Vector Clocks and Distributed Snapshots

COS 418: *Distributed Systems*
Lecture 5

Kyle Jamieson

Today

1. **Logical Time: Vector clocks**
2. Distributed Global Snapshots

2

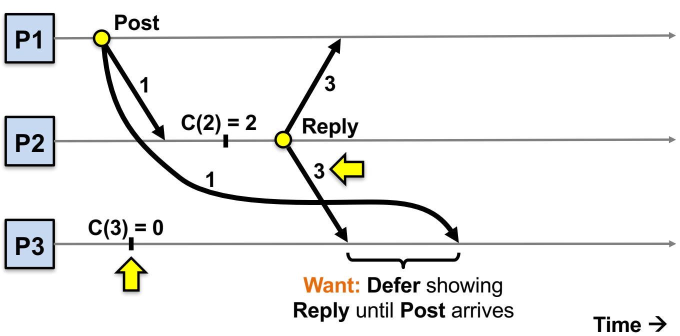
Motivation: Distributed discussion board

3

Distributed discussion board

- Users join specific **discussion groups**
  - Each user runs a **process** on a different machine
  - Messages (**posts** or **replies**) sent to all users in group
- **Goal:** Ensure **replies follow posts**
- **Non-goal:** Sort **posts and replies chronologically**
- *Can Lamport Clocks help here?* 

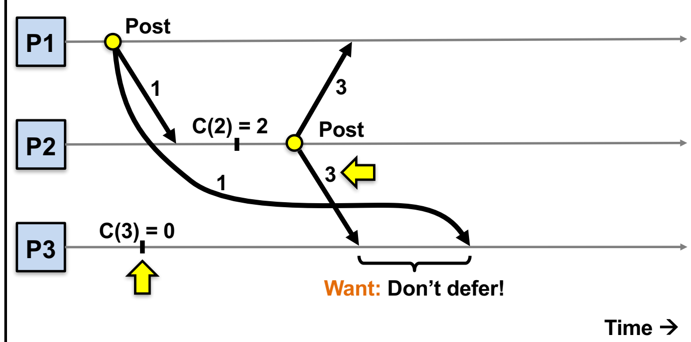
Lamport Clock-based discussion board



- **Defer showing message if C(message) > local clock + 1?**

5

Lamport Clock-based discussion board



- **No! Gap could be due to other independent posts**

6

Lamport Clocks and causality

- **Problem generalizes: Replies to replies to posts** intermingle with **replies to posts**
- 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:

  a → b → ... → y → **z**

- **Chain of events** captures **replies to posts** in our example

7

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**

8

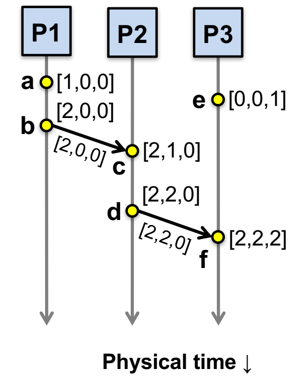
Vector clock: Update rules

- 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 $c_k = \max\{c_k, d_k\}$
   - 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

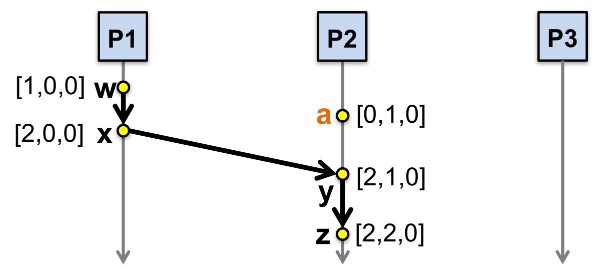


Comparing vector timestamps

- Rule for comparing vector timestamps:
  - V(**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

Vector clocks establish causality

- V(**w**) < V(**z**) **then** there is a chain of events linked by Happens-Before (→) between **a** and **z**
- If V(**a**) || V(**w**) then there is **no such chain of events** between **a** and **w**



Two events **a, z**

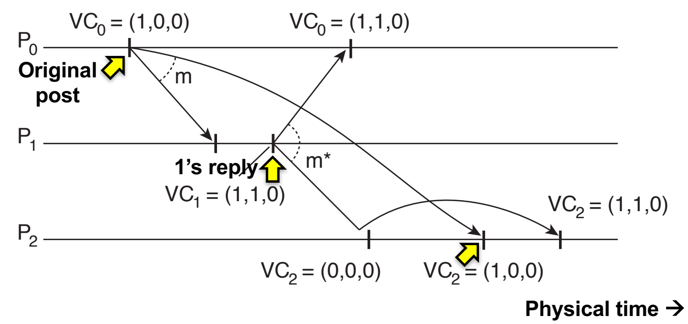
Lamport clocks: C(a) < C(z)
Conclusion: None

Vector clocks: V(a) < V(z)
Conclusion: a → … → z

Vector clock timestamps tell us about causal event relationships

13

VC application: Causally-ordered bulletin board system



- User 0 posts, user 1 replies to 0's post; user 2 observes

14

Today

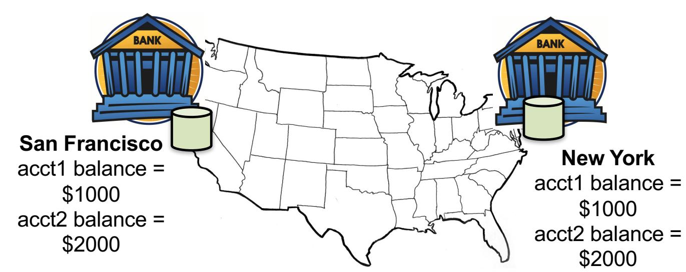
1. Logical Time: Vector clocks

2. **Distributed Global Snapshots**
   - **Chandy-Lamport algorithm**
   - Reasoning about C-L: Consistent Cuts

15

Distributed Snapshots

- What is the state of a distributed system?



16

Example of a global snapshot

But that was easy

- In our system of world leaders, we were able to capture their 'state' (*i.e.*, likeness) easily
  - Synchronized in space
  - Synchronized in time
- How would we take a global snapshot if the leaders were all at home?
- What if Obama told Trudeau that he should really put on a shirt?
- This message is part of our system state!

System model

- *N* **processes** in the system with no process failures
  - Each process has some **state** it keeps track of
- There are two first-in, first-out, unidirectional **channels** between every process pair P and Q
  - Call them **channel(P, Q)** and channel**(Q, P)**
  - The channel has **state,** too: the set of messages inside
  - For today, assume all messages sent on channels arrive intact and unduplicated

Global snapshot is global state

- Each distributed application has a number of processes (leaders) running on a number of physical servers
- These processes communicate with each other via channels
- A ***global snapshot*** captures
  1. The **local states of each process** (*e.g.*, program variables), along with
  2. The state of **each communication channel**

Why do we need snapshots?

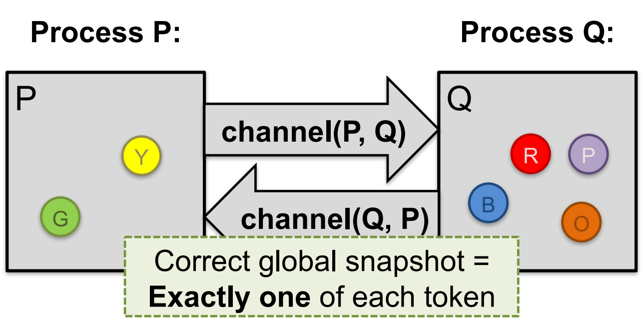
- **Checkpointing:** Restart if the application fails
- **Collecting garbage:** Remove objects that don't have any references
- **Detecting deadlocks:** The snapshot can examine the current application state
  - **Process A** grabs **Lock 1**, **B** grabs **2**, **A** waits for **2**, **B** waits for **1... ... ...**
- **Other debugging:** A little easier to work with than printf…

Just synchronize local clocks?

- Each process **records state** at **some agreed-upon time**
- But **system clocks skew,** significantly with respect to CPU process' clock cycle
  - And we **wouldn't record messages** between processes
- Do we need synchronization?
- What did Lamport realize about ordering events?

System model: Graphical example

- Let's represent process state as a set of colored ***tokens***
- Suppose there are two processes, **P** and **Q:**



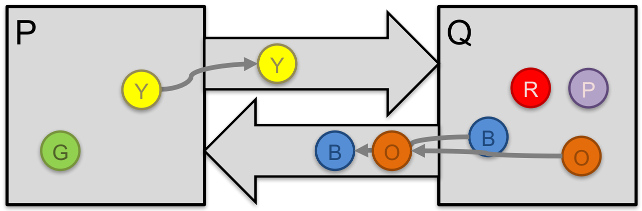
When is inconsistency possible?

- Suppose we take snapshots **only from a process perspective**
- Suppose snapshots happen **independently** at each process
- Let's look at the implications...

Problem: Disappearing tokens

- P, Q put tokens into channels, **then** snapshot

This snapshot **misses** Y, B, and O tokens



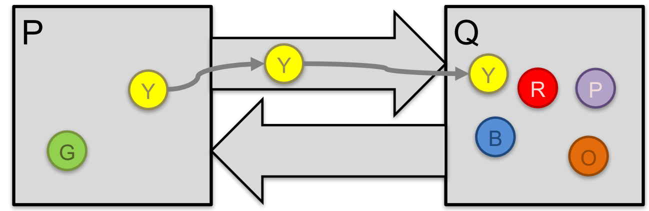
P = { G } **Q = { R, P }**

25

Problem: Duplicated tokens

- P snapshots, **then** sends Y
- Q receives Y, **then** snapshots

This snapshot **duplicates** the Y token



P = { G, Y } **Q = { Y, R, P, B, O }**

26

Idea: "Marker" messages

- What went wrong? We should have captured the state of the **channels** as well
- Let's send a ***marker message*** ▲ to track this state
  - Distinct from other messages
  - Channels deliver marker and other messages FIFO

27

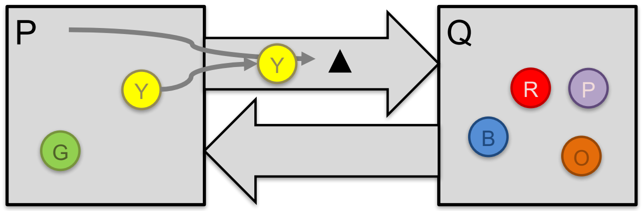
Chandy-Lamport algorithm: Overview

- We'll designate one node (say **P**) to **start** the snapshot
  - Without any steps in between, **P:**
    1. Records its local state ("snapshots")
    2. Sends a marker on each outbound channel
- Nodes remember **whether they have snapshotted**
- **On receiving a marker,** a **non-snapshotted** node performs steps (1) and (2) above

28

Chandy-Lamport: Sending process

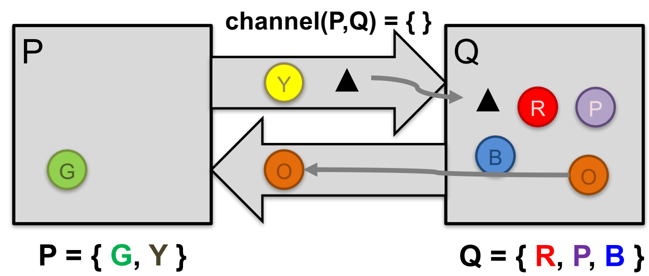
- P **snapshots and sends marker, then** sends Y
- **Send Rule:** Send marker on all outgoing channels
  - **Immediately after snapshot**
  - **Before** sending any **further messages**



snap: P = { G, Y }

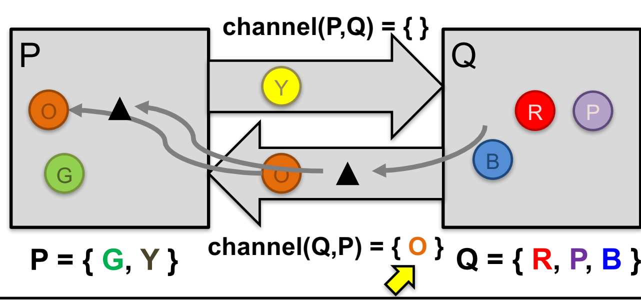
Chandy-Lamport: Receiving process (1/2)

- At the same time, Q sends orange token **O**
- Then, Q receives marker ▲
- **Receive Rule (if not yet snapshotted)**
  - On receiving marker on channel ***c*** record *c*'s state as **empty**



Chandy-Lamport: Receiving process (2/2)

- Q sends marker to P
- P receives orange token **O**, then marker ▲
- **Receive Rule (if already snapshotted):**
  - On receiving marker on ***c*** record *c*'s state: **all msgs from *c* since snapshot**



Terminating a snapshot

- **Distributed algorithm:** No one process decides when it terminates
- Eventually, all processes have received a marker (and recorded their own state)
- All processes have received a marker on all the *N*–1 incoming channels (and recorded their states)
- Later, a central server can **gather the local states** to build a global snapshot

Today

1. Logical Time: Vector clocks

2. Distributed Global Snapshots

- Chandy-Lamport algorithm
- **Reasoning about C-L: Consistent Cuts**

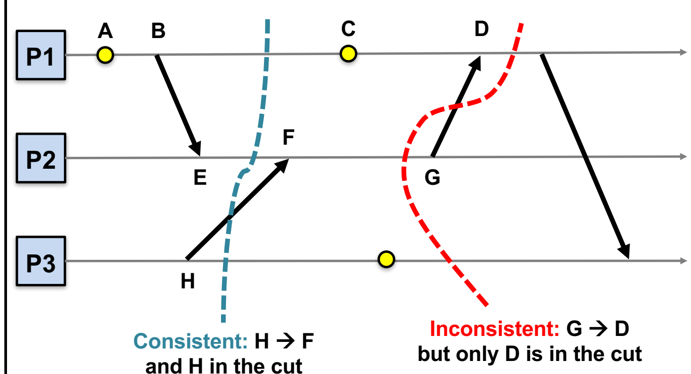
Global states and cuts

- ***Global state*** is a *n*-tuple of local states (one per process **and** channel)
- A **cut** is a subset of the global history that contains an initial prefix of each local state
  - Therefore every cut is a natural global state
  - Intuitively, a cut **partitions** the space time diagram along the time axis
- ***Cut*** = { The last **event** of each **process,** and **message** of each **channel** that is in the cut }

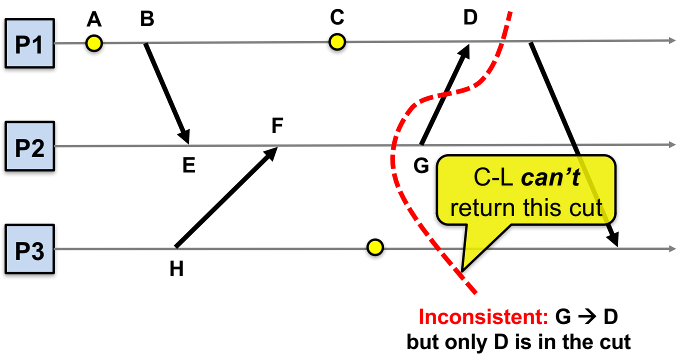
Inconsistent versus consistent cuts

- A **consistent cut** is a cut that **respects causality of events**
- A cut **C** is ***consistent*** when:
  - For each pair of events **e** and **f**, if:
    1. **f** is in the cut, and
    2. **e → f,**
  - then, event **e** is also **in the cut**

Consistent versus inconsistent cuts

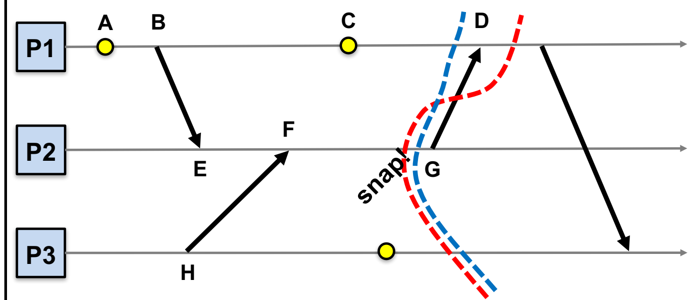


C-L returns a consistent cut



C-L ensures that if **D** is in the cut, then **G** is in the cut

C-L can't return this inconsistent cut



Friday Precept:
RPCs in Go

Monday Topic:
Eventual Consistency & Bayou