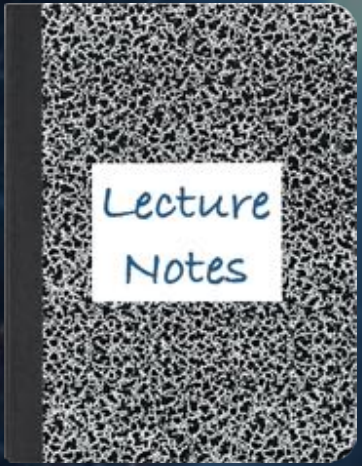


CS 417 – DISTRIBUTED SYSTEMS

Week 3: Part 2

Logical Clocks

Paul Krzyzanowski

Logical clocks

Assign sequence numbers to messages

- All cooperating processes can agree on <u>order of events</u>
- vs. *physical clocks*: report time of day

Assume no central time source

- Each system maintains its own local clock
- No total ordering of events
  - No concept of *happened-when*

- **Assume multiple actors (processes)**
  - Each process has a unique ID
  - Each process has its own incrementing counter

Happened-before

Lamport's "happened-before" notation

a $\rightarrow$ ***b*** event ***a*** happened before event ***b***

e.g.: ***a:*** message being sent, ***b***: message received

Transitive:

if ***a*** $\rightarrow$ ***b*** and ***b*** $\rightarrow$ ***c*** then ***a*** $\rightarrow$ ***c***

Logical clocks & concurrency

Assign a "clock" value to each event

- if $a \rightarrow b$ then $clock(a) < clock(b)$ since time cannot run backwards

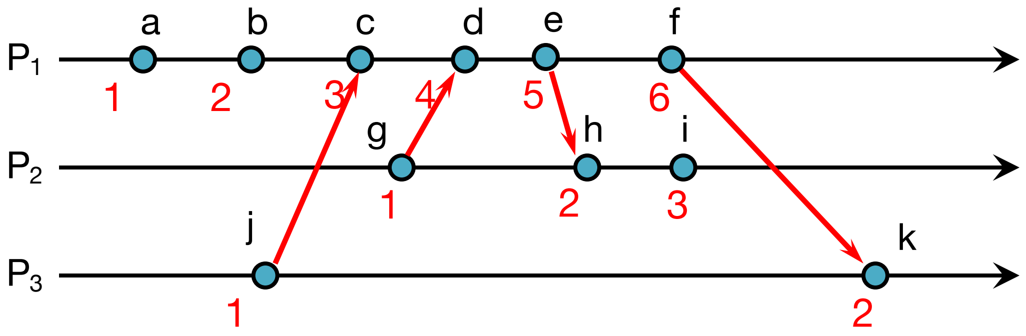
If *a* and *b* occur on different processes that do not exchange messages, then neither $a \rightarrow b$ nor $b \rightarrow a$ are true

- These events are **concurrent**
- Otherwise, they are **causal**

Event counting example

- Three systems: P_1, P_2, P_3
- Events *a*, *b*, *c*, …
- Local event counter on each system
- Systems occasionally communicate

Event counting example



Bad ordering:

e → h but 5 ≥ 2

f → k but 6 ≥ 2

Lamport Timestamps

- Each process has its own clock (sequence #)
- Clock is incremented before each event
- Each message carries a timestamp of the sender's clock
- When a message arrives:

    if receiver's *clock* ≤ *message_timestamp*

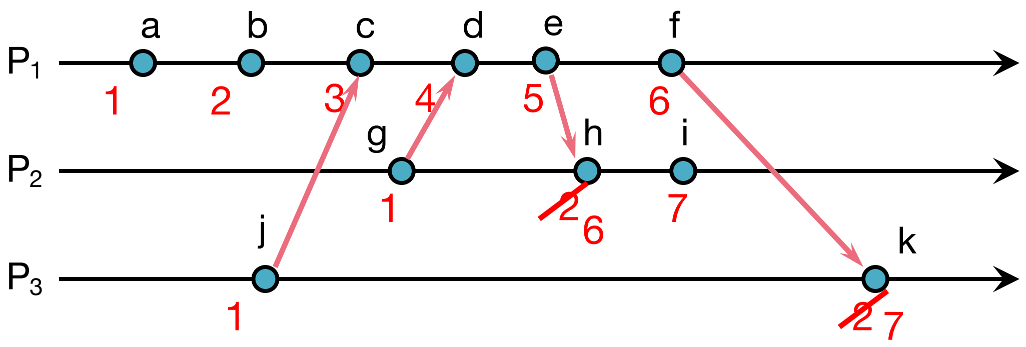
        set system clock to *(message_timestamp + 1)*

    set event timestamp to the system's clock

Lamport timestamps allow us to maintain time ordering among related events ⇒ **Partial ordering**

Event counting example

Applying Lamport timestamps



We have good ordering where we used to have bad ordering:

e → h and 5 < 6

f → k and 6 < 7

Summary

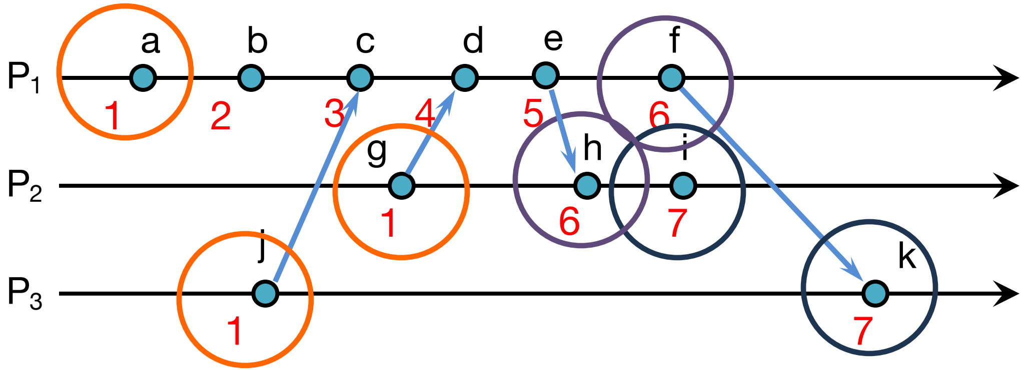
- Lamport timestamps need a monotonically increasing software counter

- Incremented when events that need to be timestamped occur
  - Every message that is sent contains the timestamp
  - Every received message sets the clock to ***max*(msg_timestamp + 1, clock)**
  - The event is associated with the value of the clock (Lamport timestamp)

- For any two events, where $a \rightarrow b$:

  $L(a) < L(b)$

Problem: Identical timestamps



a→*b*, *b*→*c*, … : local events sequenced

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

Concurrent events (e.g., *b* & *g*; *i* & *k*) *<u>may</u>* have the same timestamp … or not

Unique timestamps (total ordering)

We can force each timestamp to be unique

- Define global logical timestamp (T_i, i)
  - T_i represents local Lamport timestamp
  - i represents process number (globally unique)
    - e.g., (host address, process ID)
- Compare timestamps:

$$(T_i, i) < (T_j, j)$$

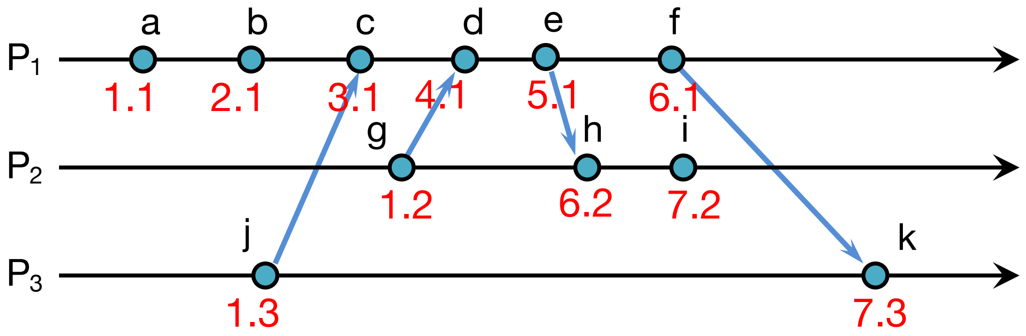
if and only if

$$T_i < T_j \text{ or}$$

$$T_i = T_j \text{ and } i < j$$

Does not necessarily relate to actual sequence of events

Unique (totally ordered) timestamps



Problem: Detecting causal relations

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**

Vector clocks are a way to prove the sequence of events by keeping version history based on each process that created an event

Example

- Group of processes: *Alice, Bob, Cindy, David*
- They send messages to decide: *"what food should we eat?"*
- Each process keeps a <u>local</u> counter

Alice writes the value & sends to group

Alice: 1

Pizza

To Bob

To Cindy

To David

Bob reads ("Pizza", <alice:1>), modifies the value & sends to group

Alice: 1, Bob: 1

Chinese

To Alice

To Cindy

To David

Bob's version updates Alice's choice

<u>Receivers</u>

<alice: 1, bob:1> is causal to & follows **<alice: 1>**

Alice reads ("Chinese", <alice:1, bob:1>), modifies the value & sends to group

Alice: 2, Bob: 1

Moroccan

To Bob

To Cindy

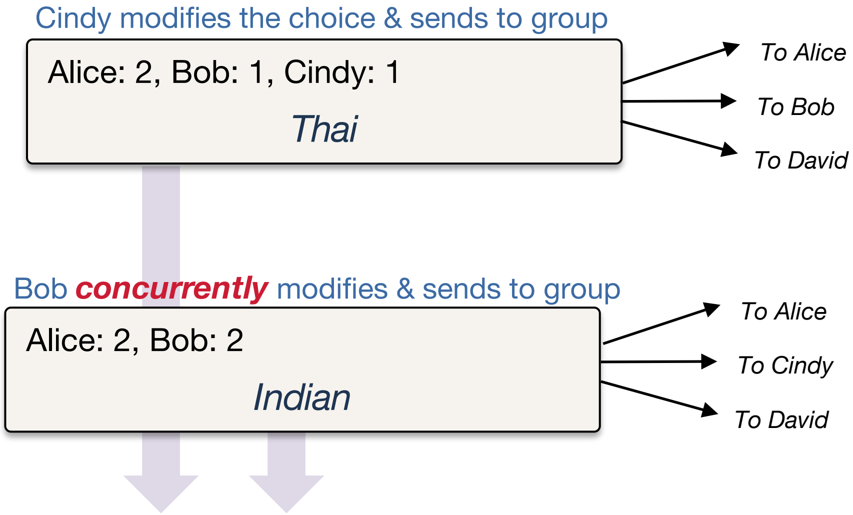
To David

Alice makes changes over Bob's choice

<u>Receivers</u>

<alice: 2, bob:1> is causal to & follows **<alice: 1, bob:1>**

Example



<u>Receivers</u>
<alice: 2, bob:1, cindy:1> is causal to & follows **<alice: 1, bob:1>** and **<alice: 2, bob:1>**

<u>Receivers</u>
<alice: 2, bob:2> is causal to & follows **<alice: 1, bob:1>** and **<alice: 2, bob:1>**

Cindy & Bob's changes are concurrent – members must resolve conflict

<u>Receiver</u>
<alice: 2, bob:1, cindy:1> is *<u>concurrent</u>* with **<alice: 2, bob:2>**

Vector clocks: Rules

1. Vector initialized to 0 at each process *i* for *N* processes
   $V_i\,[\,j\,] = 0$ **for** $i, j = 1, \ldots, N$

2. Process increments its element of the vector in local vector before timestamping event:
   $V_i\,[\,i\,] = V_i\,[\,i\,] + 1$

3. Message is sent from process P_i with V_i attached to it

4. When P_j receives message, compares vectors element by element and sets local vector to higher of two values
   $V_j\,[\,i\,] = max(V_i\,[\,i\,], V_j\,[\,i\,])$ **for** $i = 1, \ldots, N$

For example,
received: ***[0, 5, 12, 1]***, have: ***[2, 8, 10, 1]***
new timestamp: ***[2, 8, 12, 1]***

Comparing vector timestamps

Define

$V = V'$ iff $V[i] = V'[i]$ for $i = 1 \ldots N$
$V < V'$ iff $V \leq V'$ and $V[i] \leq V'[i]$ for $i = 1 \ldots N$

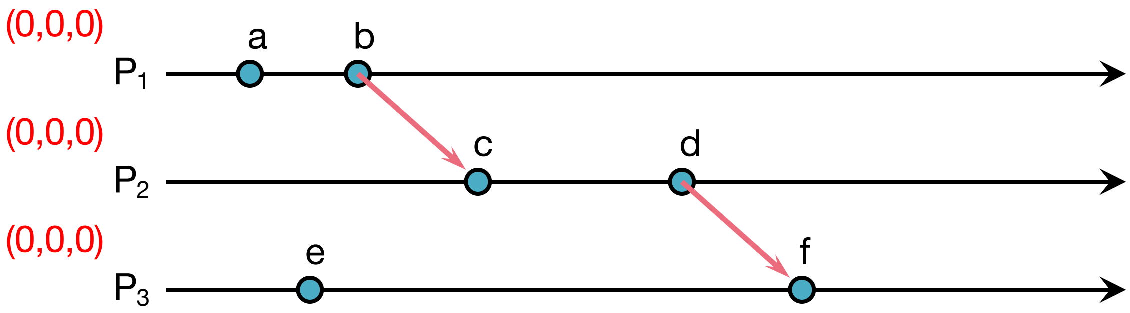
For any two events e, e'

if $e \rightarrow e'$ then $V(e) < V(e')$ … just like Lamport timestamps

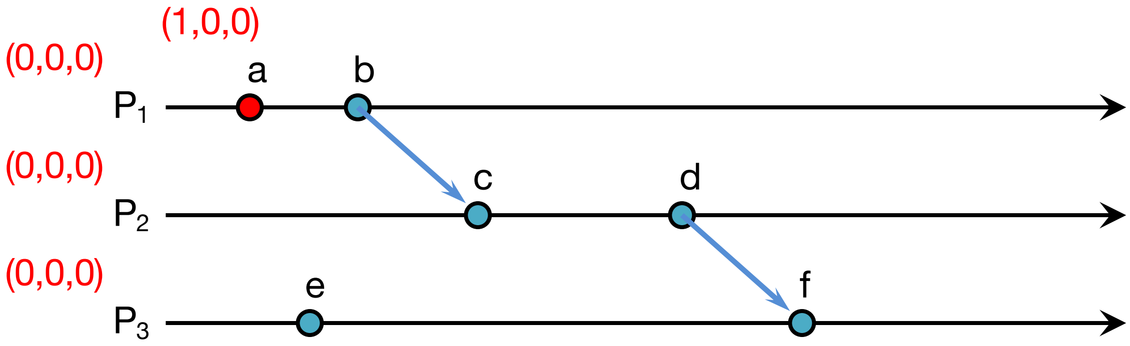
if $V(e) < V(e')$ then $e \rightarrow e'$

Two events are **concurrent** if **neither** $V(e) < V(e')$ **nor** $V(e') < V(e)$

Vector timestamps

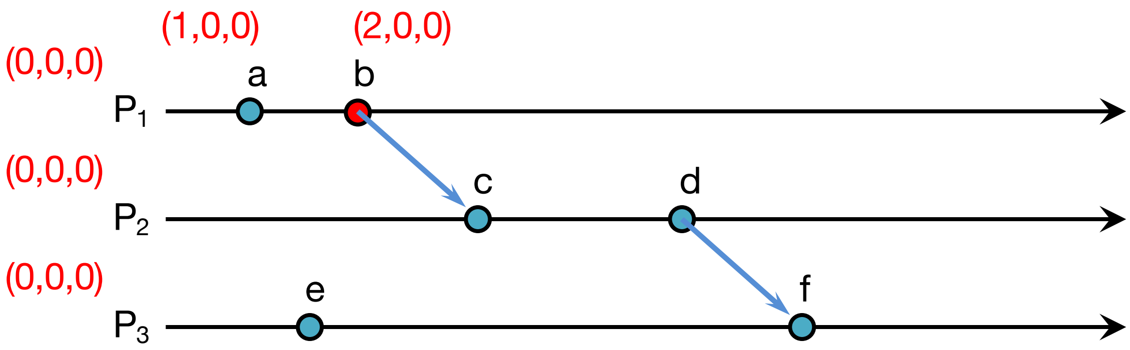


Vector timestamps



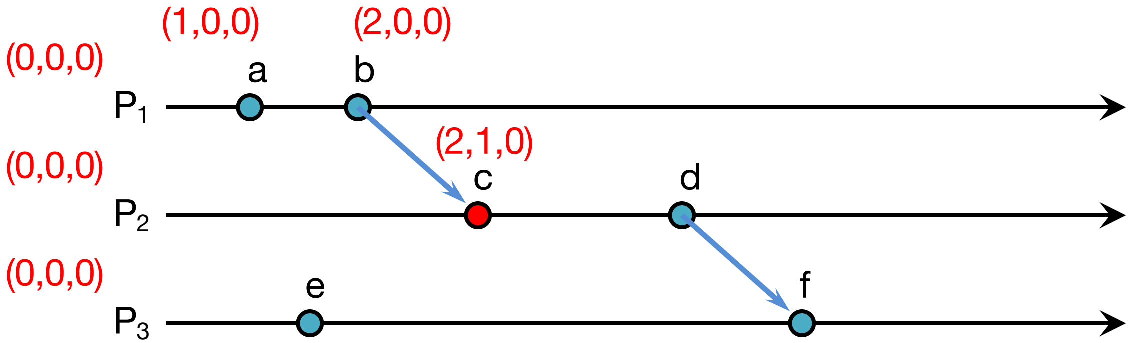
Event	timestamp
a	(1,0,0)

Vector timestamps



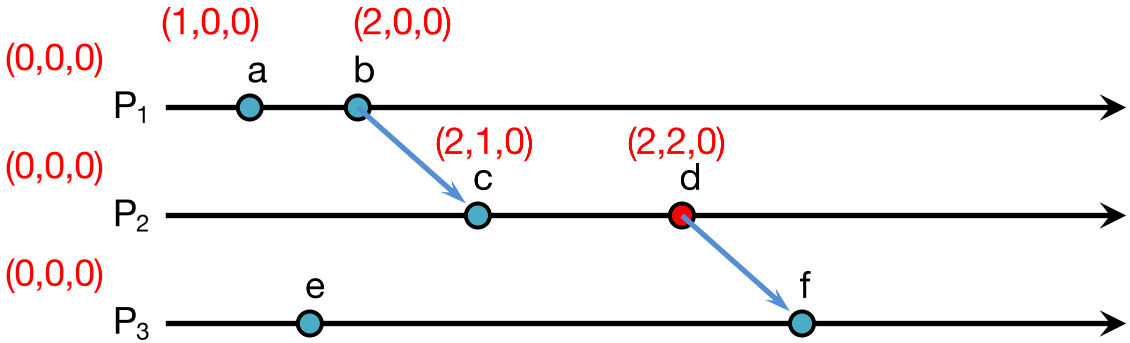
Event	timestamp
a	(1,0,0)
b	(2,0,0)

Vector timestamps



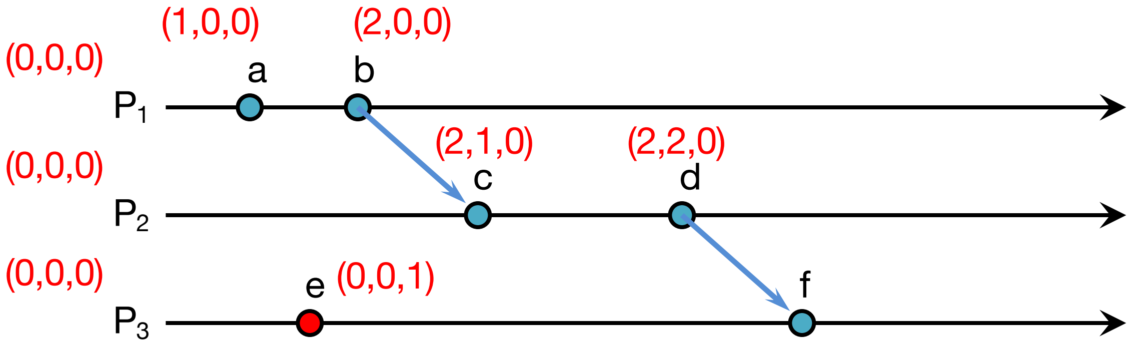
Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)

Vector timestamps



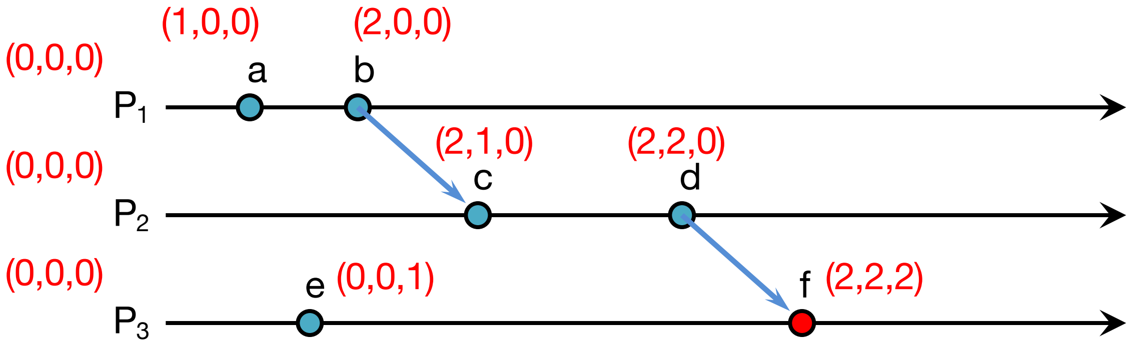
Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)

Vector timestamps



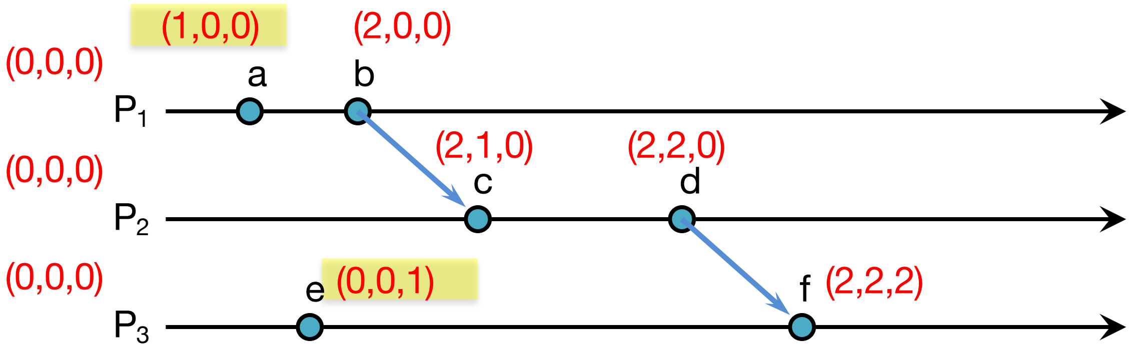
Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)

Vector timestamps



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)

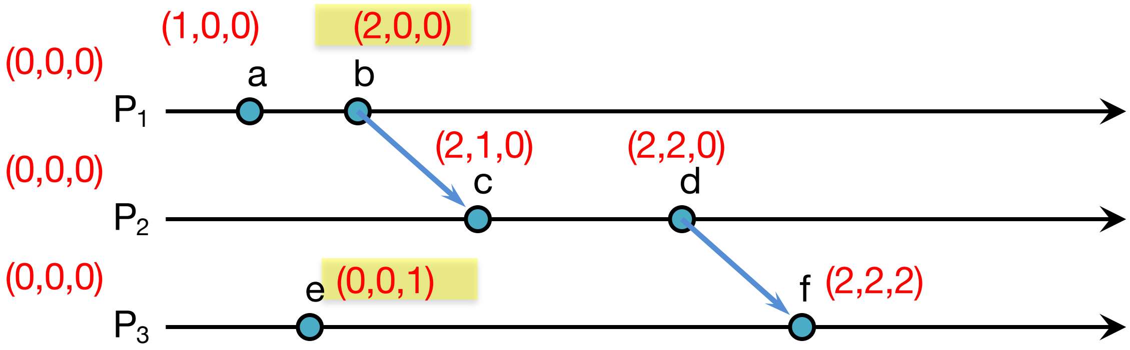
Vector timestamps



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

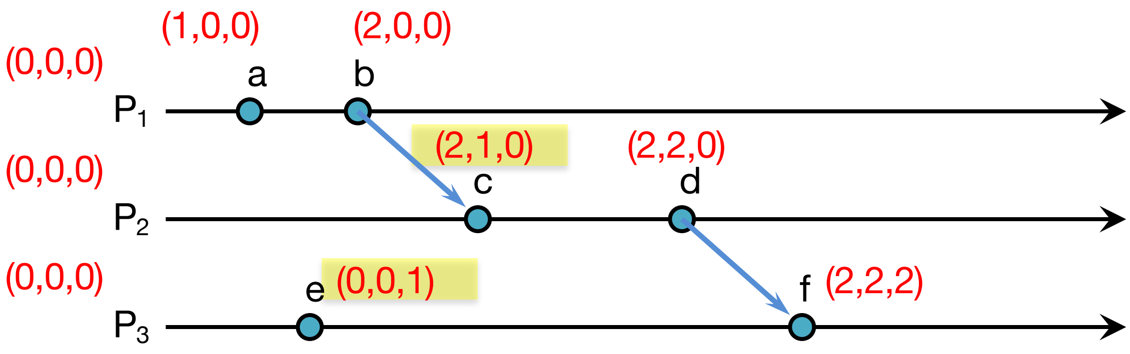
Vector timestamps



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

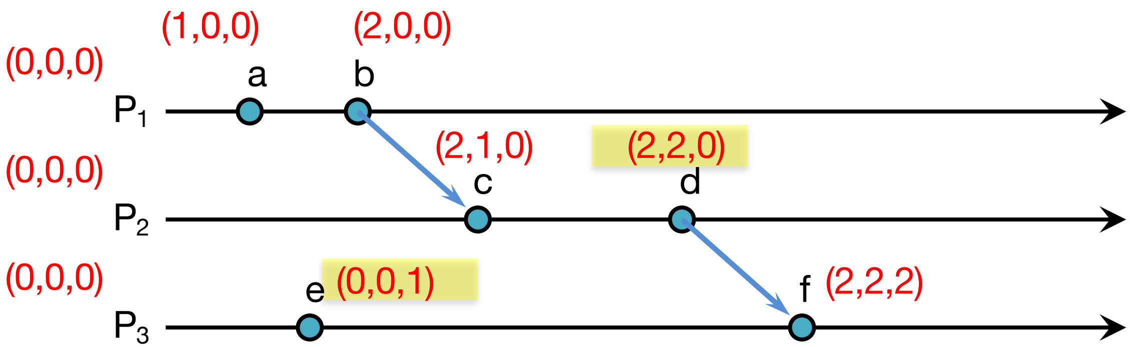
Vector timestamps



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 (c and e)

Vector timestamps



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 (d and e)

Generalizing Vector Timestamps

- **A "vector" can be a list of tuples instead of a vector of numbers:**
  - For processes $P_1, P_2, P_3, \ldots$
  - Each process has a globally unique Process ID, P_i (e.g., *MAC_address:PID*)
  - Each process maintains its own timestamp: $T_{P1}, T_{P2}, \ldots$
  - Vector: { $<P_1, T_{P1}>$, $<P_2, T_{P2}>$, $<P_3, T_{P3}>$, … }

- **One process may only have only partial knowledge of others**
  - New timestamp for a received message:
    - Compare all matching sets of process IDs: set to highest of values
    - Any non-matched $<P, T>$ sets get added to the timestamp
  - For a *happened-before* relation:
    - At least one set of process IDs must be common to both timestamps
    - Match all corresponding $<P, T>$ sets: A:$<P_i, T_a>$, B:$<P_i, T_b>$
    - If $T_a \leq T_b$ for all common processes P, then $A \rightarrow B$

Vector Clocks Summary

- Vector clocks give us a way of identifying which events are causally related
- We are guaranteed to get the sequencing correct

But

- The size of the vector increases with more actors
  … and the entire vector must be stored with the data
- Comparison takes more time than comparing two numbers
- What if messages are concurrent?
  - App will have to decide how to handle conflicts

Summary: Logical Clocks & Partial Ordering

- Causality
  - If $a \rightarrow b$ then event *a* can affect event *b*

- Concurrency
  - If neither $a \rightarrow b$ nor $b \rightarrow a$ then one event cannot affect the other

- Partial Ordering
  - Causal events are sequenced

- Total Ordering
  - All events are sequenced

The End