CS 582: Distributed Systems

Lamport Clocks & Vector Clocks

Dr. Zafar Ayyub Qazi

Fall 2024

Acknowledgements: Slides use some material from Ryan Stutsman (University of Utah), Wyatt Lloyd (Princeton University) and Martin Kleppmann (University of Cambridge)

Reminder

- We have recommended book readings for each lecture

Recap: Driving question

- Can we avoid synchronizing physical time altogether and yet order events in a distributed system?

Recap: Logical Time

- Happens-before relation (a→ b)
  1. Local update: If same process and a occurs before b, then $a \rightarrow b$
  2. Message communication: If b is a message receipt of a, then $a \rightarrow b$
  3. Transitivity: If $a \rightarrow c$ and $c \rightarrow b$, then $a \rightarrow b$

- Lamport Clock algorithm
  - Each process maintains an event counter
  - Before executing an event, increment the counter
  - Whenever a process sends a message, include the counter value
  - When a message is received, set the counter to:
    - $max(local\_counter, received\_counter) + 1$

Recap: Total Order

- Append process number to each event
  1. Process P_i timestamps event e with $C_i(e).i$
  2. $C(a).i < C(b).j$ when:
     - $C(a) < C(b)$, *or* $C(a) = C(b)$ *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

Recap: Limitation of 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 potential causal relationships between events

Today's Lecture

- Application of Lamport clocks
  - Totally-ordered multicast for multi-site database replication
- Vector Clocks
  - What are vector clocks?
  - How they can be implemented?
  - What do they offer in addition to Lamport clocks?
- Application of Vector Clocks
  - Causally ordered Slack-like Application

Specific learning outcomes

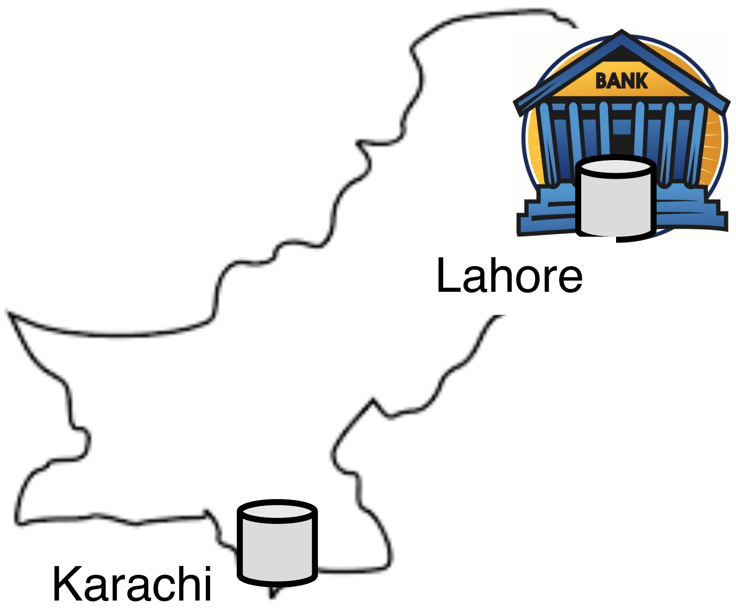
By the end of today's lecture, you should be able to:

- ❑ Explain how Lamport clocks can be used to provide totally ordered communication
- ❑ Apply Lamport clocks to solve multi-site database replication problem
- ❑ Analyze the limitations of Lamport Clocks
- ❑ Explain how Vector clocks work
- ❑ Explain how Vector clock timestamps can be compared
- ❑ Analyze how Vector clock timestamps can be used to infer the happens-before relation
- ❑ Apply vector clocks to provide causally ordered communication

Example: Multi-site database replication

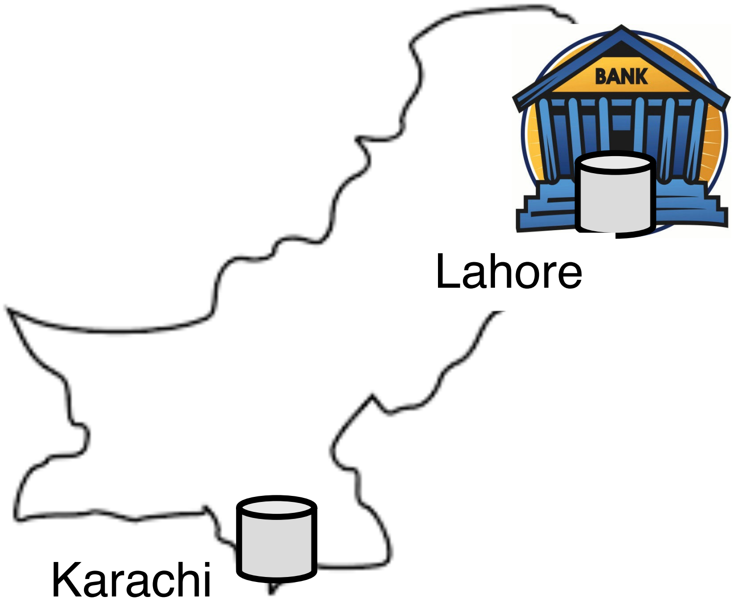
Example: Multi-site database replication

- A Lahore-based bank wants to replicate an accounts database to improve its query performance
- The bank replicates the database, keeps one copy in Lahore, one in Karachi



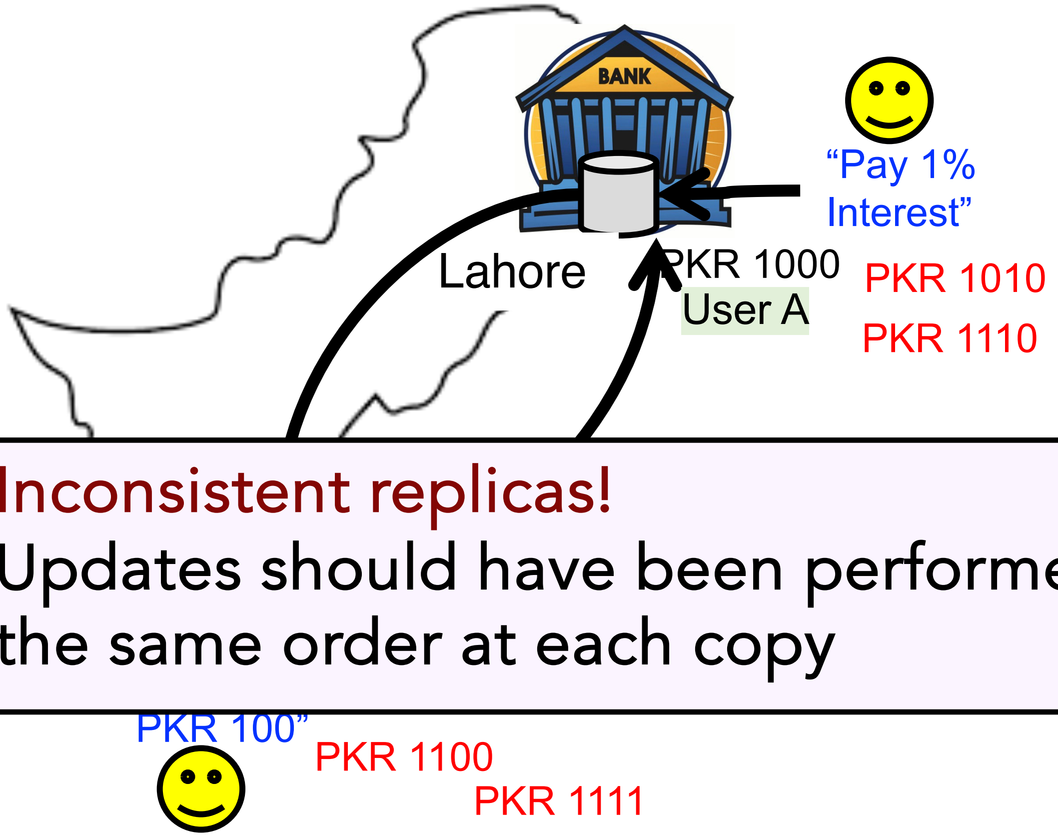
Example: Multi-site database replication protocol

- Replicate the database, keep one copy in Lahore, one in Karachi
  - A query is always forwarded to the nearest copy
  - Then send update to the other copy



The consequences of concurrent updates

- Replicate the database, keep one copy in KHI, one in LHR
  - A query is always forwarded to the nearest copy
  - Then send update to the other copy



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

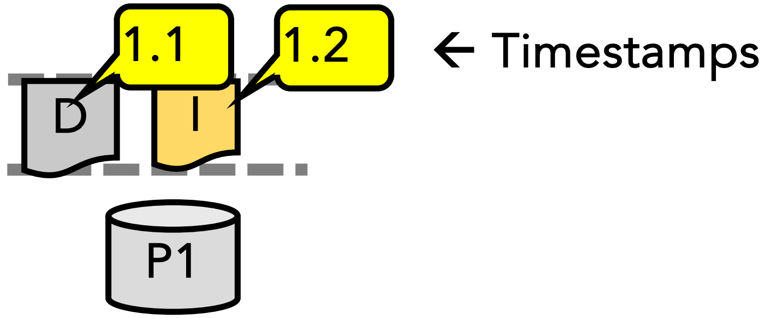
Class Exercise

- Devise a solution to make multi-site updates consistent using Lamport Clocks
- Please consult your neighbors
- You have 10mins to devise a solution
- Assume
  - There are no node failures
  - There are no message drops

Key solution idea

- Goal: All sites apply updates in the same Lamport clock order
- Client sends update to one replica site j
  - Replica assigns it Lamport timestamps C_j and concatenates site id: $C_j.j$
  - Sends a message with $C_j.j$ to all sites (including itself)
- Key idea: Place events into a sorted local queue
  - Sorted by increasing Lamport timestamps

Example: P1's
local queue:



Lets look at a complete solution

- Chapter 6.2 of the course textbook also provides an explanation for the example

Totally-Ordered Multicast (Almost Correct)

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

2. On receiving or processing an update:
   a) Add it to your local queue
   b) Broadcast an acknowledgement message to every replica (including yourself)

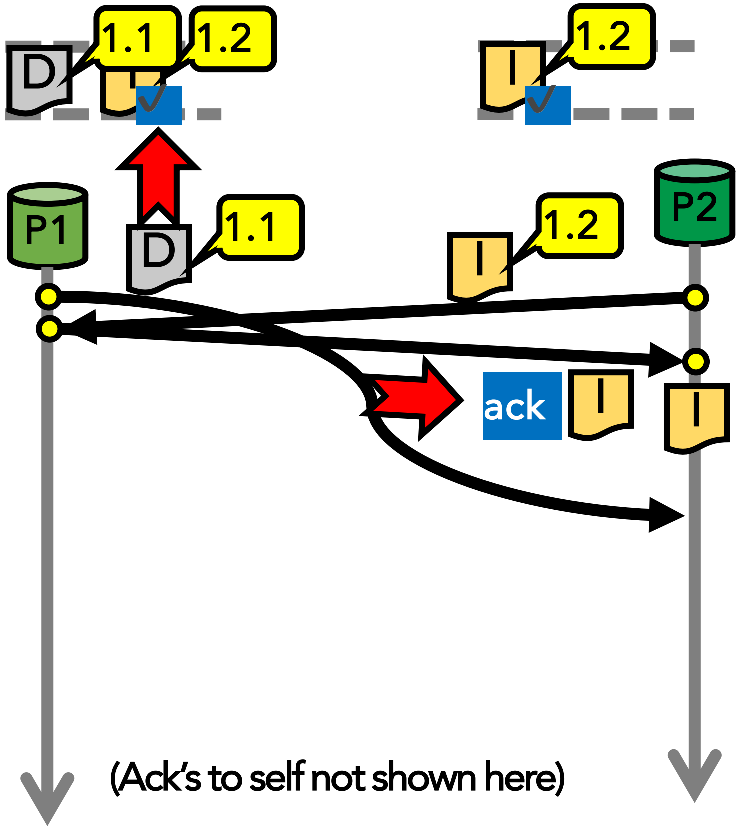
3. On receiving an acknowledgement:
   - Mark corresponding update acknowledged in your queue

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

Totally-Ordered Multicast (Almost Correct)

- P1 queues D, P2 queues I
- P1 queues and ack's I
  - P1 marks I fully ack'ed
- P2 marks I fully ack'ed

✗ P2 processes I



Totally-Ordered Multicast (Correct Version)

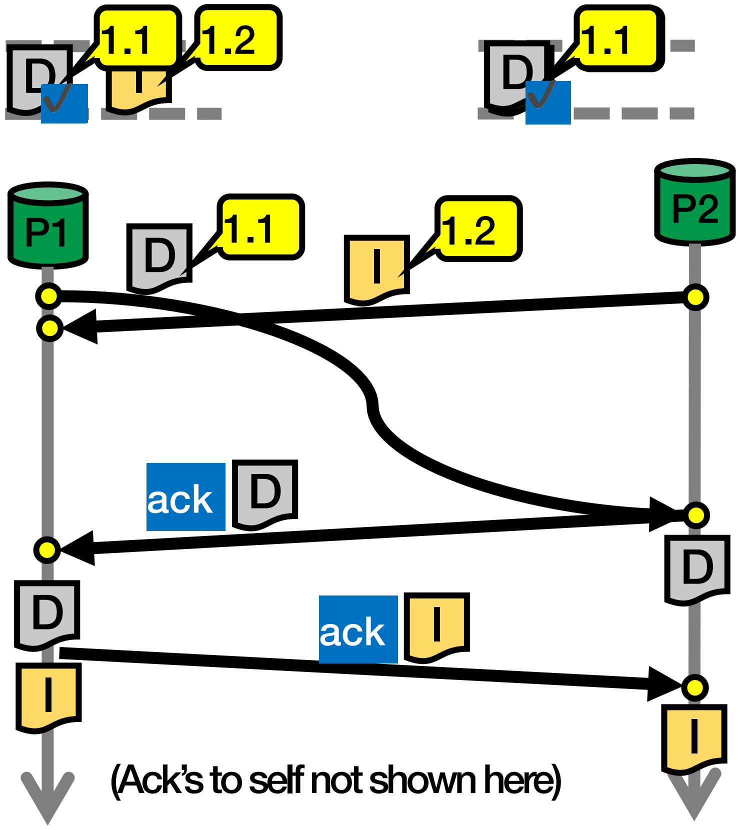
1. On receiving an update from client, broadcast to others (including yourself)

2. On receiving or processing an update:
   a) Add it to your local queue
   b) Broadcast an acknowledgement message to every replica (including yourself) only from head of queue

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

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

Totally-Ordered Multicast (Correct Version)



So are we done?

- Does totally-ordered multicast solve the problem of multi-site replication in general?
- Our protocol assumed:
  - No node failures
  - No message loss
- All to all communication may not not scale
- Waits forever for message delays (performance?)

Rest of the lecture: Inferring potential causality

- Given two timestamps C(a) and C(z), want to know whether there's a chain of events linking them:

$$a \rightarrow b \rightarrow \ldots \rightarrow y \rightarrow z$$

Driving Question

- How can we design logical clocks which can allow us to infer potential causality?
  - More precisely, whether a → b?

Vector Clocks

First described in a paper by Barbara Liskov and Rivka Ladin

Vector Clocks

- A Vector Clock is a vector of integers, one entry for each process in the system

- Label each event e with a vector $V(e) = \langle c_1, c_2, \ldots, c_n \rangle$
  - Where c_i is the count of events in process i that "happen-before" e
  - And we have n processes in the system

- Initially, all vectors are $\langle 0, 0, \ldots, 0 \rangle$

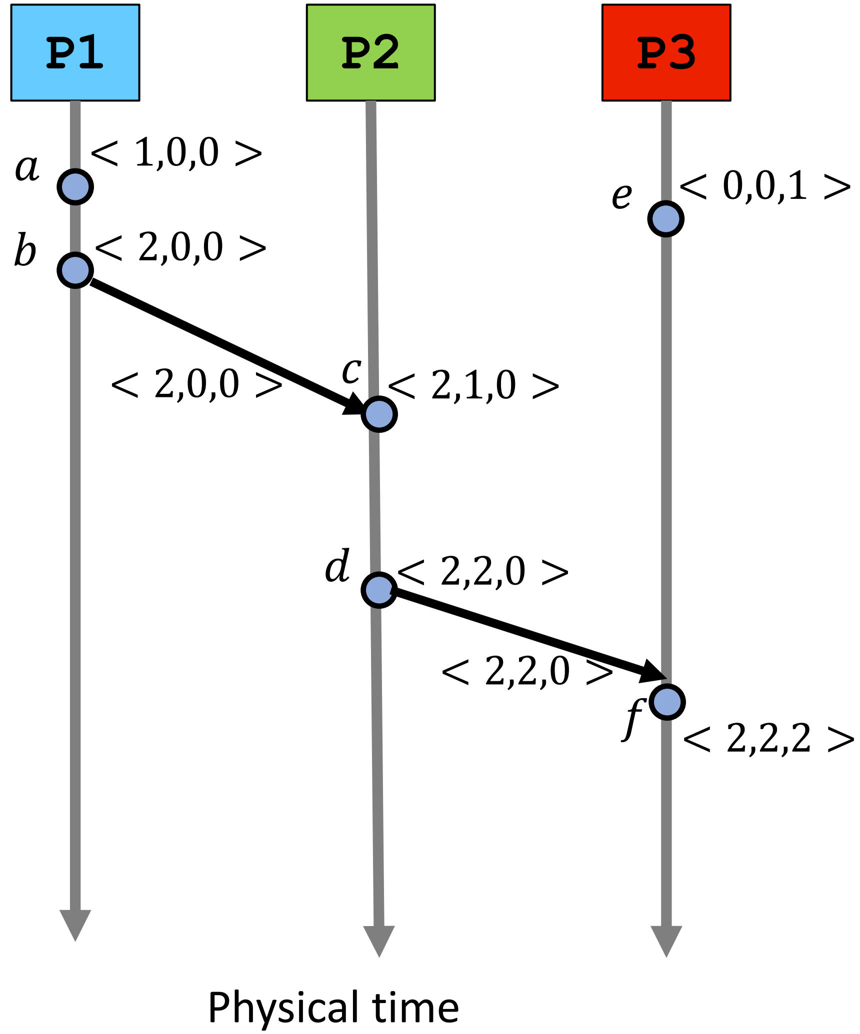
Rules for updating Vector Clocks

- Local update rule:
  - For each local event on process i, increment local entry c_i (by 1)
- Message rule:
  - If process j receives message with vector $\langle d_1, d_2, \ldots, d_n \rangle$:
    - Set each local entry $c_k = \max\{c_k, d_k\}$
    - Increment local entry c_j (by 1)

Vector Clock: Example

Assumptions: P1, P2, and P3 are single threaded processes and on different nodes

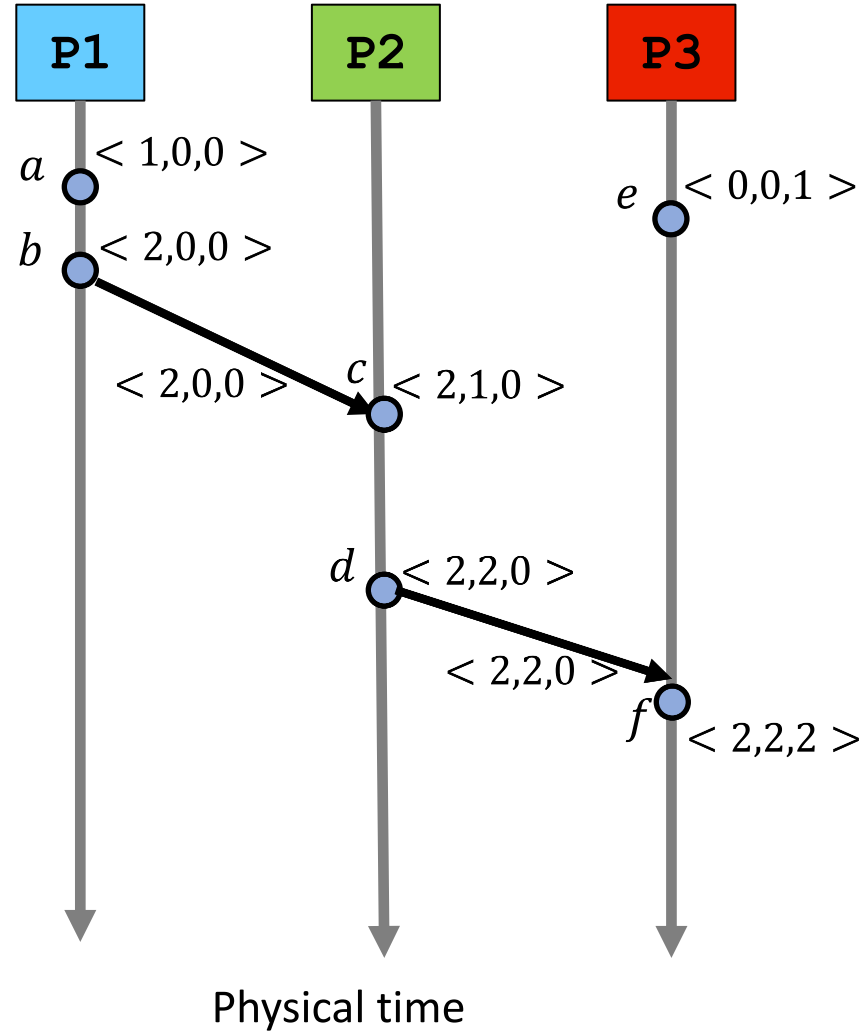
- All counters start at <0,0,0>
- Apply local update rule
- Apply message rule
  - Local vector clock sent in the the messages



What do these Vector timestamps represent?

- The vector timestamp of an event d represents a set of events: d and its potential causal dependencies
  - $\{d\} \cup \{\, a \mid a \rightarrow d\}$

- For example, <2,2,0> represents the first two events from P1, the first two events from P2, and no events from P3



Vector Clock Orderings

- Rules for comparing vector timestamps:
  - $V(a) = V(b)$ iff $a_k = b_k$ for all k
  - $V(a) < V(b)$ iff $a_k \leq b_k$ for all k and $V(a) \neq V(b)$
  - $V(a) \,||\, V(b)$ iff $a_i < b_i$ and $a_j > b_j$, for some i, j
- Properties of this order
  - $(V(a) < V(b)) \Leftrightarrow (a \rightarrow b)$
  - $(V(a) = V(b)) \Leftrightarrow (a = b)$
  - $(V(a) \,||\, V(b)) \Leftrightarrow (a \,||\, b)$

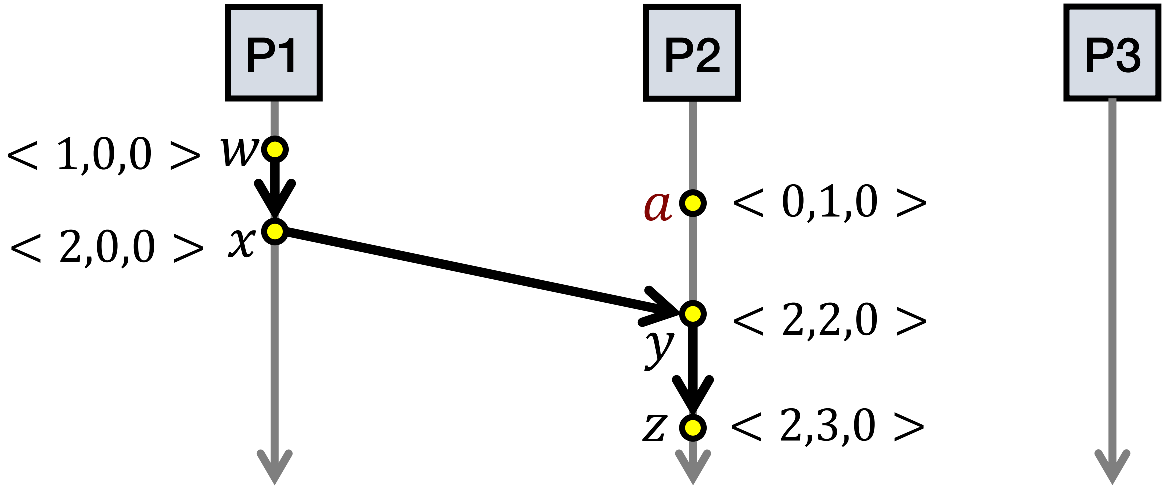
Vector Clocks Capture Potential Causality

Vector Clocks Capture Potential Causality

- V(w) < V(z) then there is a chain of events linked by

  happens-before (→) between w and z

  V(a) || V(w) then there is **no** such chain of events between a and w



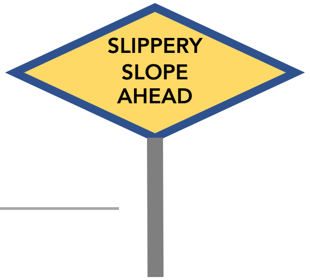
Two events **a, z**

Lamport Clocks: **C(a) < C(z)**
Conclusion: **None**

Vector Clocks: **V(a) < V(z)**
Conclusion: **a→ …→ z**

Vector clock timestamps precisely capture the happens-before relation

Causally-ordered Slack-like Application

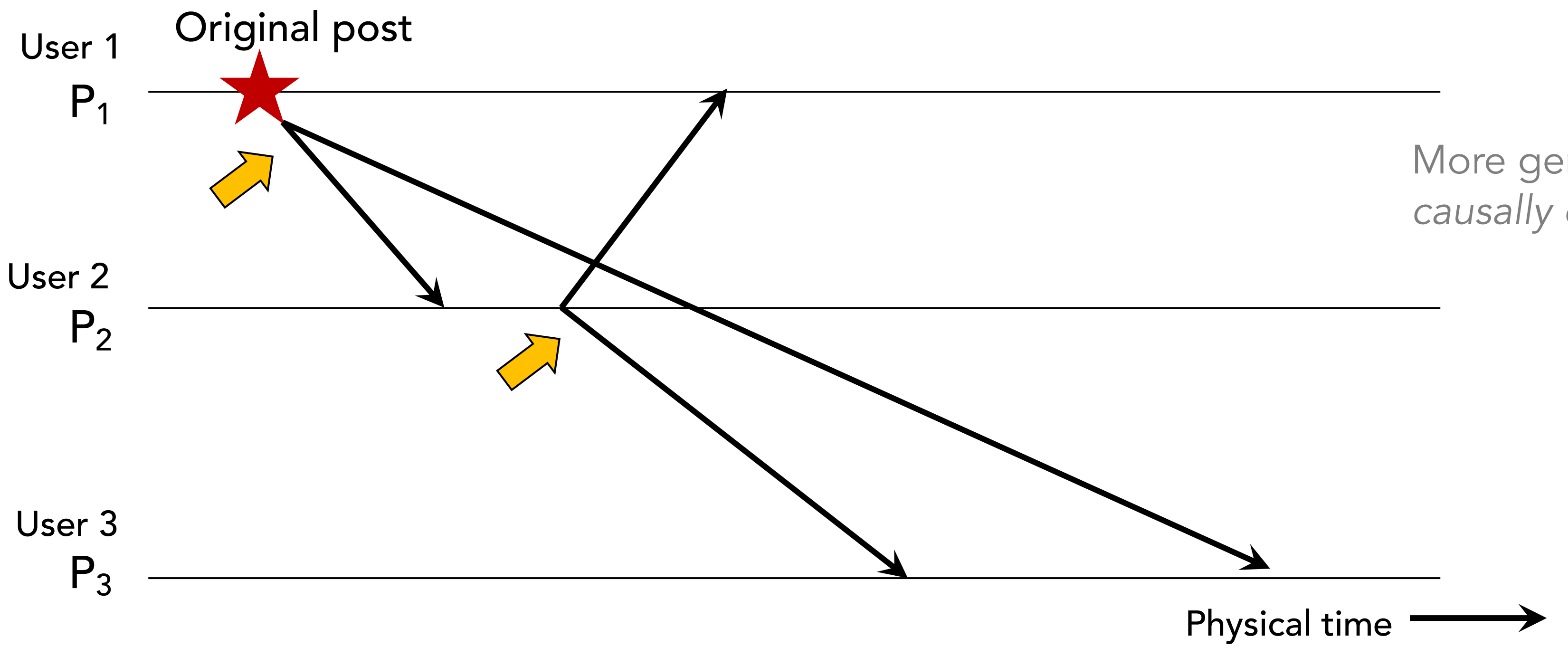


- Slack-like application
  - Broadcast posts to all other users

- **Goal:** No user should *display* a response post before the corresponding original message post

- Deliver a message only after all messages that might causally precede it have been delivered
  - Otherwise, the user would see a reply to a message they could not find

Causally-ordered Slack-like App

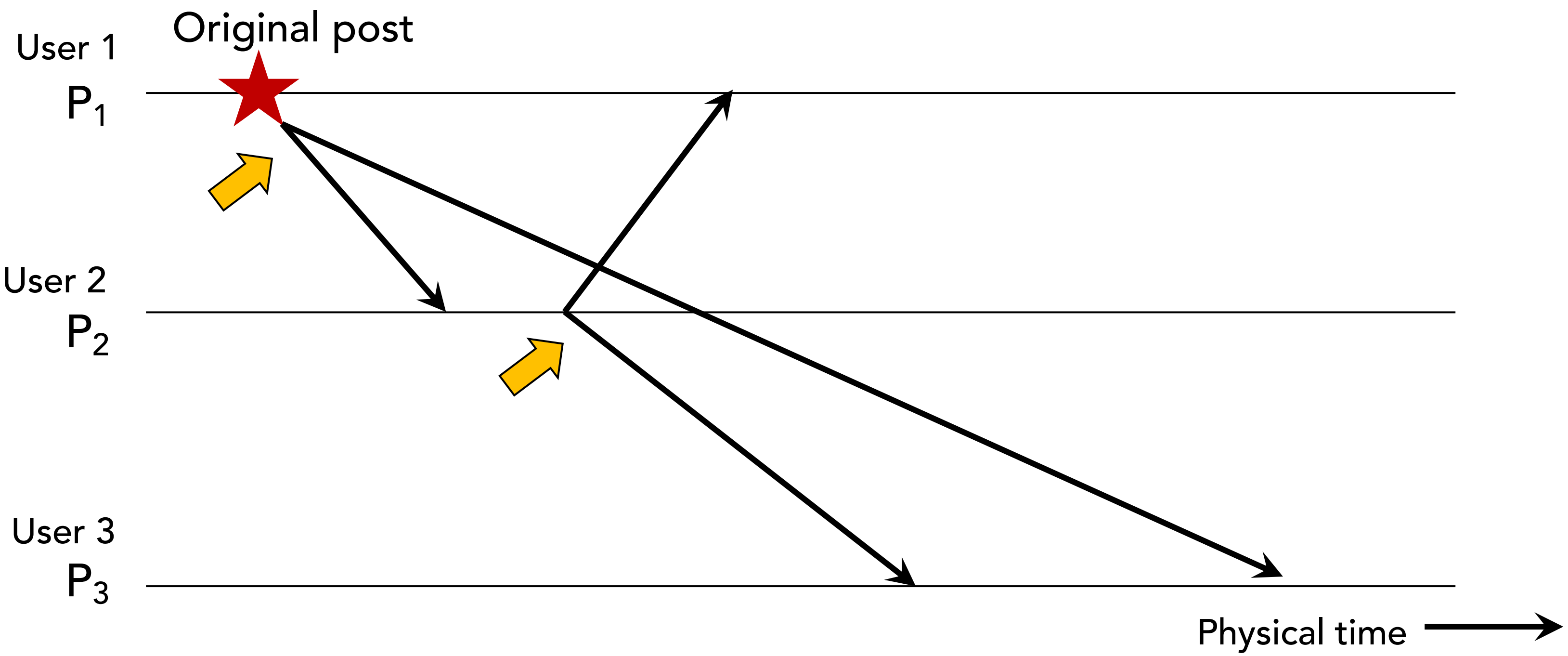


More generally this is referred to as providing *causally ordered communication*

- User 1 posts
- User 2 replies to User 1's post
- User 3 observes

Let's try to design a solution using vector clocks

Causally-ordered Slack-like App



- User 1 posts
- User 2 replies to User 1's post
- User 3 observes

Next Lecture

- Leader Elections