More Time and Clocks
(Lamport and Vector Clocks)

CS6450: Distributed Systems

Lecture 5

Ryan Stutsman

Takeaways

- Lamport Clock algorithm
  - Understand guarantee it provides:
    if $a \rightarrow b$ then $C(a) < C(b)$
  - Understand how to use it to **generate a total order** of events (even if those events happen independently)
- Vector Clocks
  - If $V(a) < V(b)$ then $a \rightarrow b$
  - If $V(a) \nless V(b)$ and $V(b) \nless V(a)$ then $a \parallel b$
  - Can use to infer when an event b was aware of/influenced by a

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**

- Cristian's algorithm used only one sided delay
  - potential inaccuracy is half the additional RTT delay

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

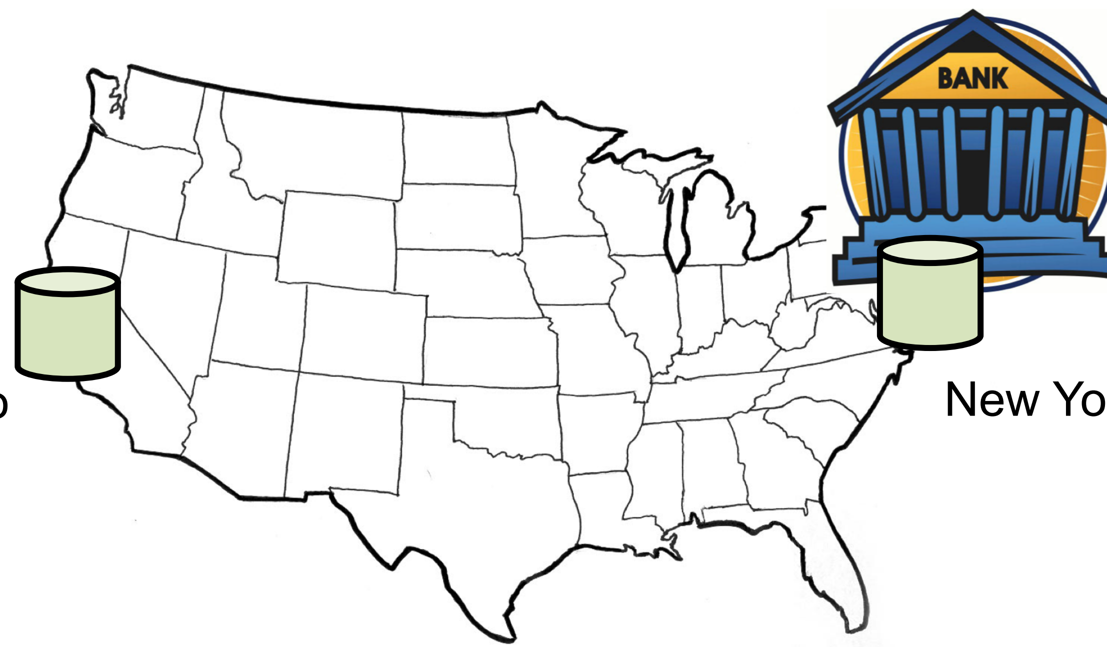
2. “Wall clock time” synchronization
   - Cristian’s algorithm, Berkeley algorithm, NTP

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

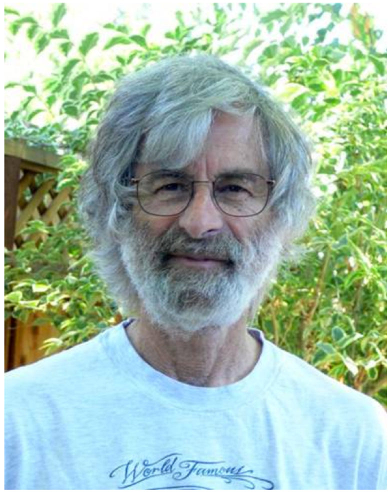


The consequences of concurrent updates

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

Idea: *Logical* clocks

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



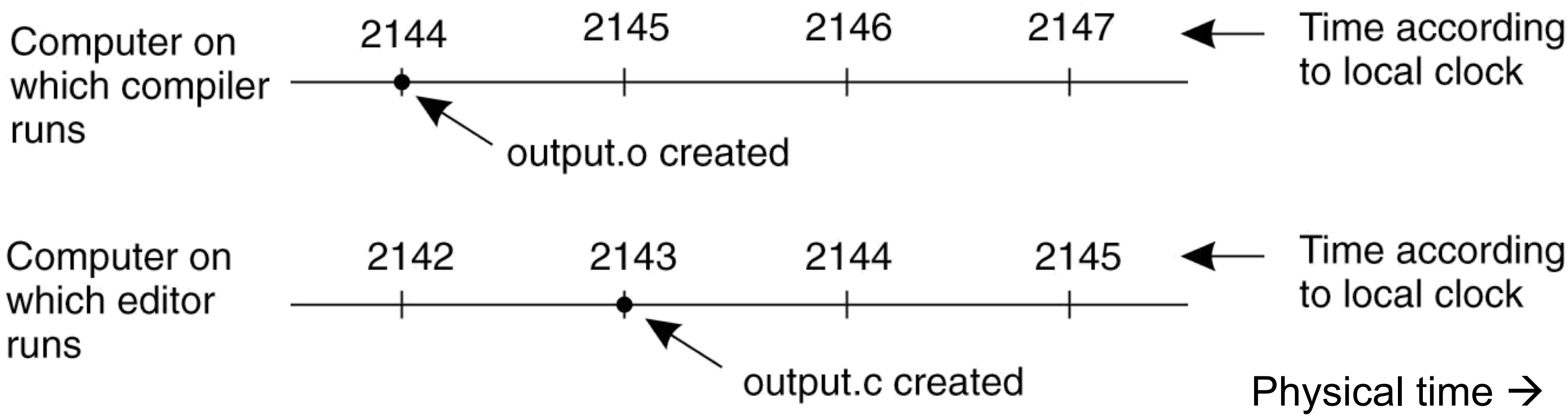
Idea: Disregard the precise clock time

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

In a classic paper, Lamport (1978) showed that although clock synchronization is possible, it need not be absolute. **If two processes do not interact, it is not necessary that their clocks be synchronized because the lack of synchronization would not be observable and thus could not cause problems.** Furthermore, he pointed out that **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**. In the *make* example, what counts is whether *input.c* is older or newer than *input.o,* not their absolute creation times.

-Tanenbaum

Will any (total) order do?



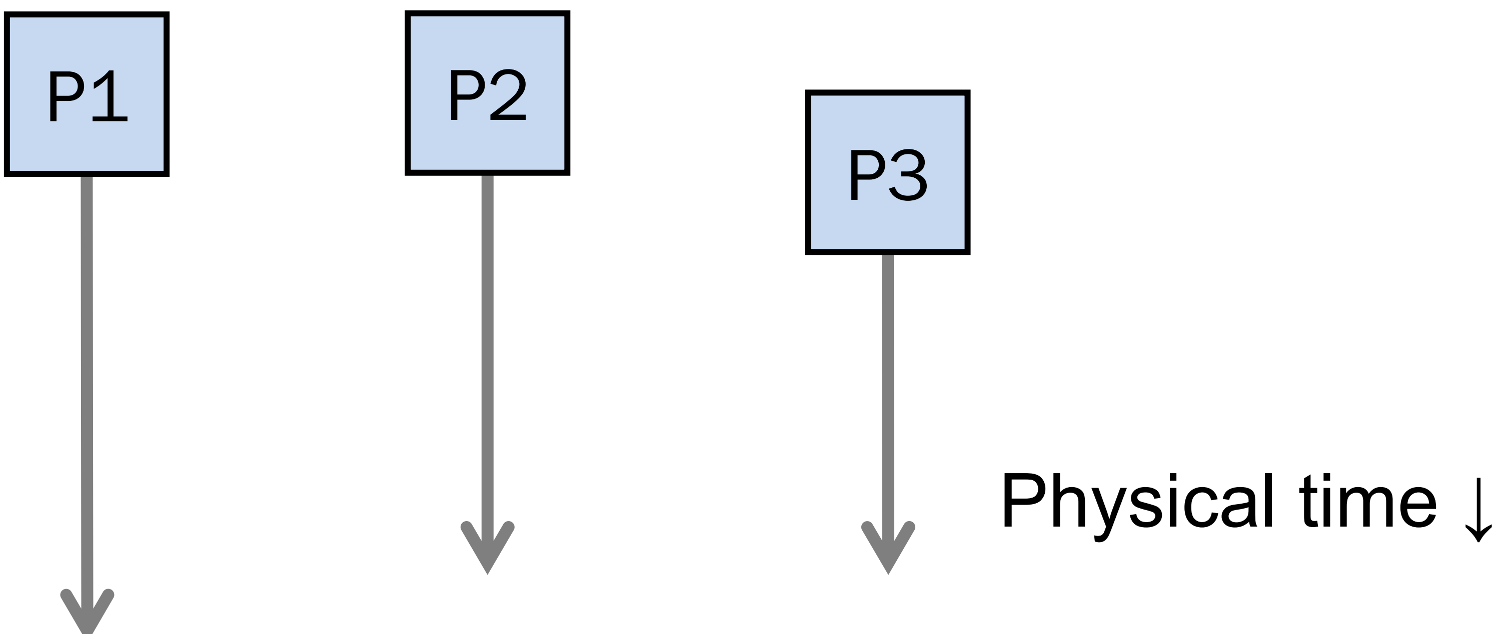
- 2143 < 2144 ➔ make **doesn't call compiler**

No – here timestamps on the events don't respect causal relationship between them!

Defining "happens-before"

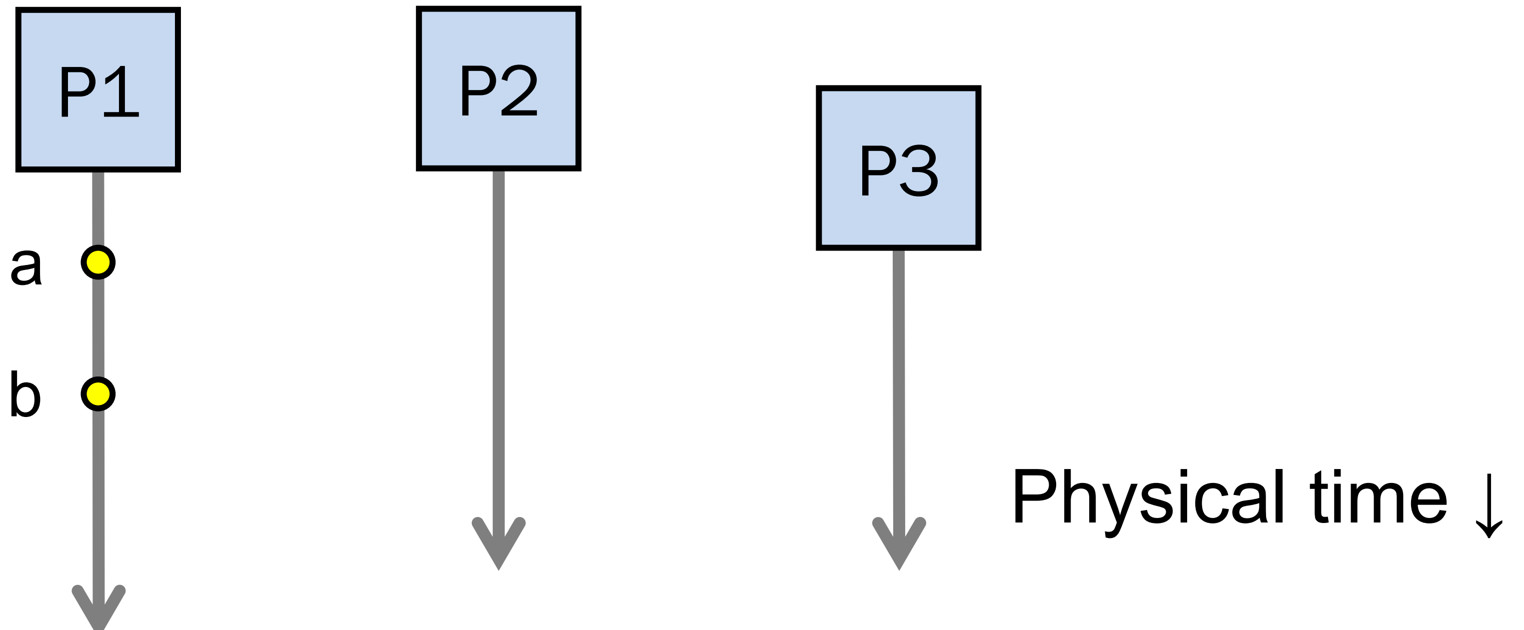
- Consider three processes: **P1**, **P2**, and **P3**

- **Notation:** Event **a** *happens before* event **b (a → b)**



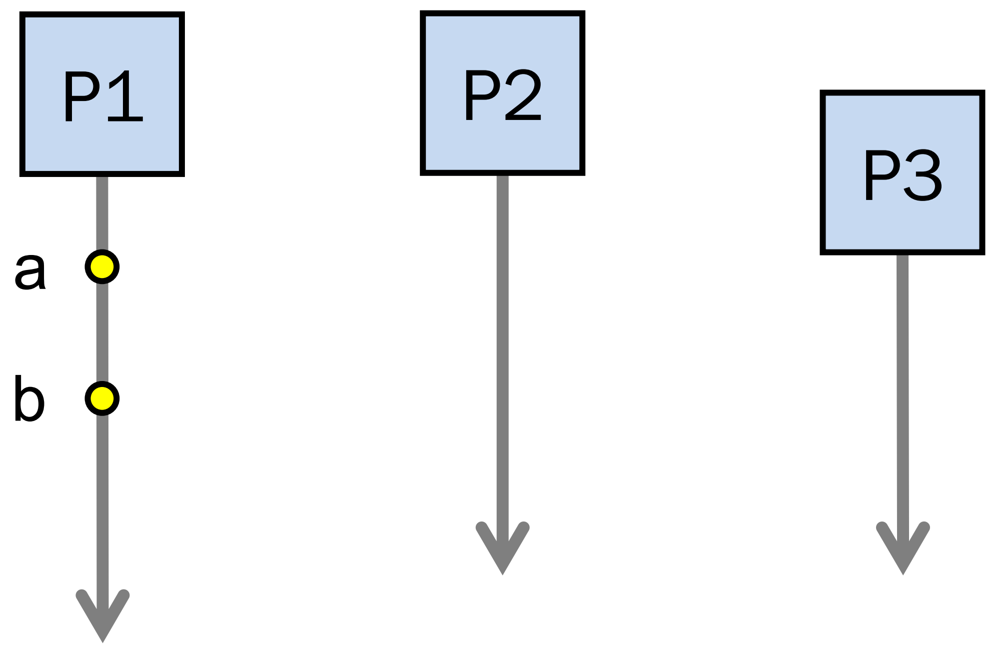
Defining “happens-before”

1. Can observe event order at a single process



Defining "happens-before"

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

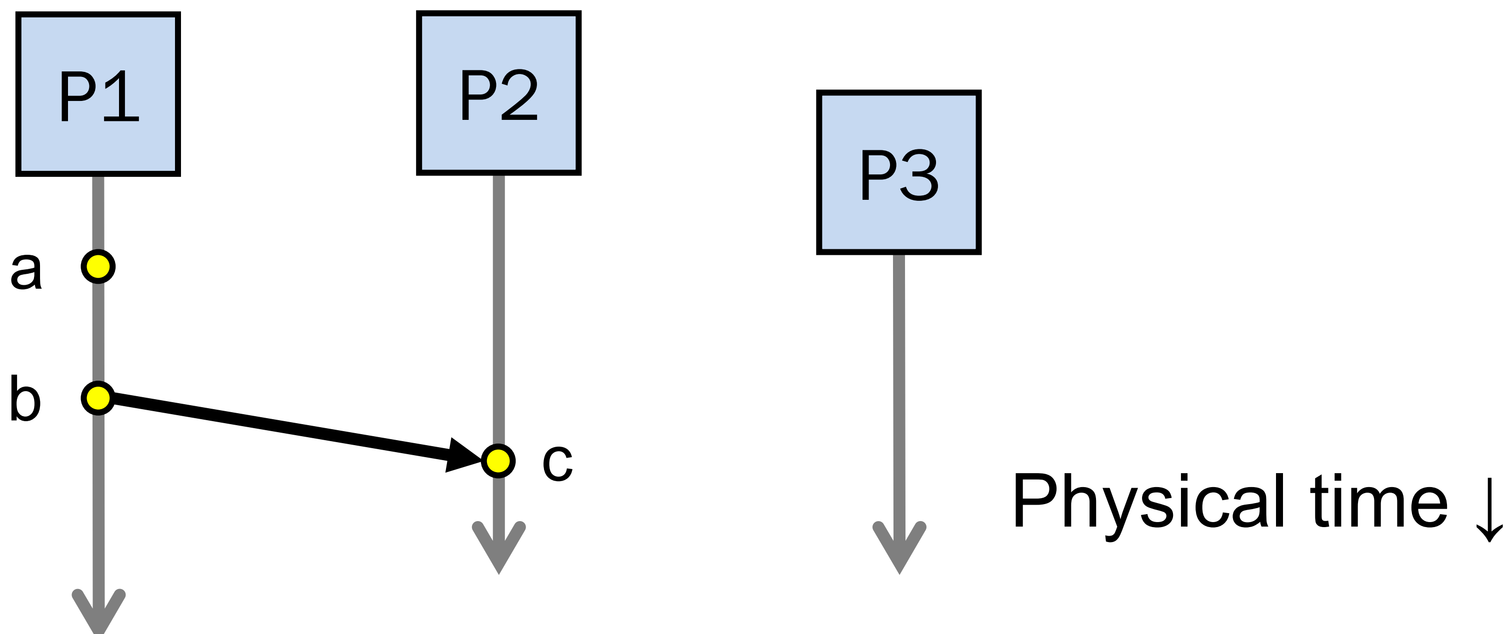


Physical time ↓

Defining "happens-before"

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

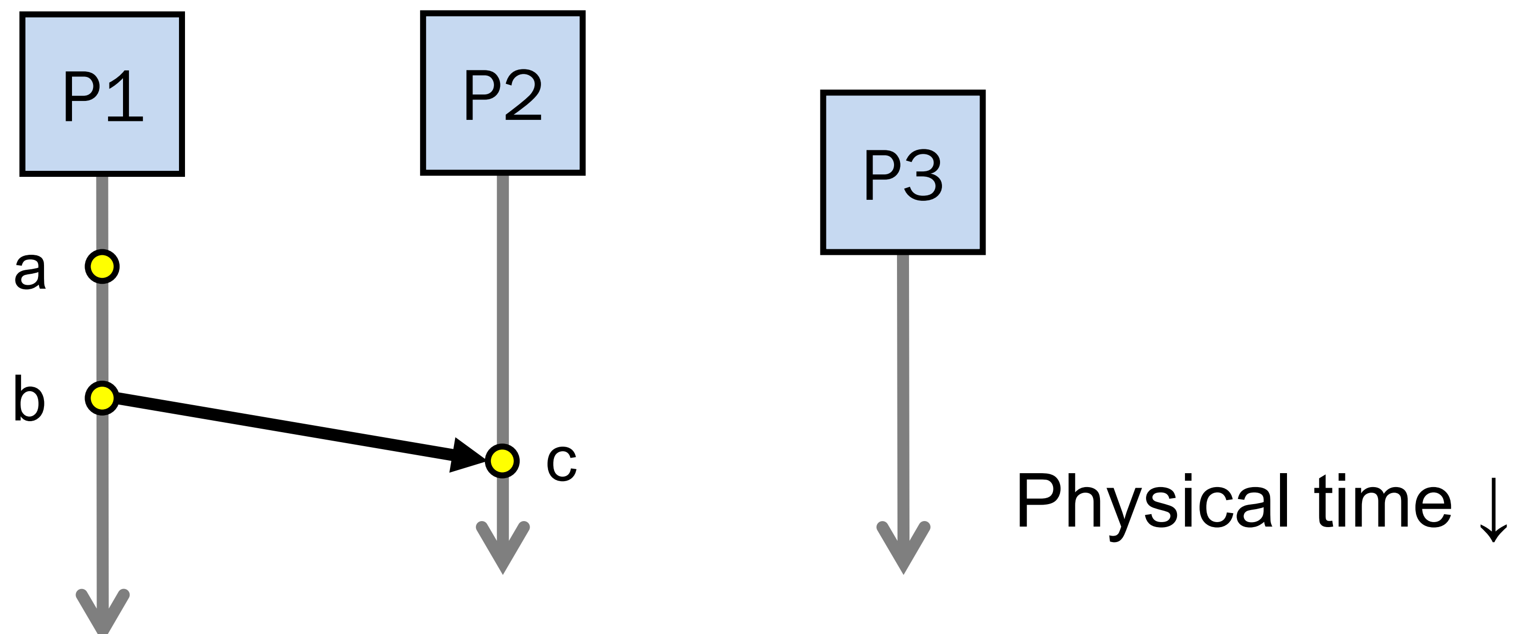
2. Can observe ordering when processes communicate



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**

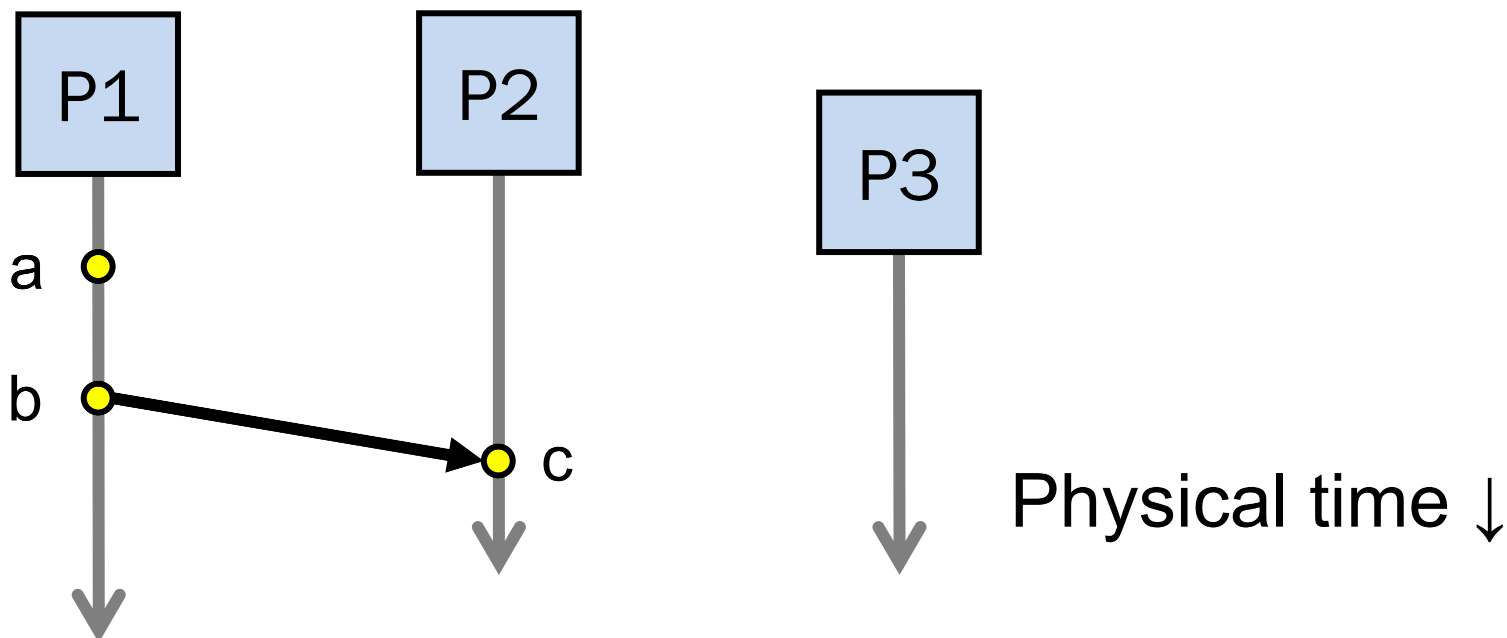


Defining "happens-before"

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

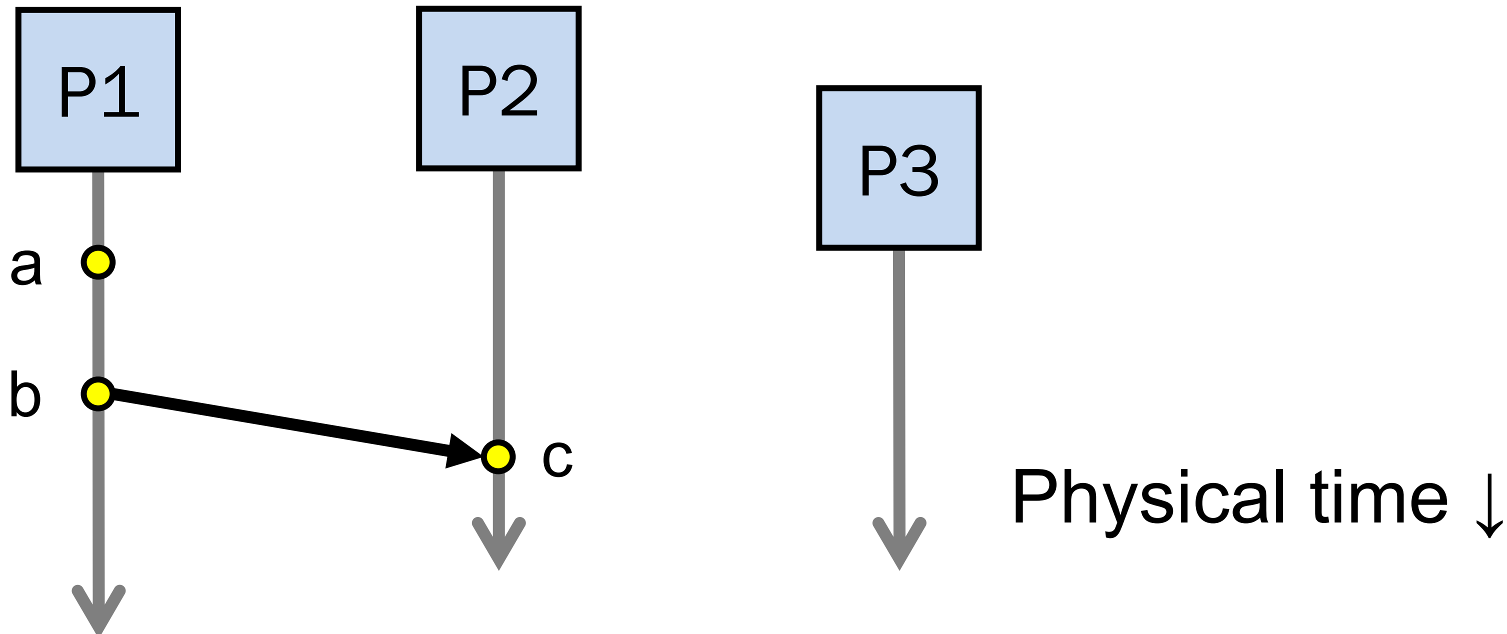


Defining "happens-before"

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

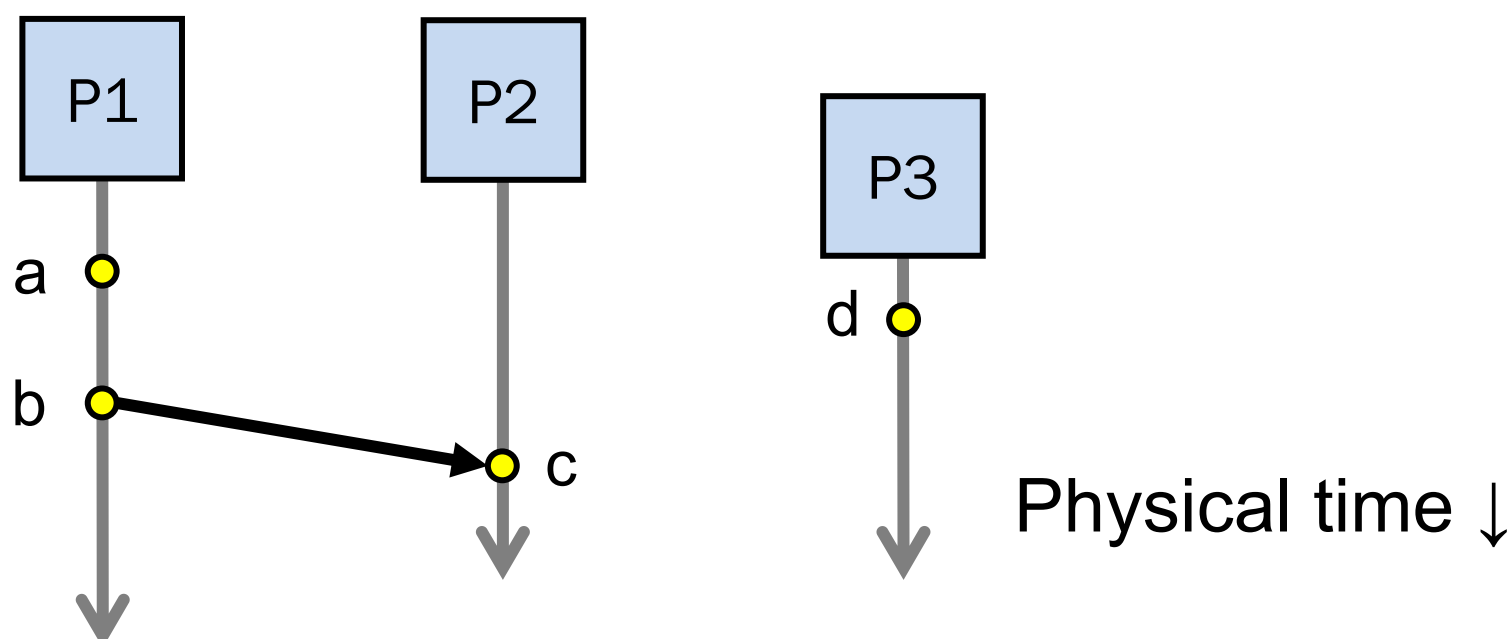
2. If **c** is a message receipt of **b**, then **b** → **c**

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**

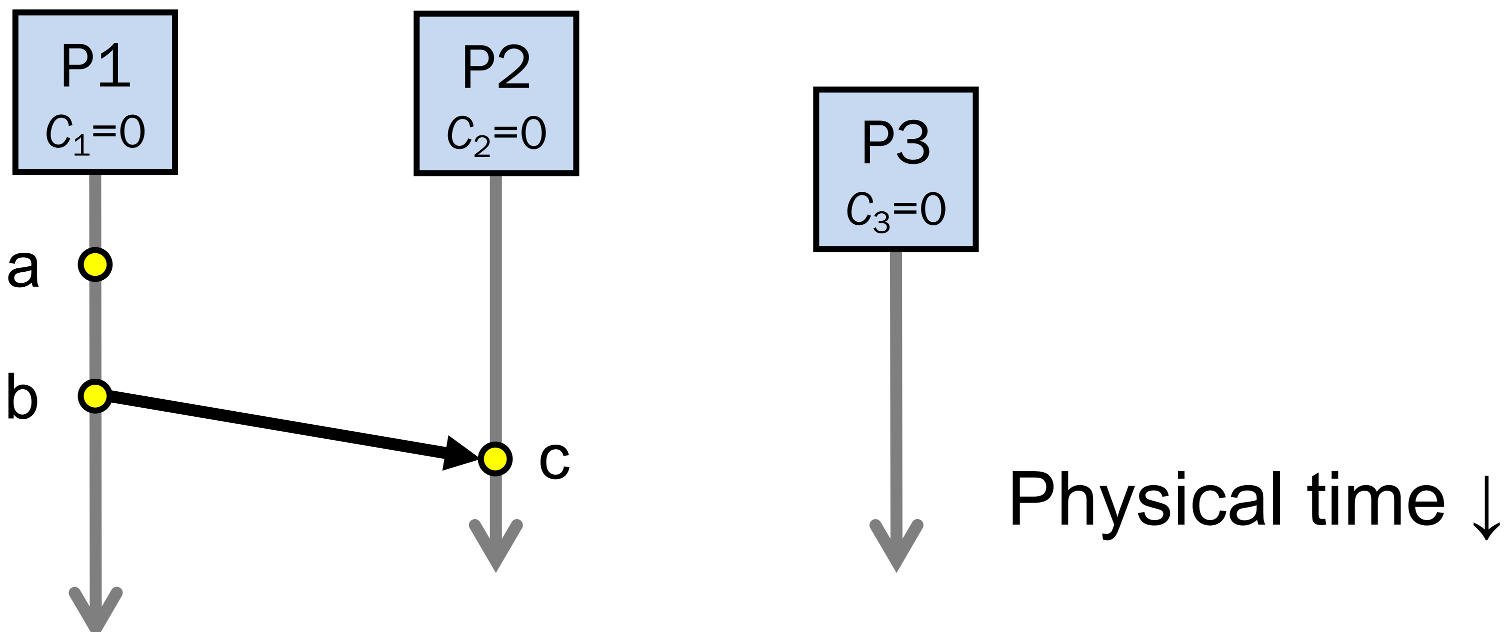
Plan: Tag events with clock times; use clock times to make distributed system correct

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

The Lamport Clock algorithm

- Each process P_i maintains a local clock C_i

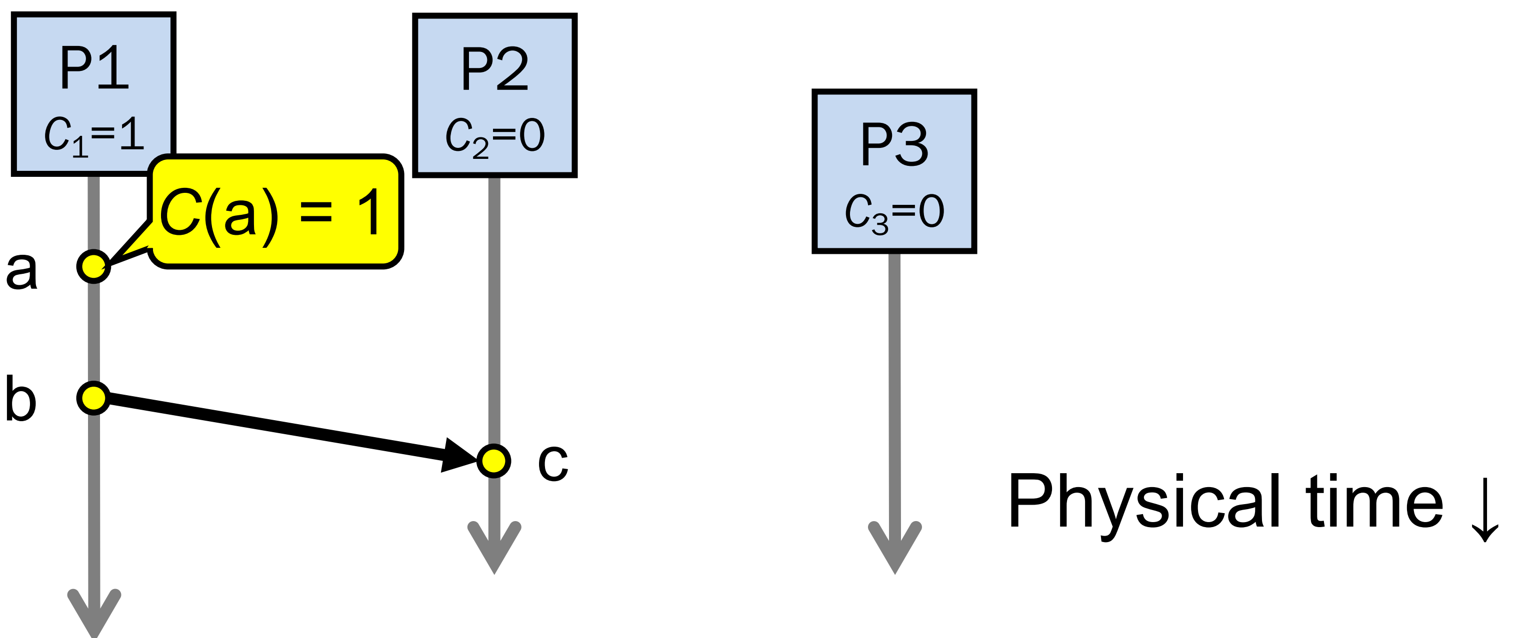
1. Before executing an event, $C_i \leftarrow C_i + 1$



The Lamport Clock algorithm

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

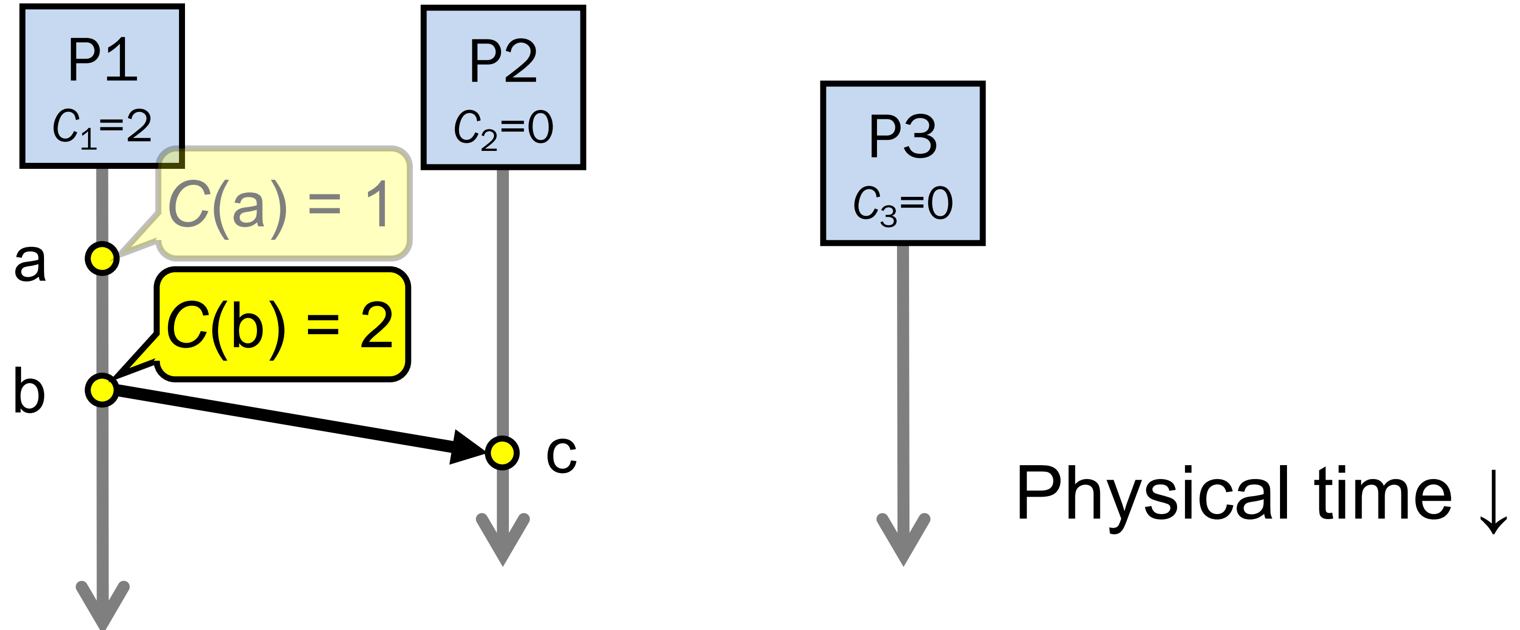
   - Set event time $C(\mathbf{a}) \leftarrow C_i$



The Lamport Clock algorithm

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

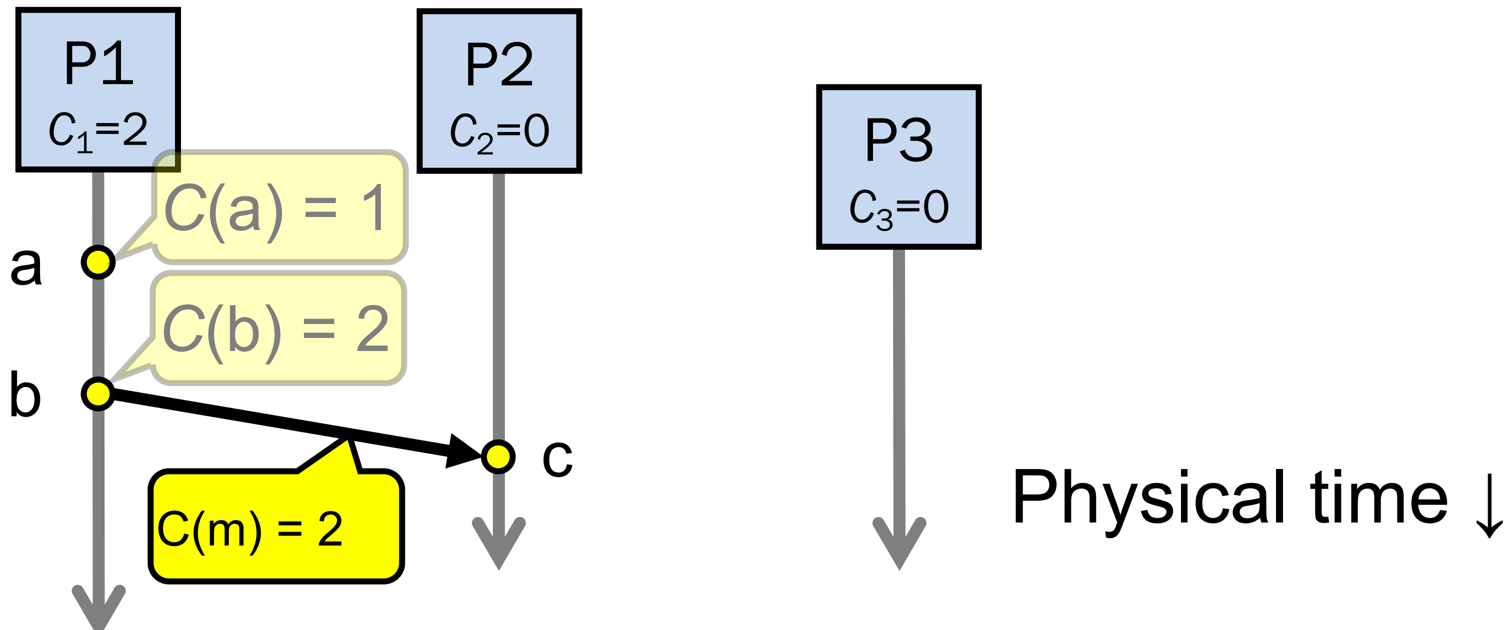
- Set event time $C(\mathbf{b}) \leftarrow C_i$



The Lamport Clock algorithm

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

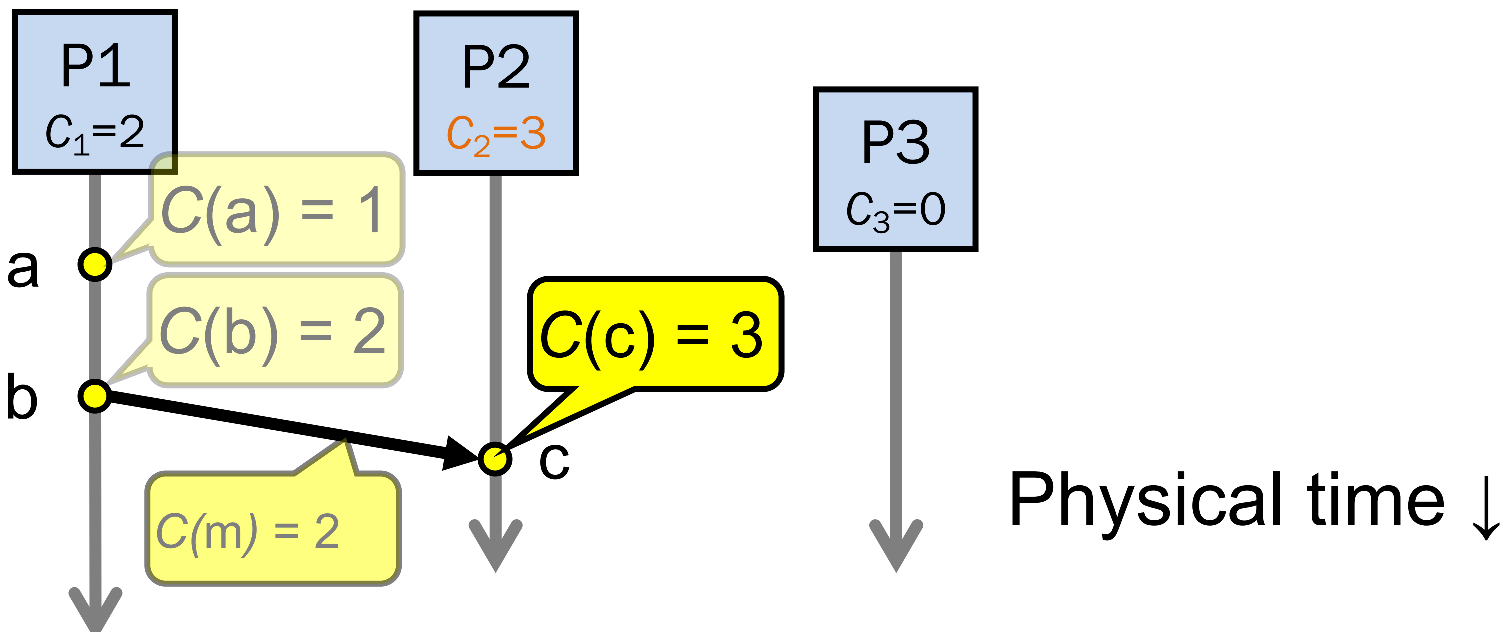
2. Send the local clock in the message **m**



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

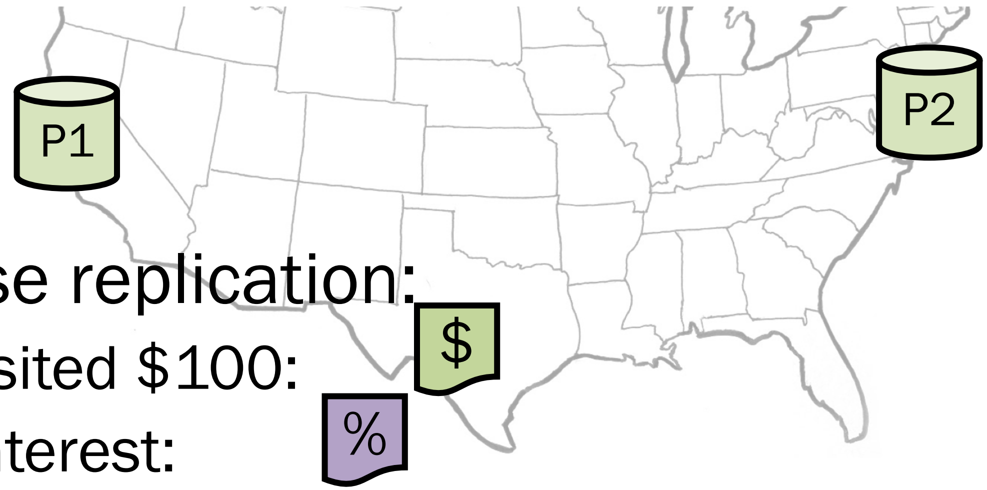
- Break ties by **appending the process number** to each event:

  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 $C(\mathbf{b}) < C(\mathbf{a})$
  - 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)

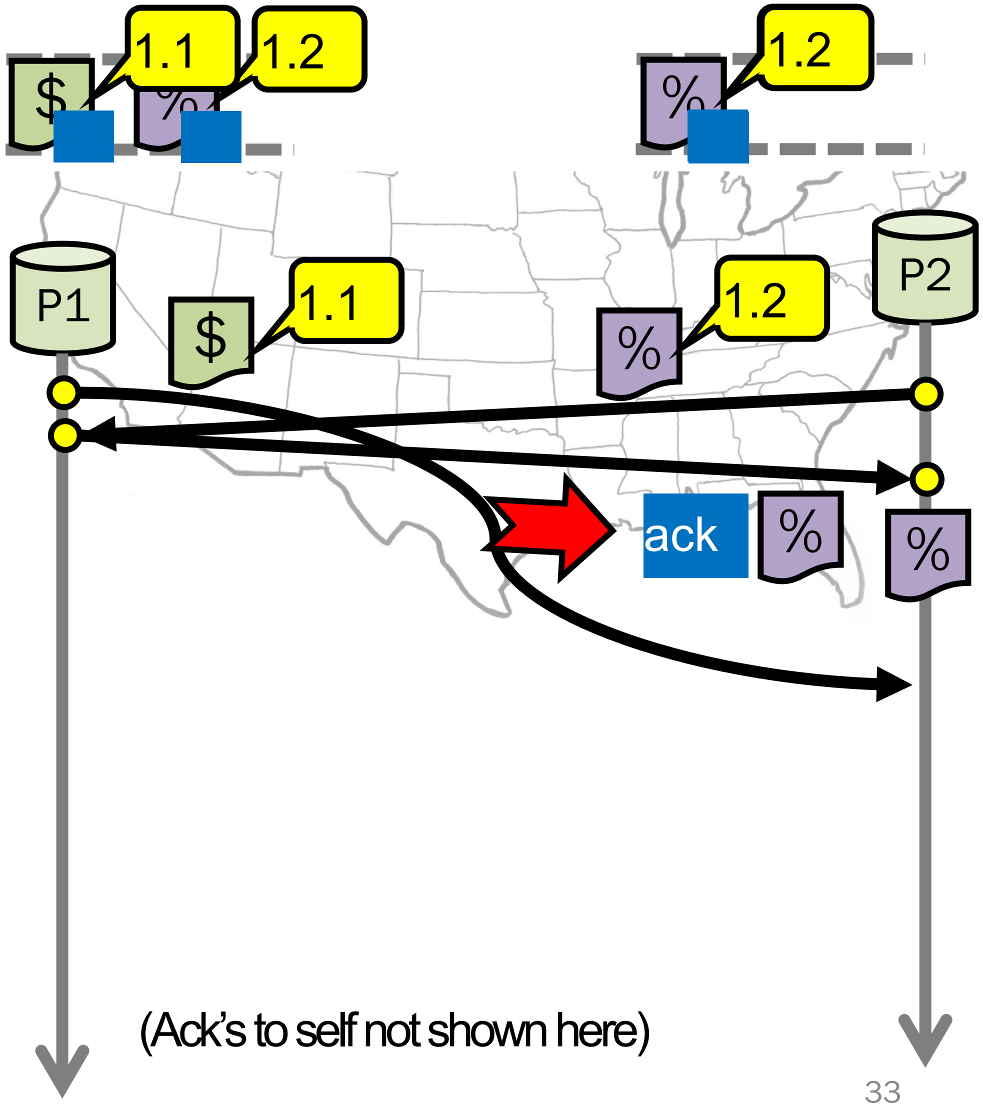
3. On **receiving** an **acknowledgement:**
   - Mark corresponding event *acknowledged* in your queue

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

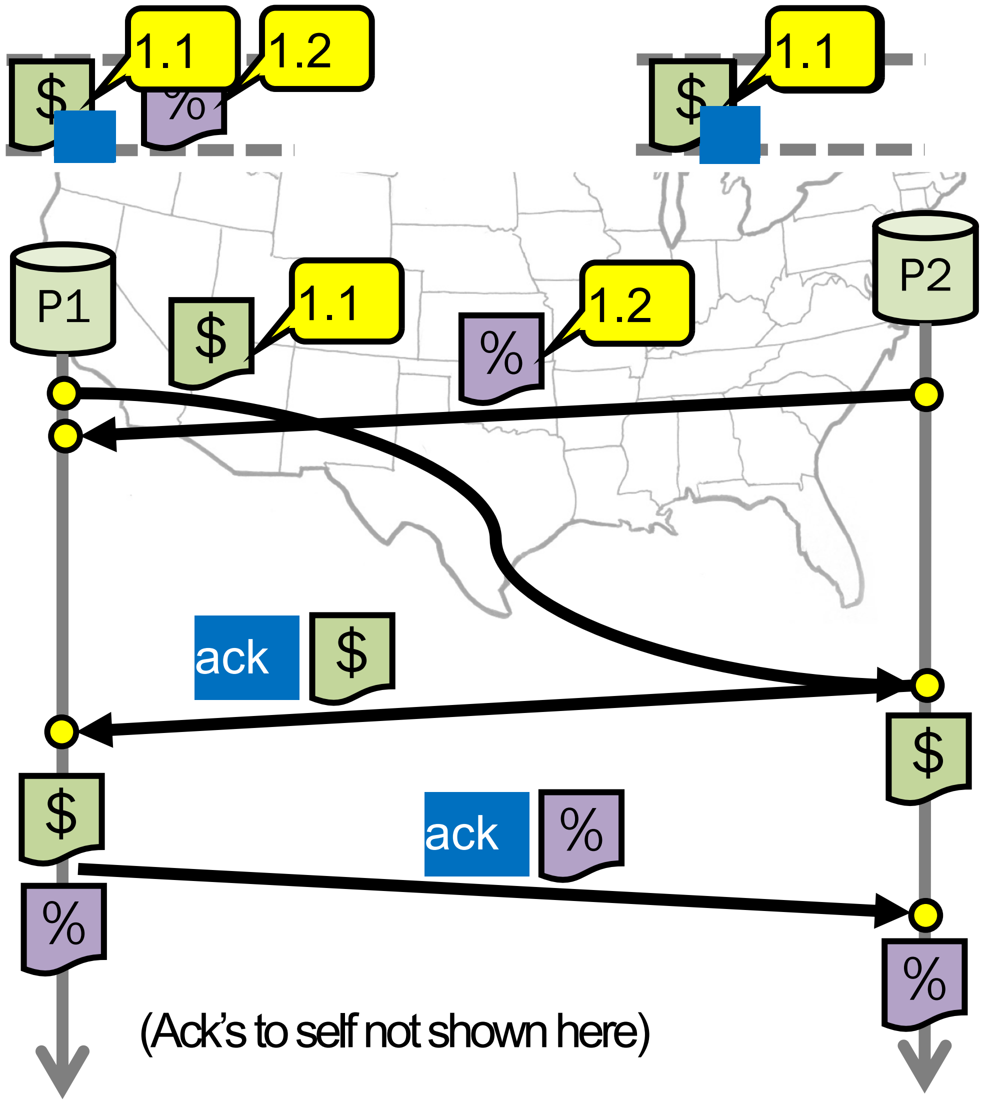
1. On **receiving** an event from **client**, broadcast to others (including yourself)

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]

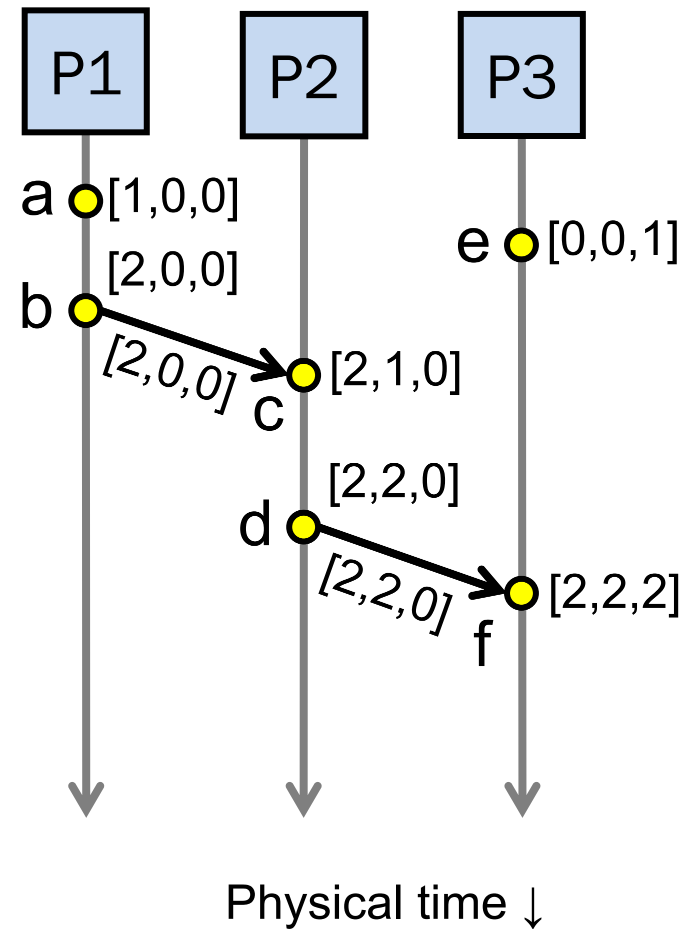
- Two **update rules:**

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

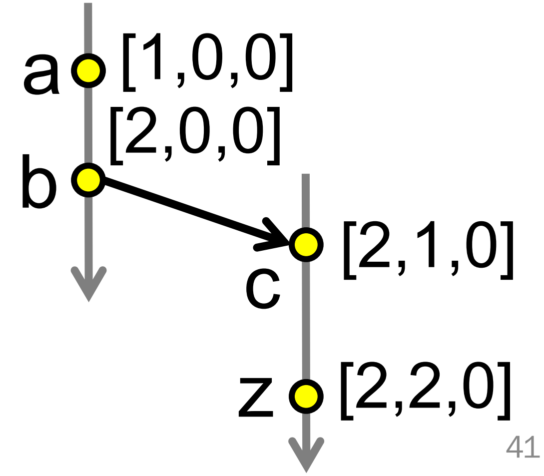


Vector clocks can establish causality

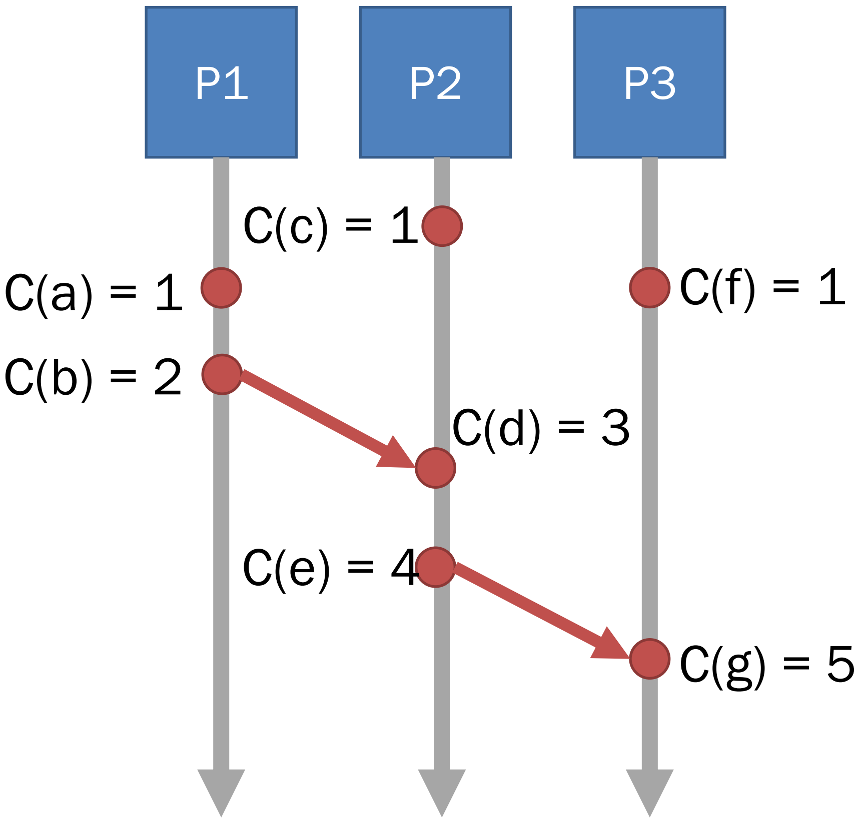
- Rule for comparing vector clocks:
  - V(**a**) = V(**b**) when $\mathbf{a}_k = \mathbf{b}_k$ for all k
  - V(**a**) < V(**b**) when $\mathbf{a}_k \leq \mathbf{b}_k$ for all k and V(**a**) $\neq$ V(**b**)

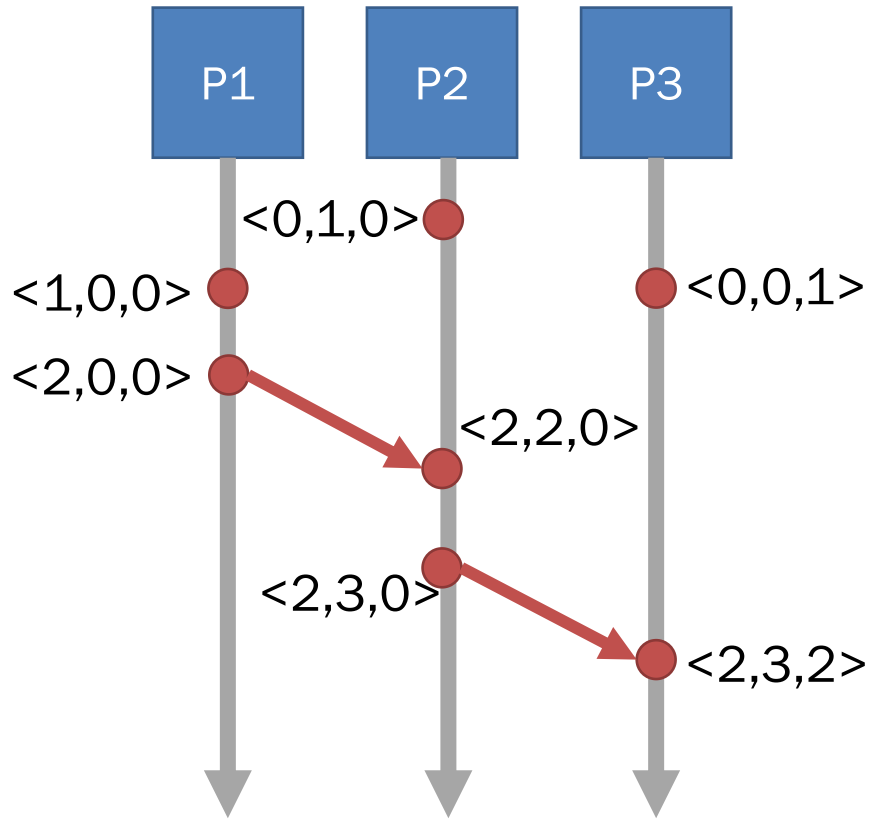
- **Concurrency:** $a \;||\; b$ if $\mathbf{a}_i < \mathbf{b}_i$ and $\mathbf{a}_j > \mathbf{b}_j$, some i, j

- V(a) < V(z) **when** there is a chain of events linked by → between a and z



Lamport vs Vector Clocks





$a \longrightarrow g$? Yes, $V(a) < V(g)$
$f \longrightarrow e$? No, $V(f) \nless V(e)$

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

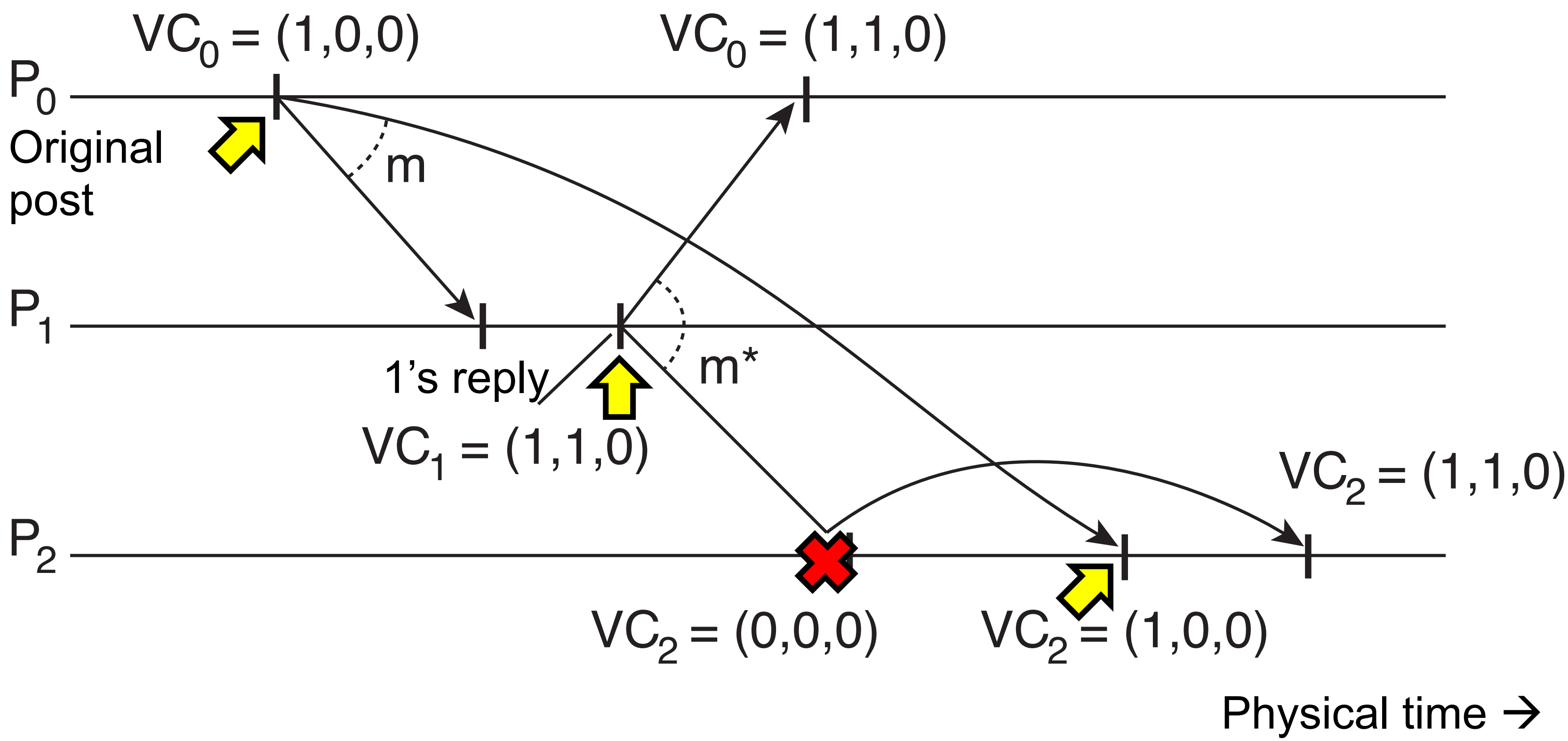
VC application: Causally-ordered bulletin board system

- 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