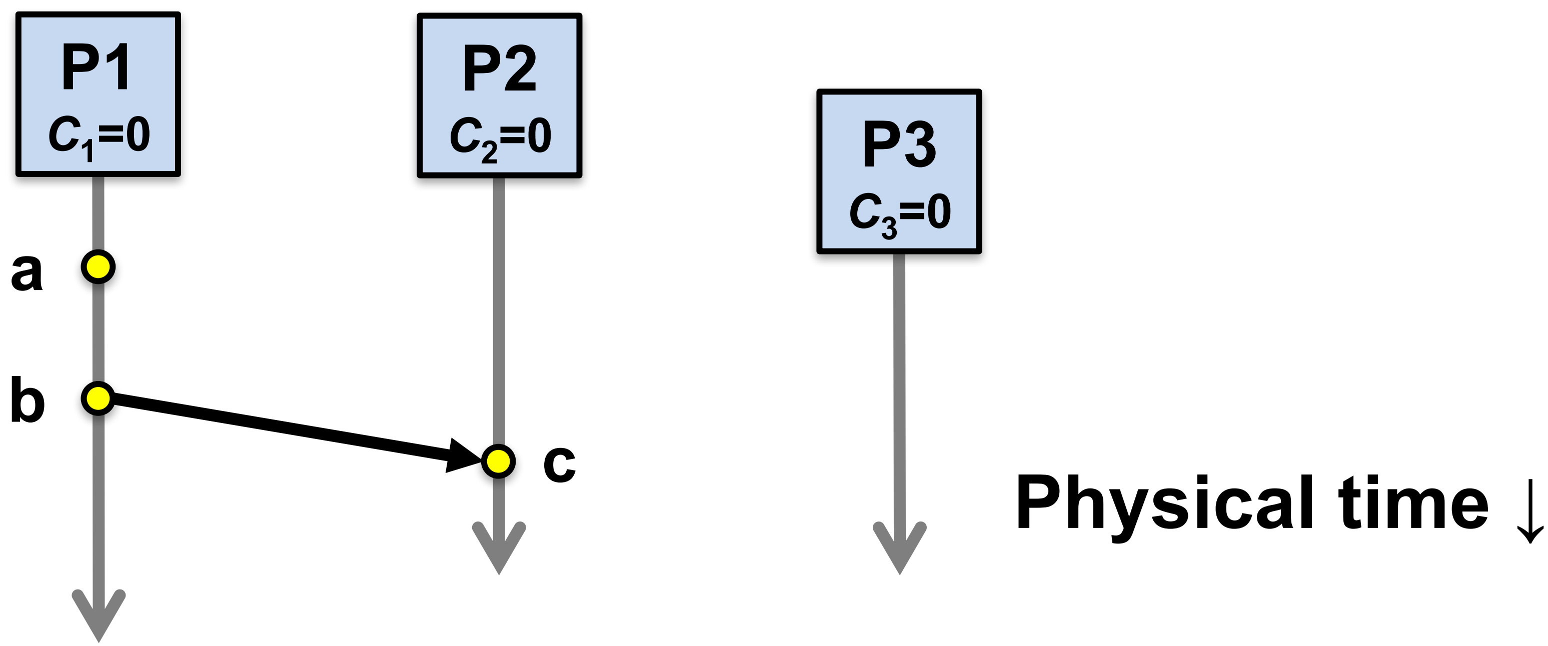
Plan: Tag events with clock times; use **clock times** to make distributed system correct

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

The Lamport Clock algorithm

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

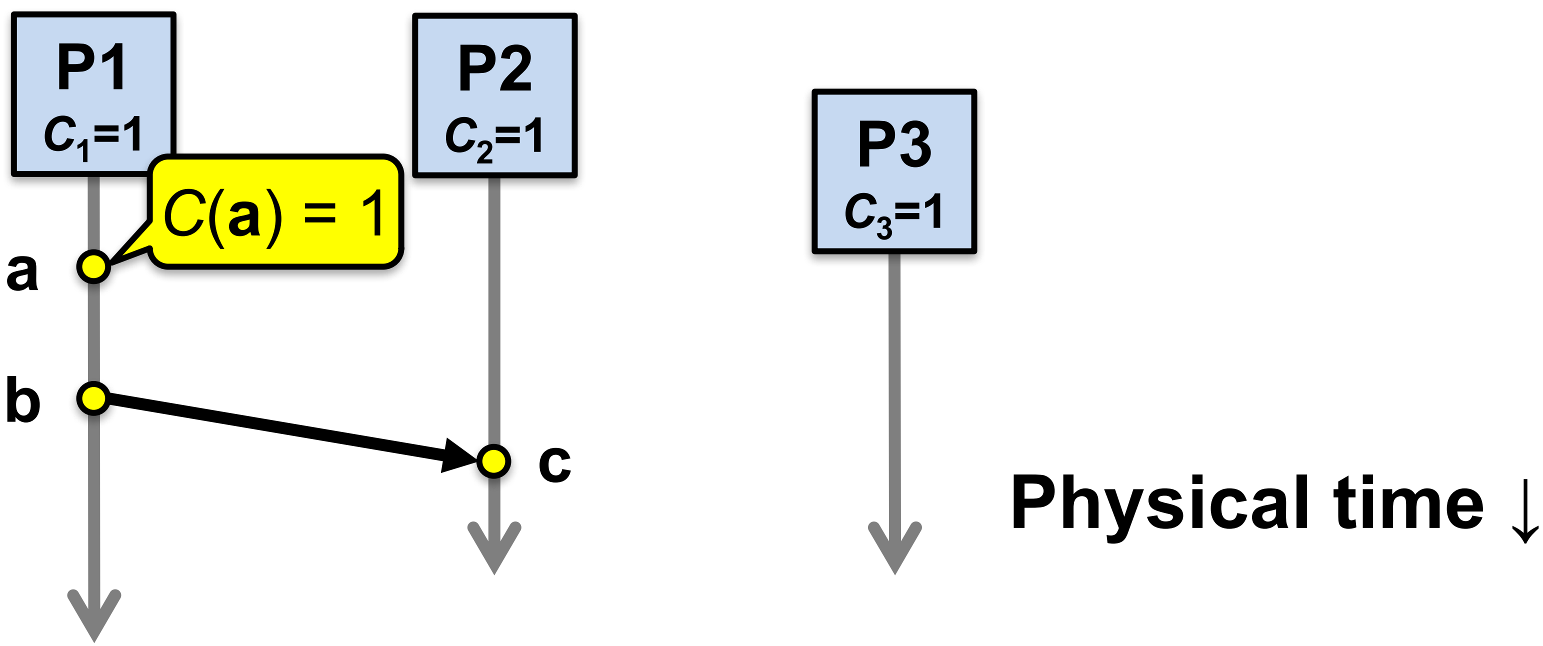
1. Before executing an event, $C_i \leftarrow C_i + 1$



The Lamport Clock algorithm

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

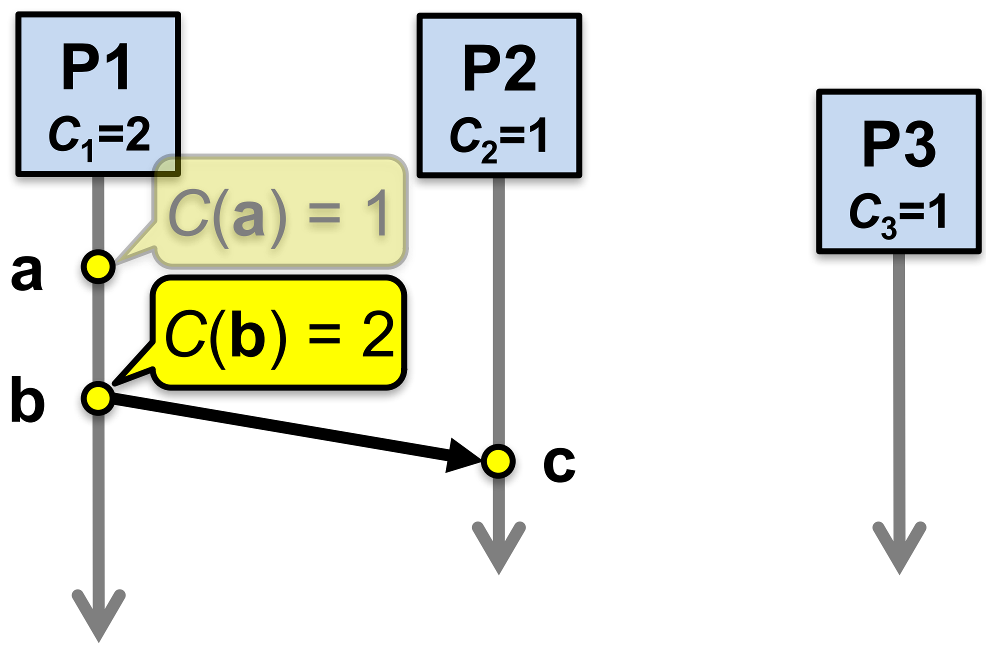
   – Set event time $C(\mathbf{a}) \leftarrow C_i$



The Lamport Clock algorithm

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

   – Set event time $C(\mathbf{b}) \leftarrow C_i$

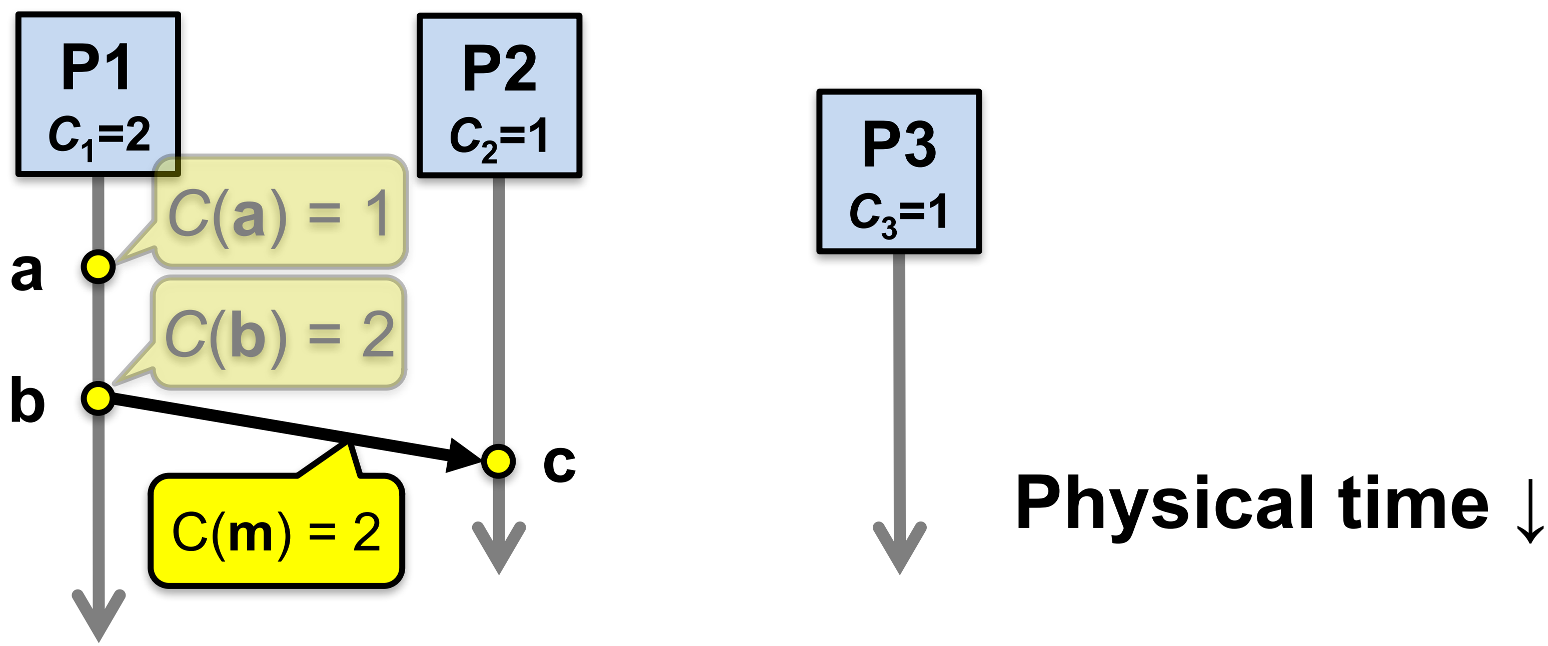


Physical time ↓

The Lamport Clock algorithm

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

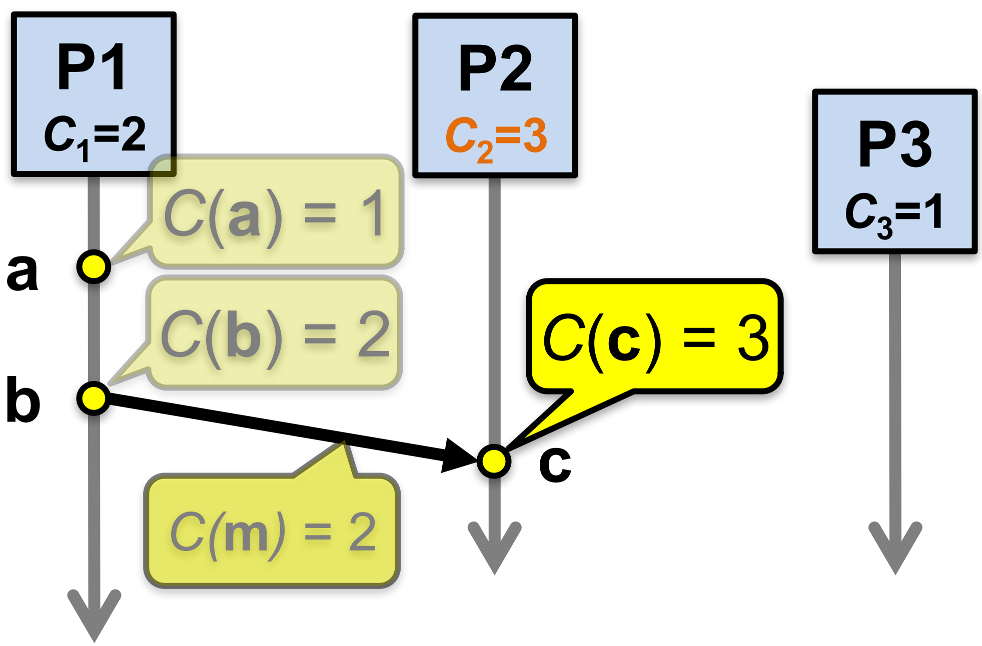
2. Send the local clock in the message **m**



The Lamport Clock algorithm

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

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



Physical time ↓

Ordering all events

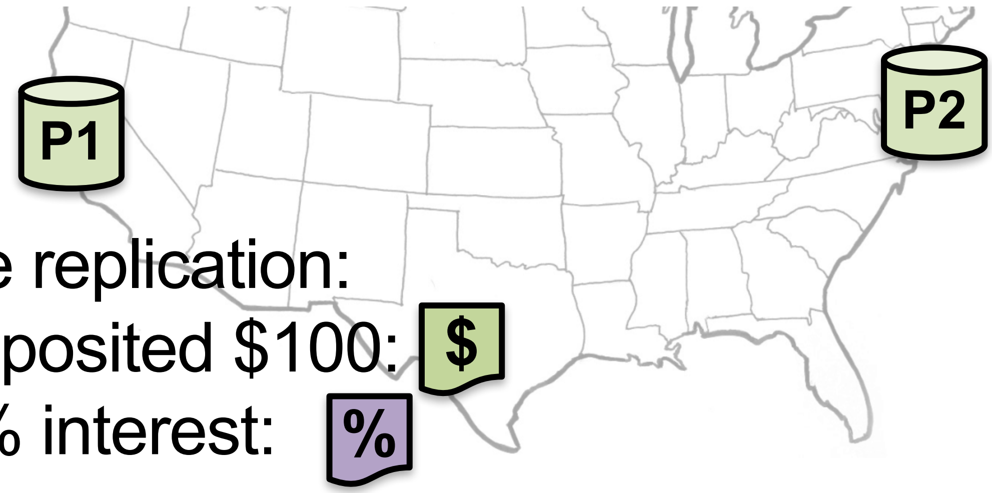
- **Break ties** by **appending the process number** to each event:

    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(**a**) < C(**b**) or C(**b**) < C(**a**)
    - This is called a **total ordering** of events

Making concurrent updates consistent



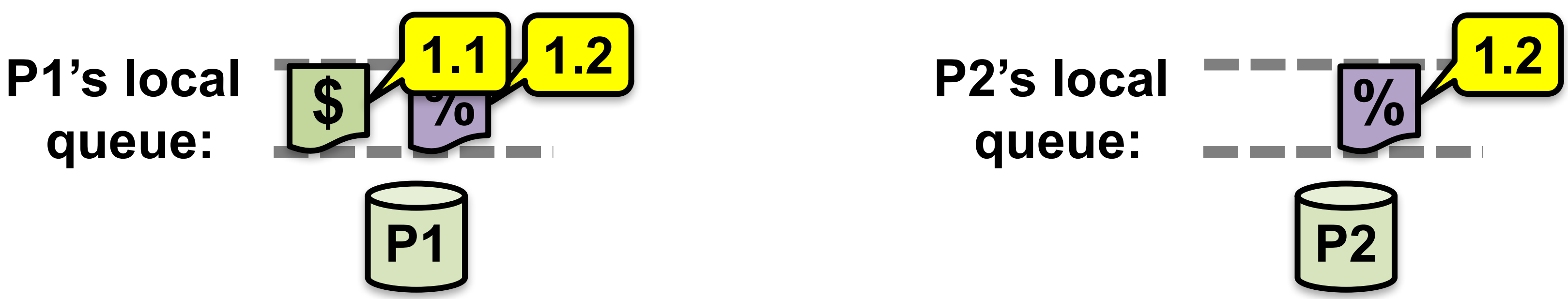
- Recall multi-site database replication:
  - San Francisco (**P1**) deposited $100:
  - New York (**P2**) paid 1% interest:

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 (Almost correct)

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

2. On **receiving** an **event from replica:**
   a) Add it to your local queue
   b) Broadcast an ***acknowledgement message*** to every process (including yourself)

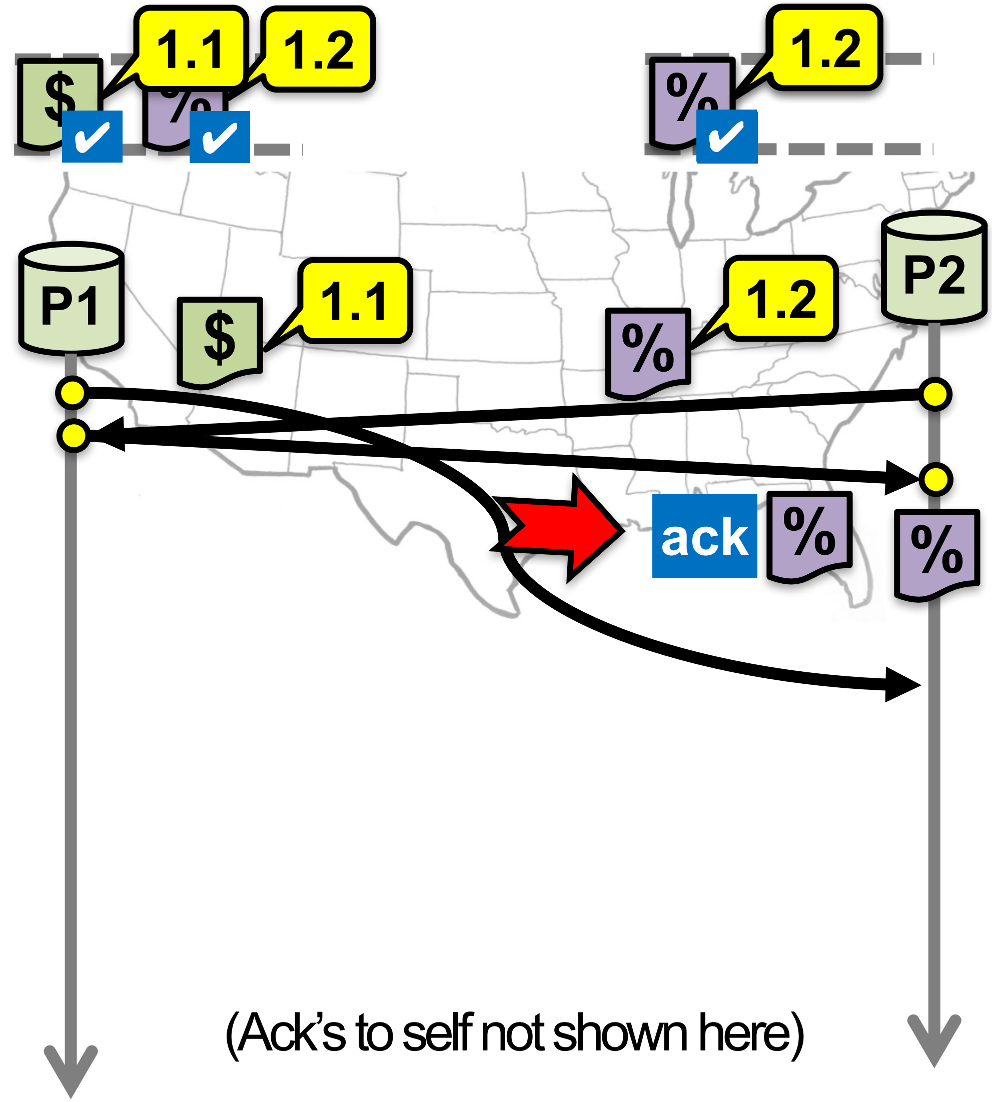
3. On **receiving** an **acknowledgement:**
   – Mark corresponding event ***acknowledged*** in your queue

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

Totally-Ordered Multicast (Almost correct)

- **P1** queues $, **P2** queues %
- **P1** queues and **ack's %**
  - **P1** marks % fully **ack'ed**
- **P2** marks % fully **ack'ed**

P2 processes %



Totally-Ordered Multicast (Correct version)

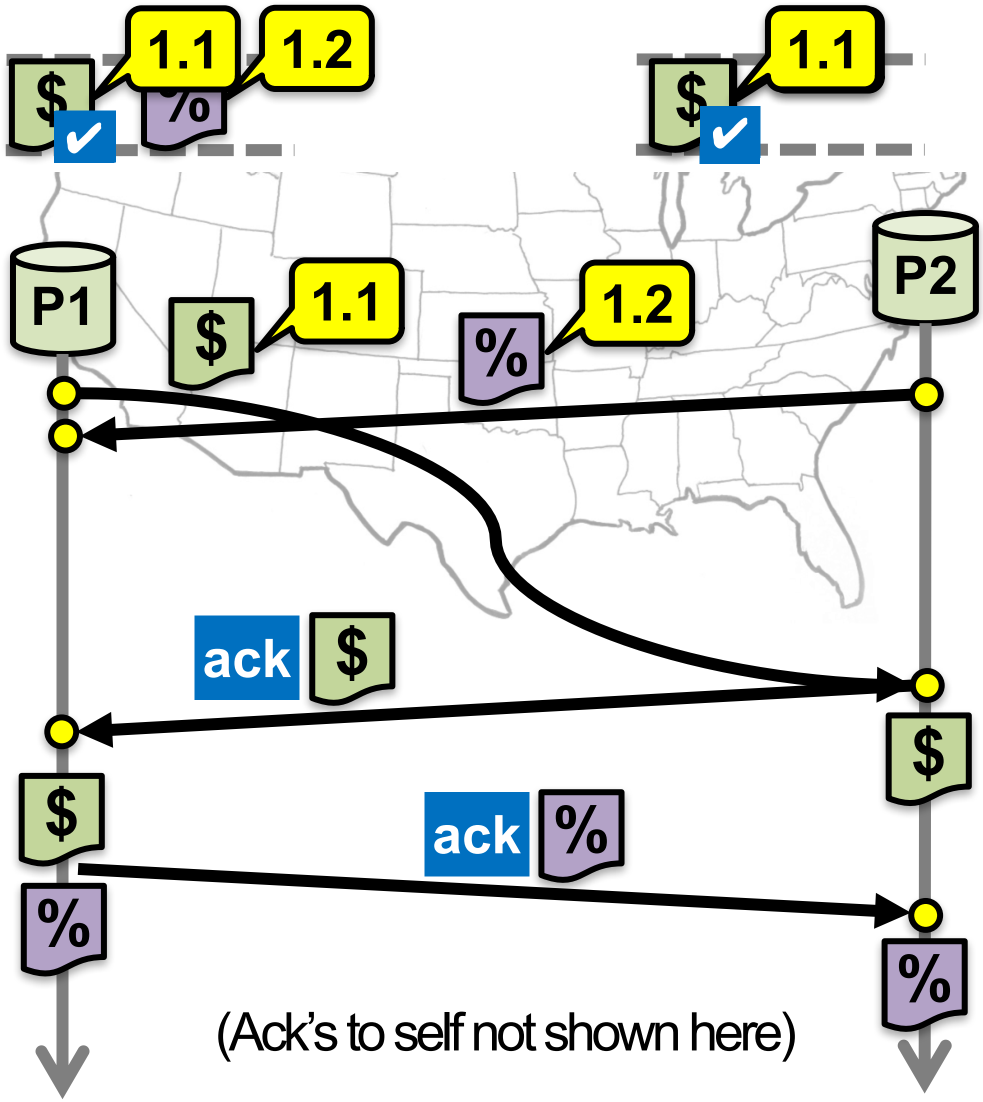
1. On **receiving** an event from **client**, broadcast to others (including yourself)

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 **everyone** has ack'ed from **head** of queue

Totally-Ordered Multicast (Correct version)



(Ack's to self not shown here)

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?)**

Take-away points: Lamport clocks

- Can **totally-order** events in a distributed system: that's useful!

- **But:** while by construction, **a** → **b** implies *C*(**a**) < *C*(**b**),
  - The converse is not necessarily true:
    - *C*(**a**) < *C*(**b**) does not imply **a** → **b** (possibly, **a** || **b**)

Can't use Lamport clock timestamps to infer causal relationships between events

Today

1. The need for time synchronization

2. "Wall clock time" synchronization
   - Cristian's algorithm, Berkeley algorithm, NTP

3. **Logical Time**
   - Lamport clocks
   - **Vector clocks**

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]

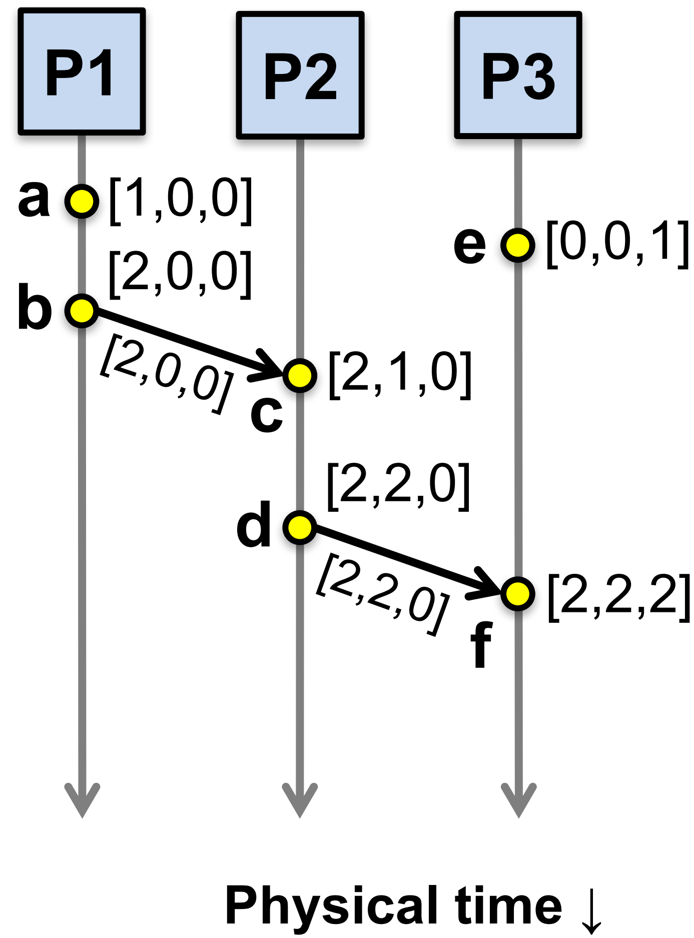
- Two **update rules:**

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

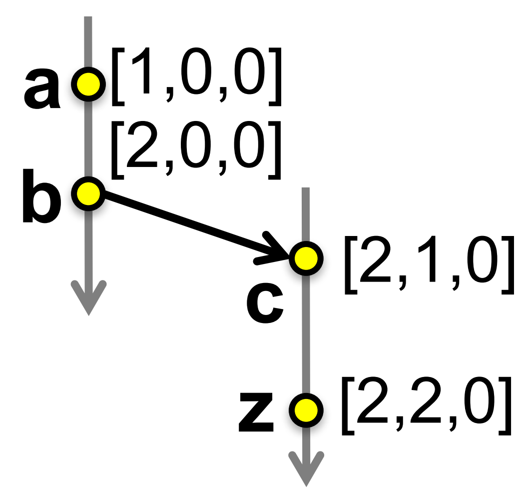


Vector clocks can establish causality

- 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



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

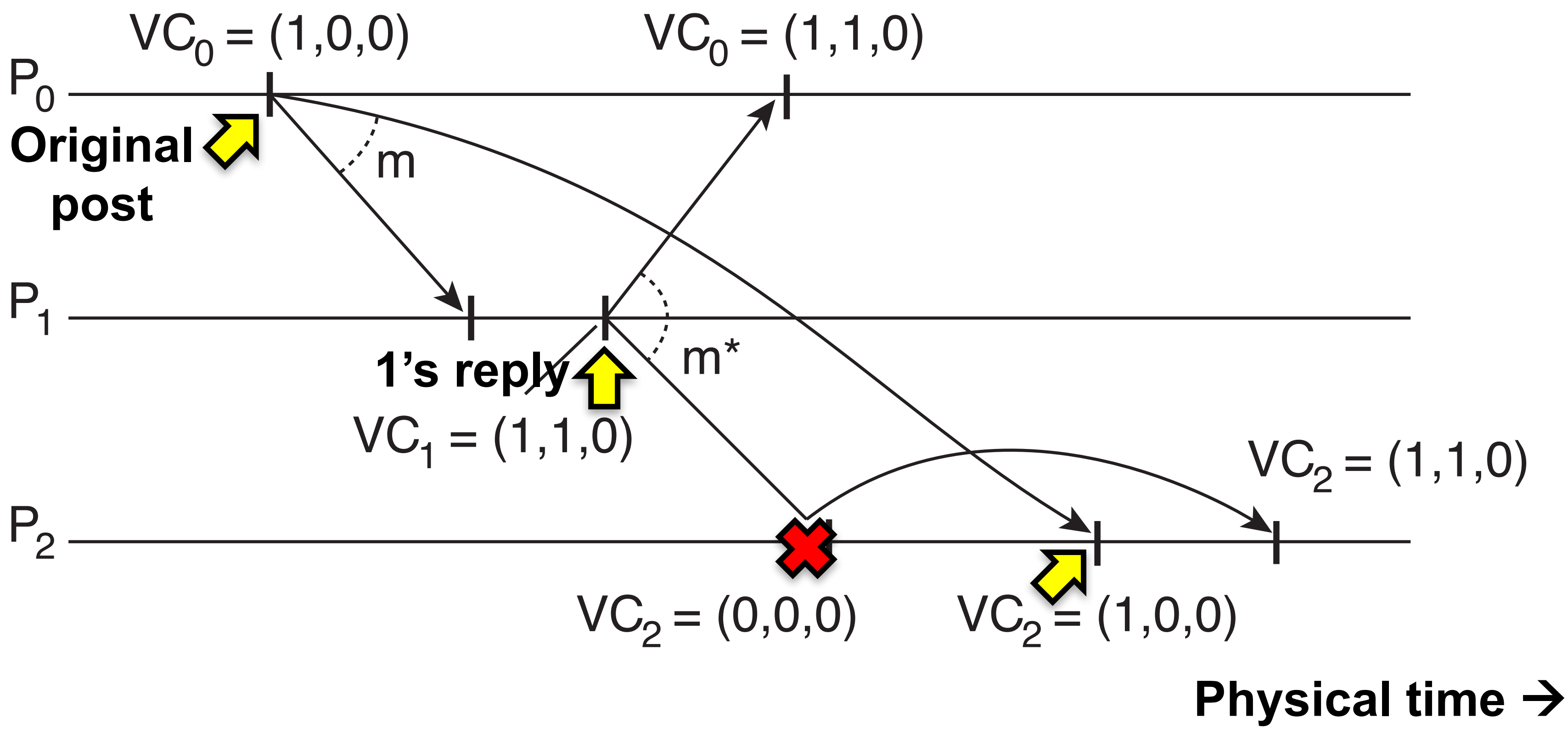
VC application: Causally-ordered bulletin board system

- 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

Wednesday Topic:

Lab 1 – Virtualization, sockets, RPCs

Why global timing?

- Suppose there were an **infinitely-precise and globally consistent** time standard

- That would be very handy. For example:

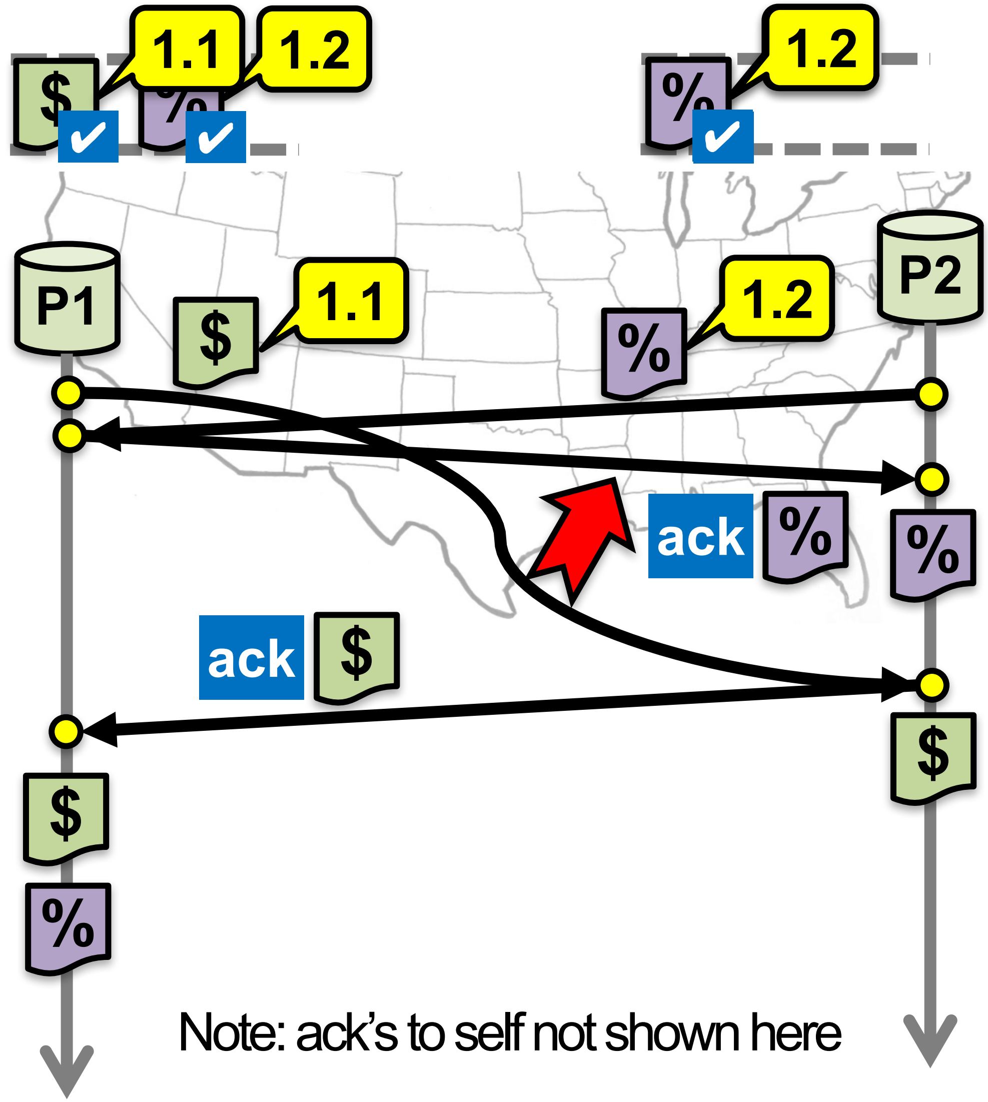
1. *Who got last seat on airplane?*

2. **Mobile cloud gaming:** *Which was first, A shoots B or vice-versa?*

3. *Does this file need to be recompiled?*

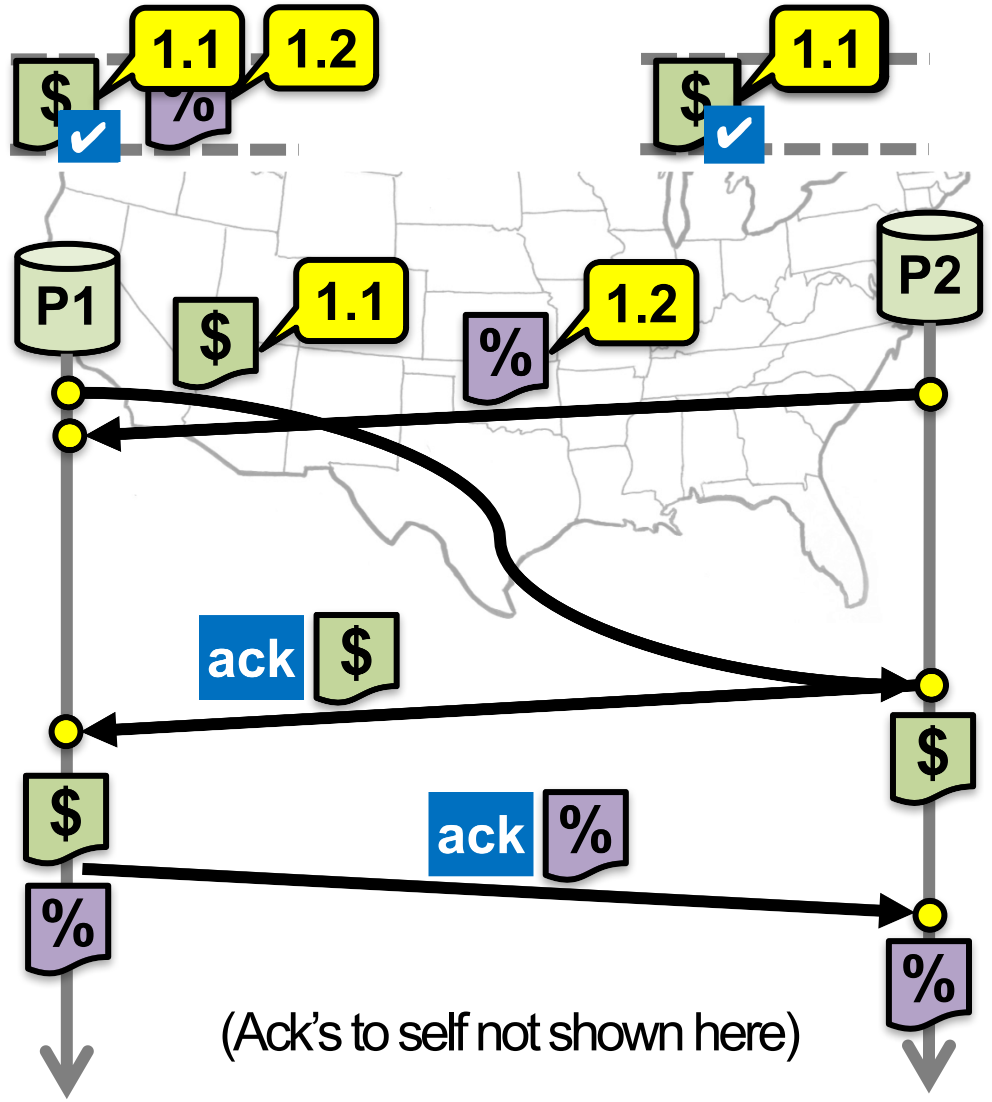
Totally-Ordered Multicast (Attempt #1)

- **P1** queues $, **P2** queues %
- **P1** queues and **ack's** %
  - **P1** marks % fully **ack'ed**
- **P2** marks % fully **ack'ed**
  - **P2 processes %**
- **P2** queues and **ack's** $
  - **P2 processes $**
- **P1** marks $ fully **ack'ed**
  - **P1** processes $, then %



Totally-Ordered Multicast (Correct version)

- **P1** queues $, **P2** queues %
- **P1** queues %
- **P2** queues and **ack's $**
- **P2** marks $ fully **ack'ed**
  - **P2** processes $
- **P1** marks $ fully **ack'ed**
  - **P1** processes $
  - **P1** **ack's %**
- **P1** marks % fully **ack'ed**
  - **P1** processes %
- **P2** marks % fully **ack'ed**
  - **P2** processes %



Time standards

- **Universal Time** (UT1)
  - In concept, based on astronomical observation of the sun at 0^{o} longitude
  - Known as "Greenwich Mean Time"

- **International Atomic Time** (TAI)
  - Beginning of TAI is midnight on January 1, 1958
  - Each second is 9,192,631,770 cycles of radiation emitted by a Cesium atom
  - Has diverged from UT1 due to slowing of earth's rotation

- **Coordinated Universal Time** (UTC)
  - TAI + leap seconds, to be within 0.9 seconds of UT1
  - Currently TAI − UTC = 36

VC application: Order processing

- Suppose we are running a **distributed order processing system**

- Each process = a different user
- Each event = an order

- A user has seen all orders with V(order) < the user's current vector