Time and Logical Clocks 2

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

CS 240: Computing Systems and Concurrency
Lecture 4

Marco Canini

Lamport Clocks Review

- Happens-Before relationship
  - Event **a** ***happens before*** event **b (a → b)**
  - **c, d** not related by → so ***concurrent,*** written as **c** || **d**
- Lamport clocks is a logical clock construction to capture the order of events in a distributed systems (disregarding the precise clock time)
  - Tag every event **a** by C(**a**)
  - If **a** → **b**, then ?
  - If *C*(**a**) < *C*(**b**), then ?
  - If **a || b**, then ?

Lamport Clocks Review

- Happens-Before relationship
  - Event **a** ***happens before*** event **b (a → b)**
  - **c, d** not related by → so ***concurrent,*** written as **c** || **d**
- Lamport clocks is a logical clock construction to capture the order of events in a distributed systems (disregarding the precise clock time)
  - Tag every event **a** by C(**a**)
  - If **a** → **b**, then *C*(**a**) < *C*(**b**)
  - If *C*(**a**) < *C*(**b**), then **NOT b → a** (**a** → **b** or **a || b**)
  - If **a || b**, then nothing

Lamport Clocks and causality

- Lamport clock timestamps **don't capture causality**

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

$$\mathbf{a} \rightarrow b \rightarrow \ldots \rightarrow y \rightarrow \mathbf{z}$$

Take-away points: Lamport clocks

- Can **totally-order** events in a distributed system: that's useful!
  - We saw an application of Lamport clocks for totally-ordered multicast

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

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

Today

1. Logical Time: Vector clocks

Vector clock: Introduction

- One integer **can't** order events in **more than one** process

- So, a ***Vector Clock* (VC)** is a **vector** of integers, **one entry for each** process in the **entire distributed system**
  - Label event **e** with $VC(\mathbf{e}) = [c_1, c_2 \ldots, c_n]$
    - Each entry $\boldsymbol{c_k}$ is a **count of events** in process $\boldsymbol{k}$ that **causally precede** **e**

Vector clock: Update rules

- Initially, all vectors are [0, 0, …, 0]

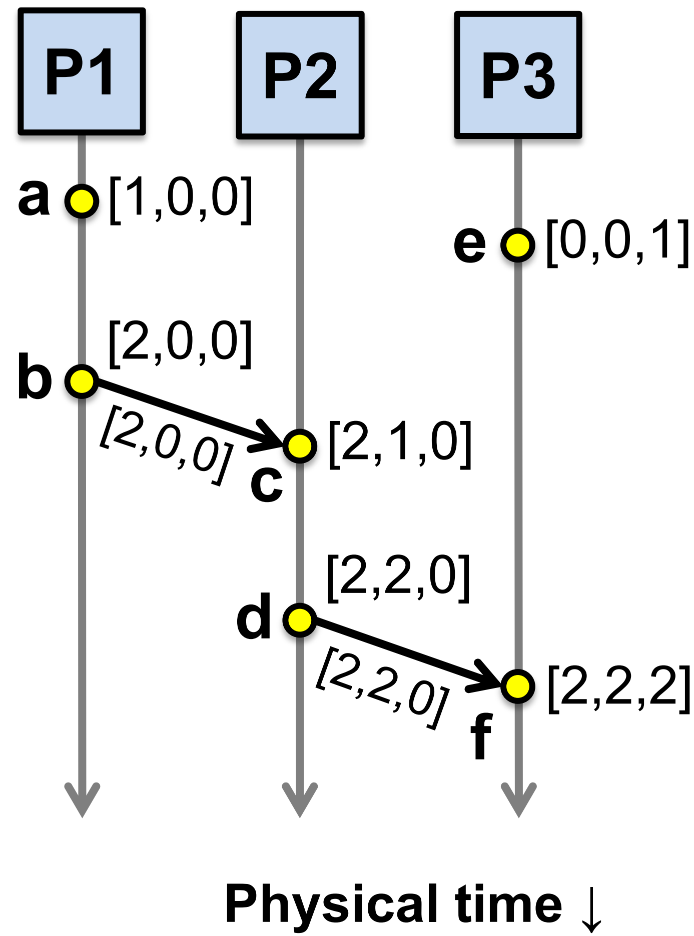
- Two **update rules:**

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 $c_k = \max\{c_k, d_k\}$, for $k = 1\ldots n$
   – Increment local entry c_j

Vector clock: Example

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



Comparing vector timestamps

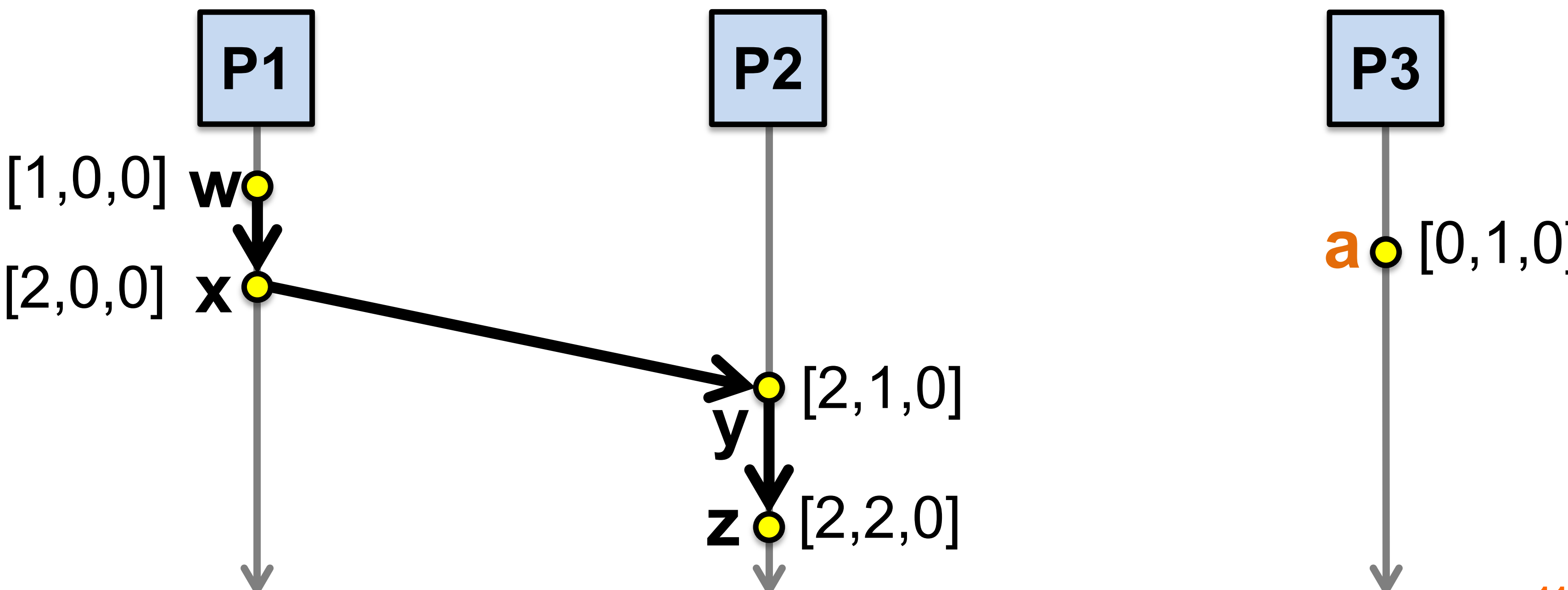
- Rule for comparing vector timestamps:
  - $V(\mathbf{a}) = V(\mathbf{b})$ when $\mathbf{a}_k = \mathbf{b}_k$ for all k
  - $V(\mathbf{a}) < V(\mathbf{b})$ when $\mathbf{a}_k \leq \mathbf{b}_k$ for all k and $V(\mathbf{a}) \neq V(\mathbf{b})$

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

Vector clocks capture causality

- V(**w**) < V(**z**) **then** there is a chain of events linked by Happens-Before (→) between **w** and **z**

- If 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: NOT z → a (either a → z or a || z)

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

Conclusion: a → z

Vector clock timestamps precisely capture Happens-Before relationship (potential causality)

Disadvantage of vector timestamps

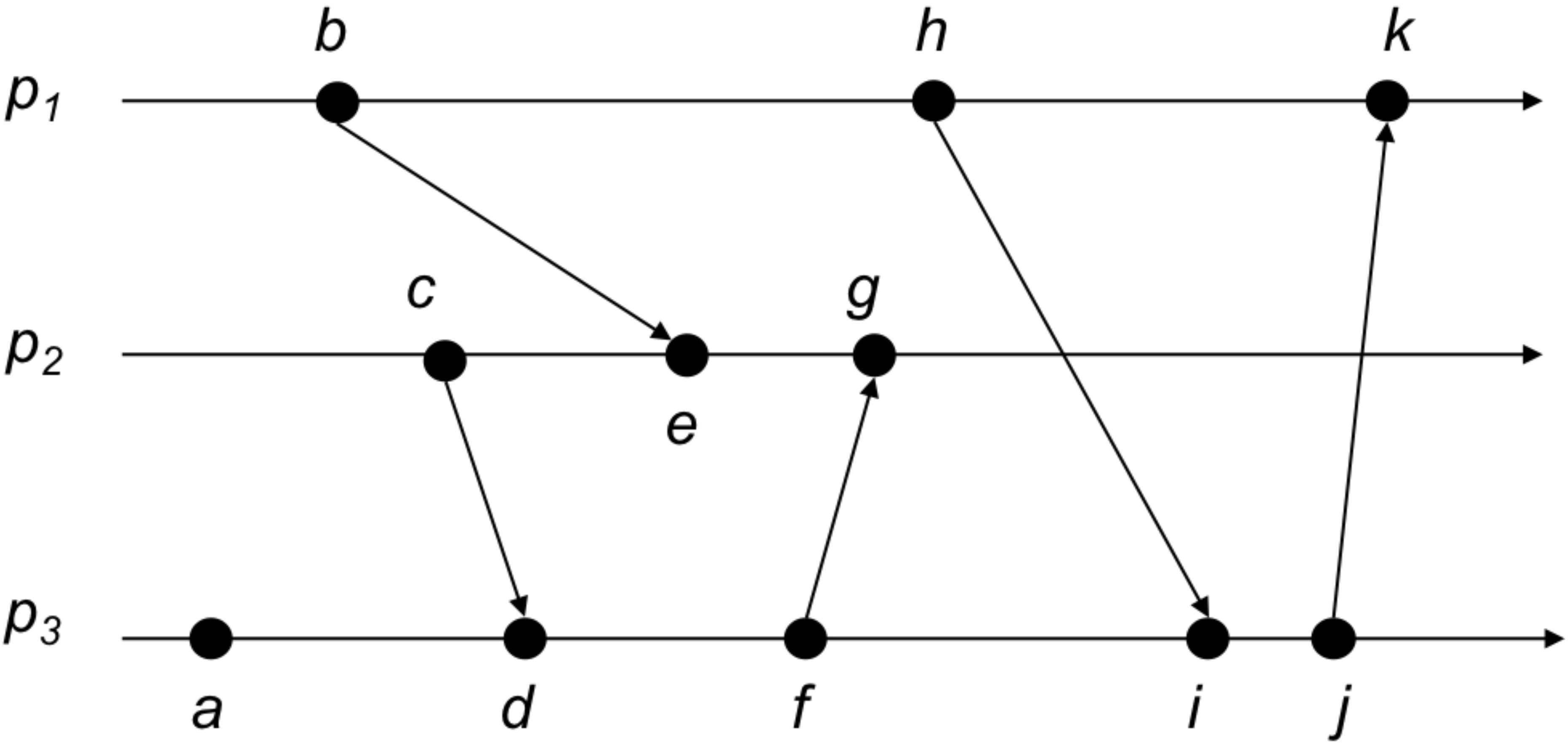
- Compared to Lamport timestamps,
  vector timestamps O(n) overhead for storage and communication, n = no. of processes

Take-away points

- Vector Clocks
  - Precisely capture happens-before relationship

VC Quiz

- Suppose these processes maintain vector clocks. Write the vector clock of each event starting from clock time 0.



Safety and liveness properties

Reasoning about fault tolerance

- This is hard!
  - How do we design fault-tolerant systems?
  - How do we know if we're successful?
- Often use "properties" that hold true for every possible execution
- We focus on **safety** and **liveness** properties

Properties

- **Property**: a predicate that is evaluated over a run of the system
  - "every message that is received was previously sent"
- Not everything you may want to say about a system is a property:
  - "the program sends an average of 50 messages in a run"

Safety properties

- “Bad things” don’t happen, ever
  - No more than k processes are simultaneously in the critical section
  - Messages that are delivered are delivered in causal order
- A safety property is “prefix closed”:
  - if it holds in a run, it holds in every prefix

Liveness properties

- “Good things” eventually happen
  - A process that wishes to enter the critical section eventually does so
  - Some message is eventually delivered
  - Eventual consistency: if a value doesn’t change, two servers will eventually agree on its value
- Every run can be extended to satisfy a liveness property
  - If it does not hold in a prefix of a run, it does not mean it may not hold eventually

Often a trade-off

- “Good” and “bad” are application-specific
- Safety is very important in banking transactions
  – May take some time to confirm a transaction
- Liveness is very important in social networking sites
  – See updates right away