Vector Clocks and Distributed Snapshots

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

CS 240: Computing Systems and Concurrency
Lecture 5

Marco Canini

Credits: Kyle Jamieson developed much of the original material.

Today

1. **Logical Time: Vector clocks**

2. Distributed Global Snapshots

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:

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

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 ***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

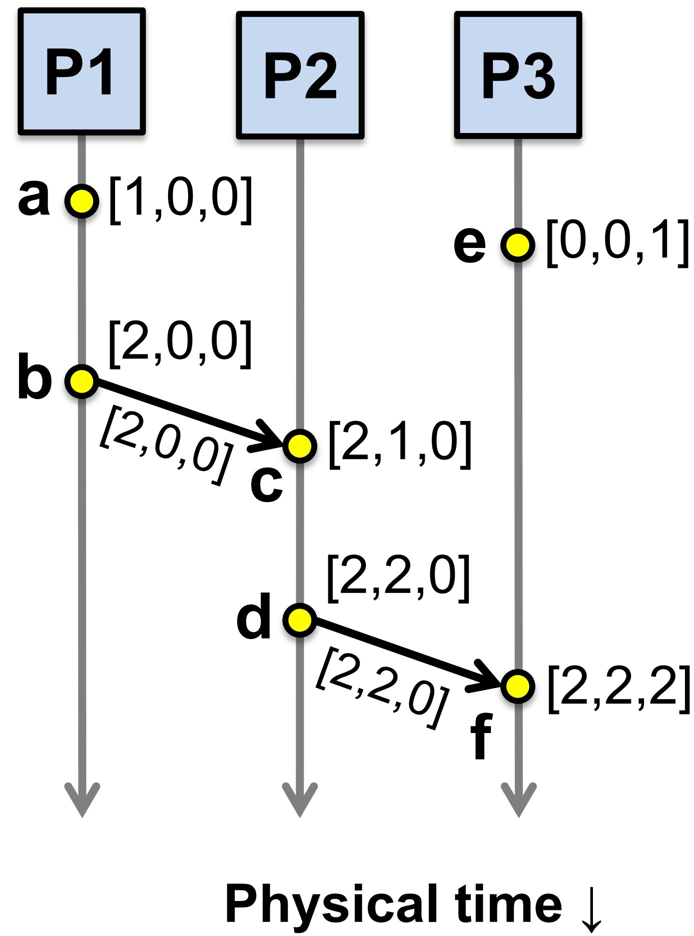
2. If process *j* **receives** message with vector [d_1, d_2, …, d_n]:
   – Set each local entry $c_k = \max\{c_k, d_k\}$, for $k = 1 \ldots n$
   – Increment local entry c_j

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

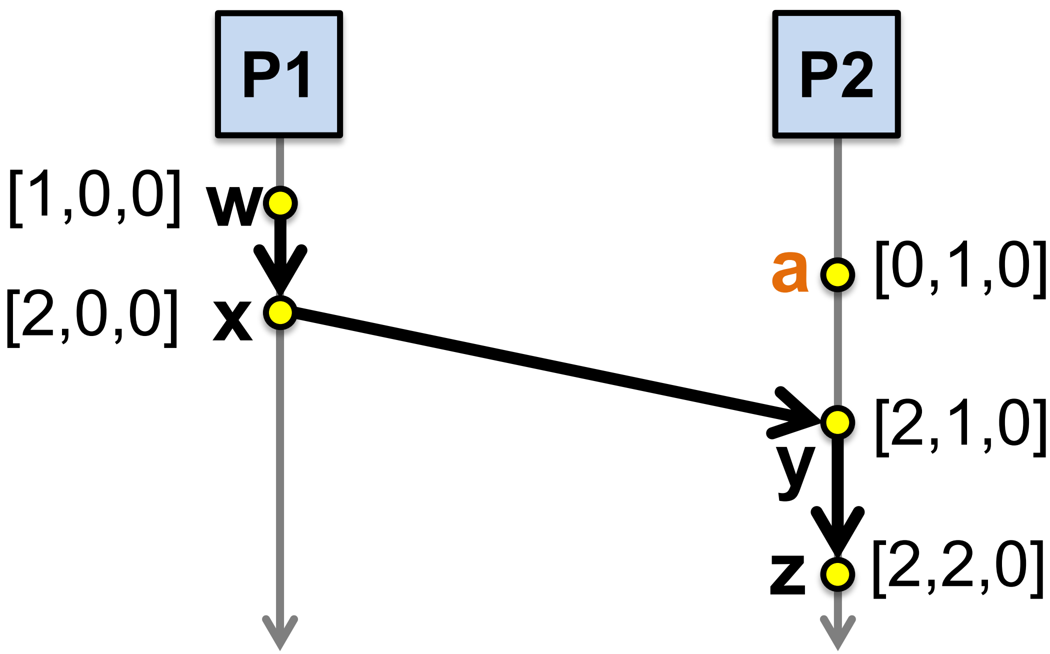
- Rule for comparing vector timestamps:
  - $V(\mathbf{a}) = V(\mathbf{b})$ when $\mathbf{a}_k = \mathbf{b}_k$ for all k
  - $V(\mathbf{a}) < V(\mathbf{b})$ when $\mathbf{a}_k \leq \mathbf{b}_k$ for all k and $V(\mathbf{a}) \neq V(\mathbf{b})$

- 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, b**

Lamport clocks: C(a) < C(b)

Conclusion: NOT b → a (either a → b or a || b)

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

Conclusion: **a → b**

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

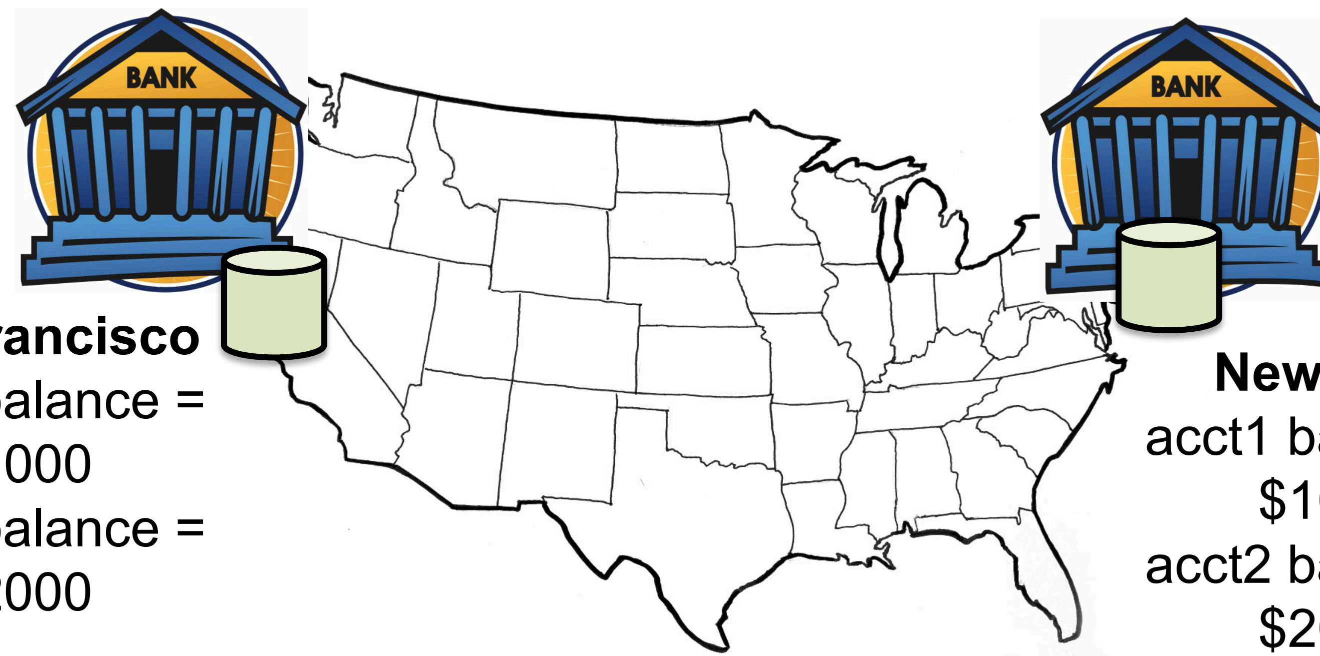
Today

1. Logical Time: Vector clocks

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

Distributed Snapshots

- What is the state of a distributed system?



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
  - All messages sent on channels arrive intact, unduplicated, in order

Aside: FIFO communication channel

"All messages sent on channels arrive intact, unduplicated, in order"

- Q: Arrive?
- Q: Intact?
- Q: Unduplicated?
- Q: In order?

- At-least-once retransmission
- Network layer checksums
- At-most-once deduplication
- Sender include sequence numbers, receiver only delivers in sequence order

- TCP provides all of these when processes don't fail

Global snapshot is global state

- Each distributed system has a number of processes 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), and

  2. The state of **each communication channel**

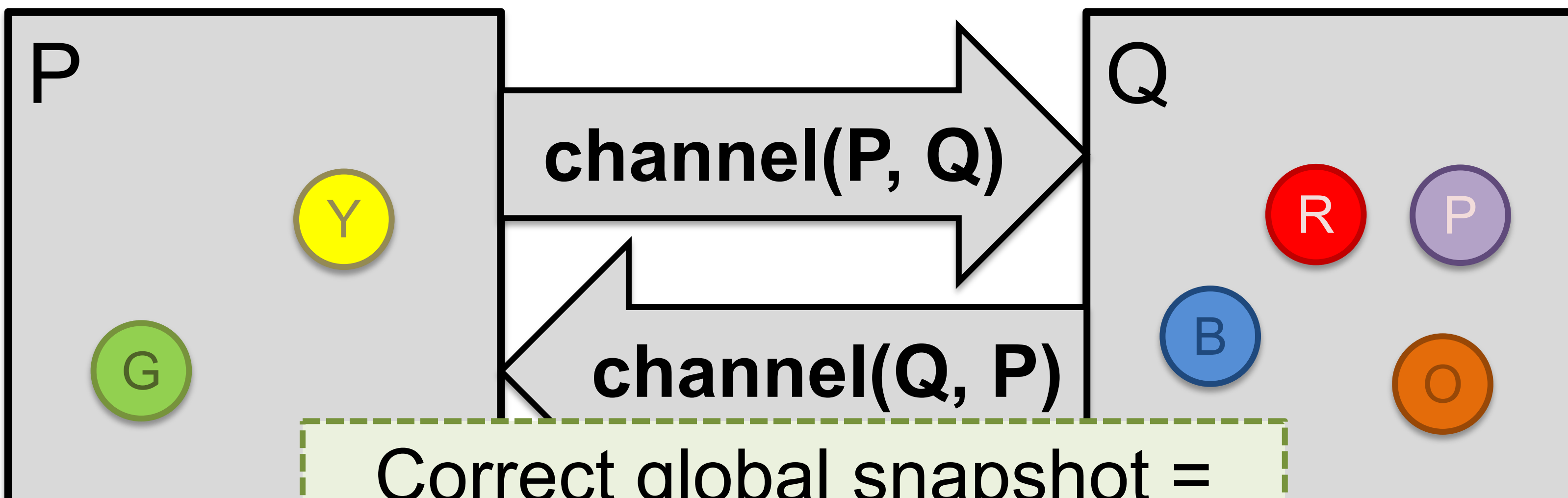
Why do we need snapshots?

- **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…

System model: Graphical example

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

Process P: **Process Q:**



Correct global snapshot =
Exactly one of each token

When is inconsistency possible?

- Suppose we take snapshots **only from a process perspective**

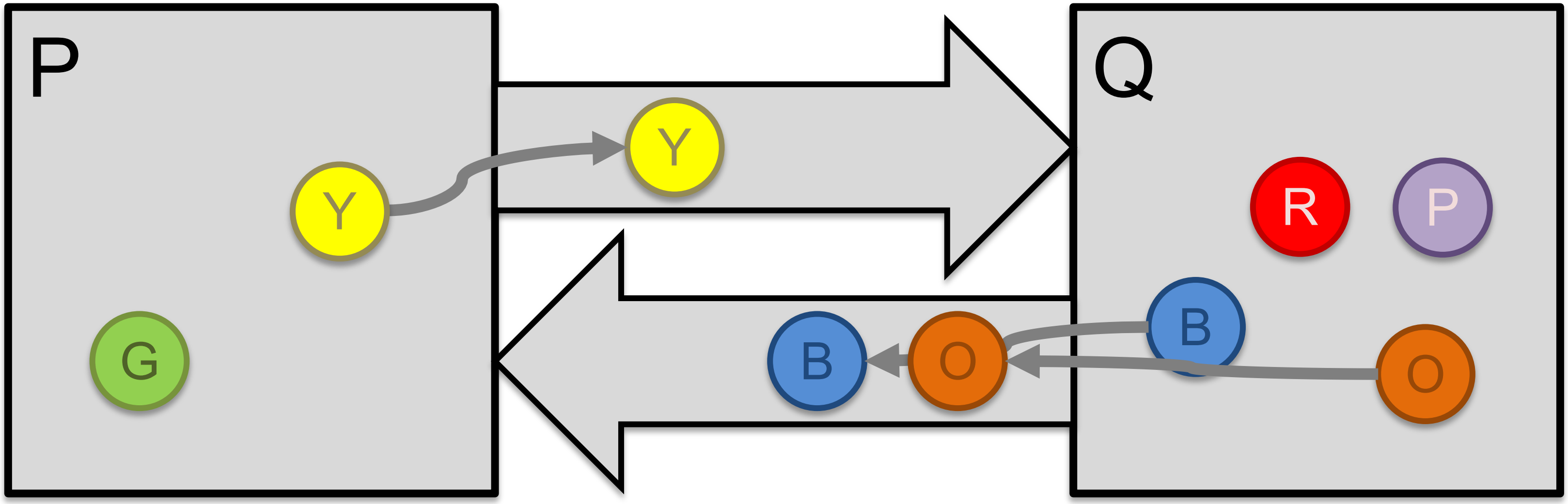
- Suppose snapshots happen **independently** at each process

- 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



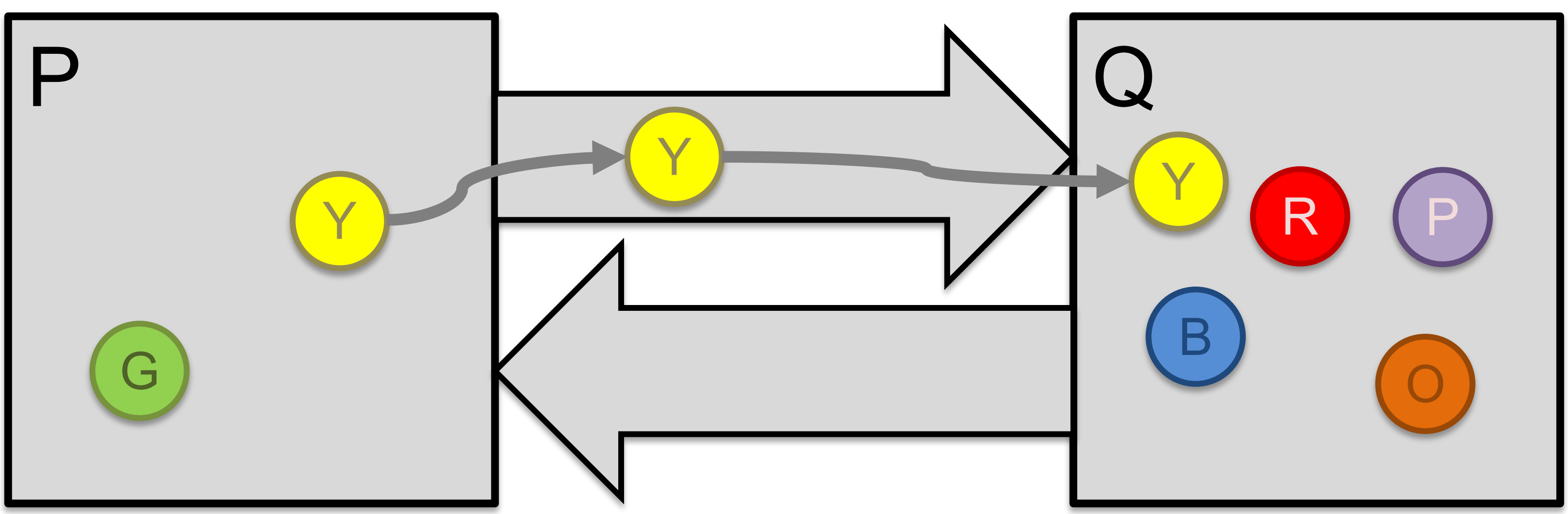
P = { G }

Q = { R, P }

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 }

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

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

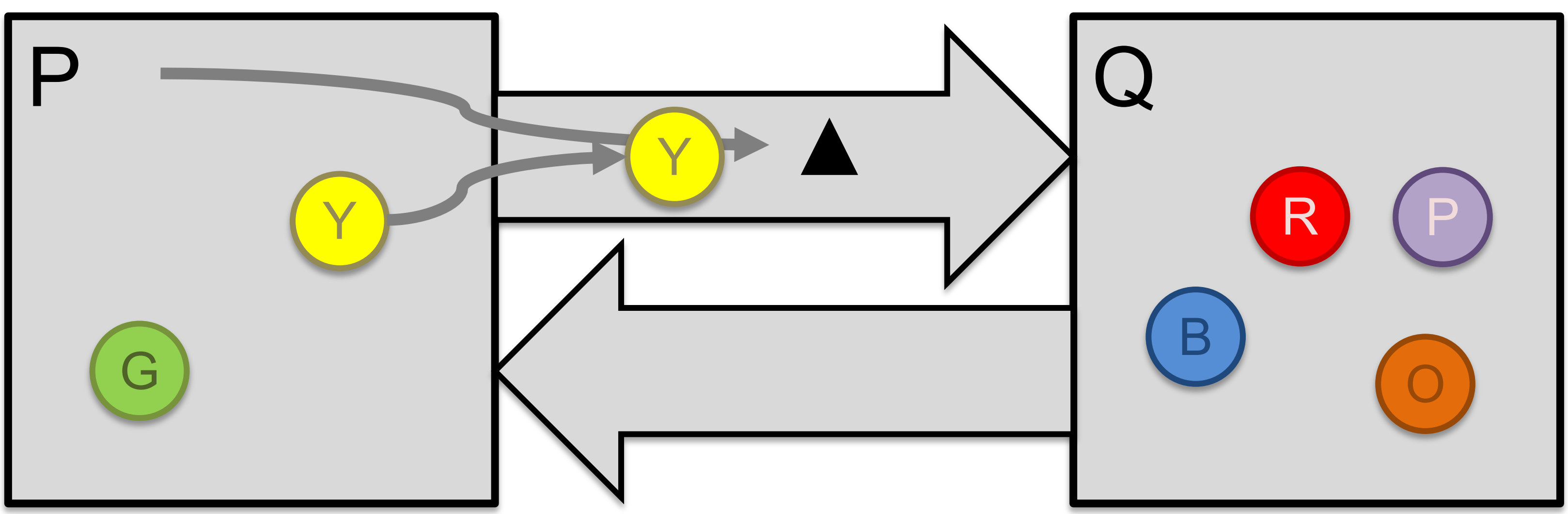
- Nodes remember **whether they have snapshotted**

- **On receiving a marker,** a **non-snapshotted** node performs steps (1) and (2) above

Chandy-Lamport: Sending process

- P **snapshots and sends marker, then** sends Y

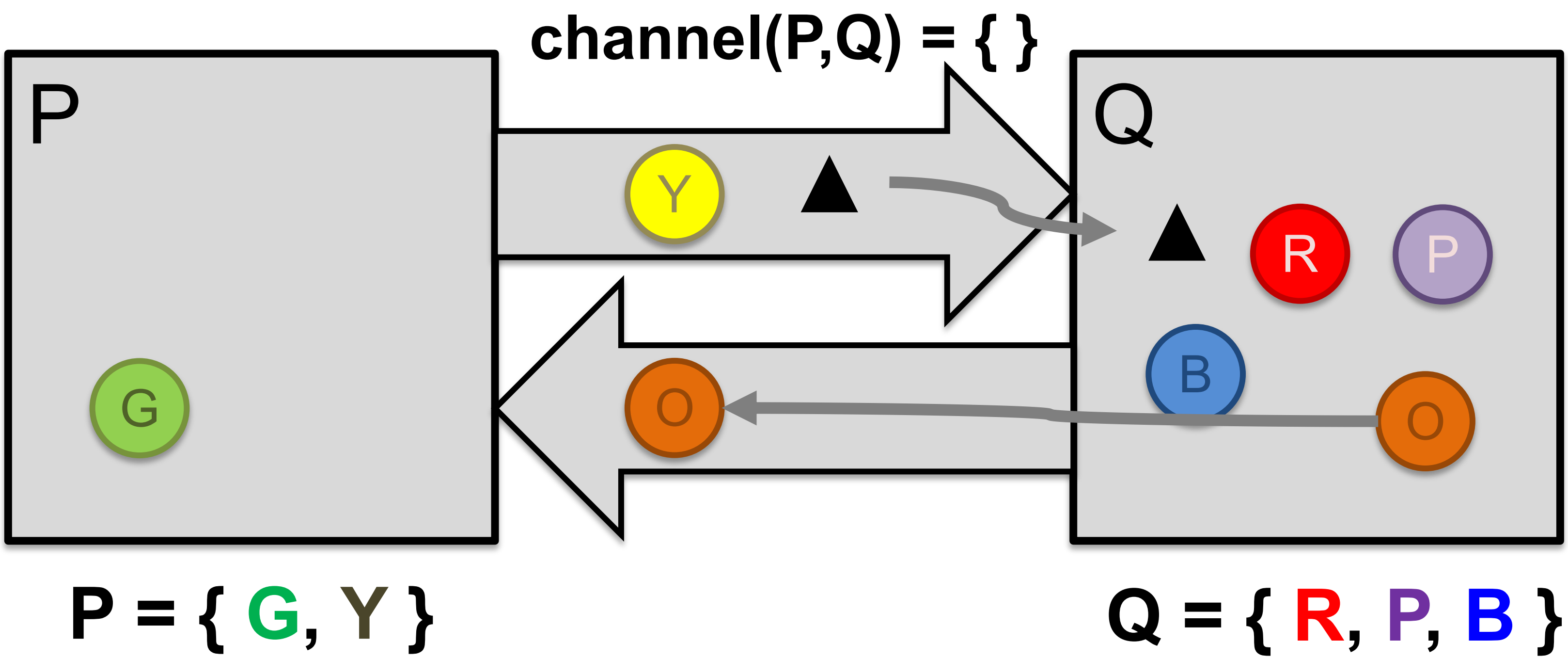
- **Send Rule:** Send marker on all outgoing channels
  - **Immediately after snapshot**
  - **Before** sending any **further messages**



snap: P = { G, Y }

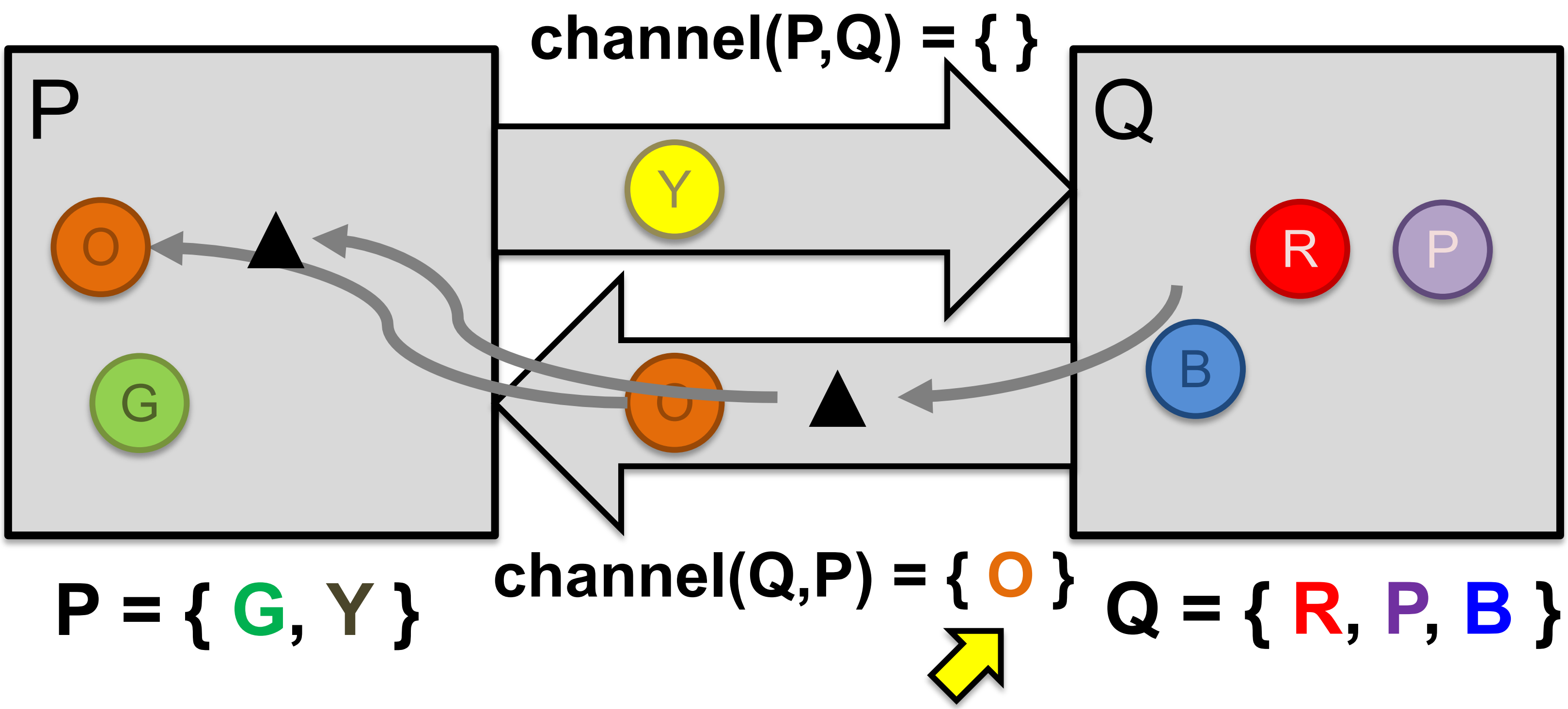
Chandy-Lamport: Receiving process (1/2)

- 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

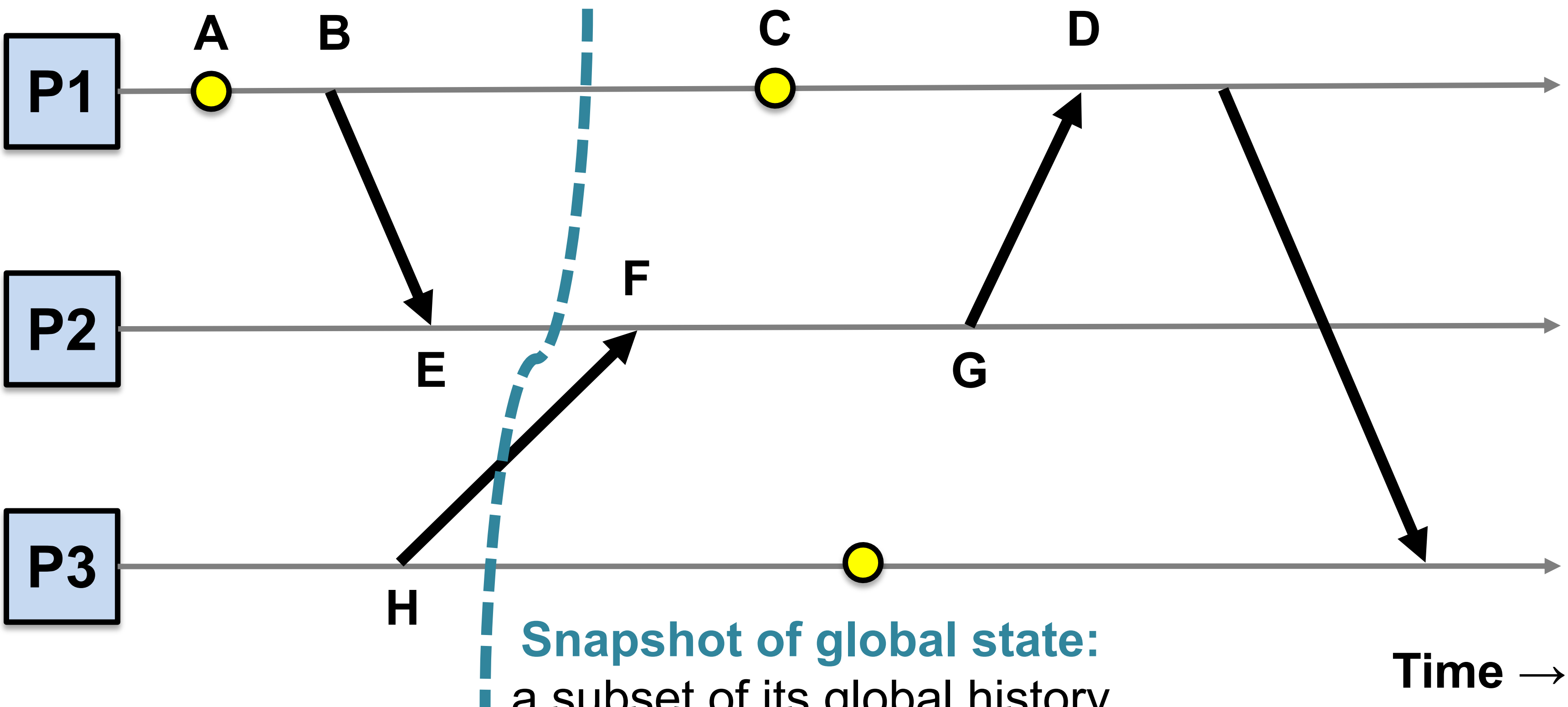
- **Distributed algorithm:** No one process decides when it terminates
- Eventually, all processes have received a marker (and recorded their own state)
- All processes have received a marker on all the N–1 incoming channels (and recorded their states)
- Later, a central server can **gather the local 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 state as cut of system's execution



Cut = { The last **event** of each **process**, and **message** of each **channel** that is in the cut }

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 }

Consistent versus inconsistent cuts

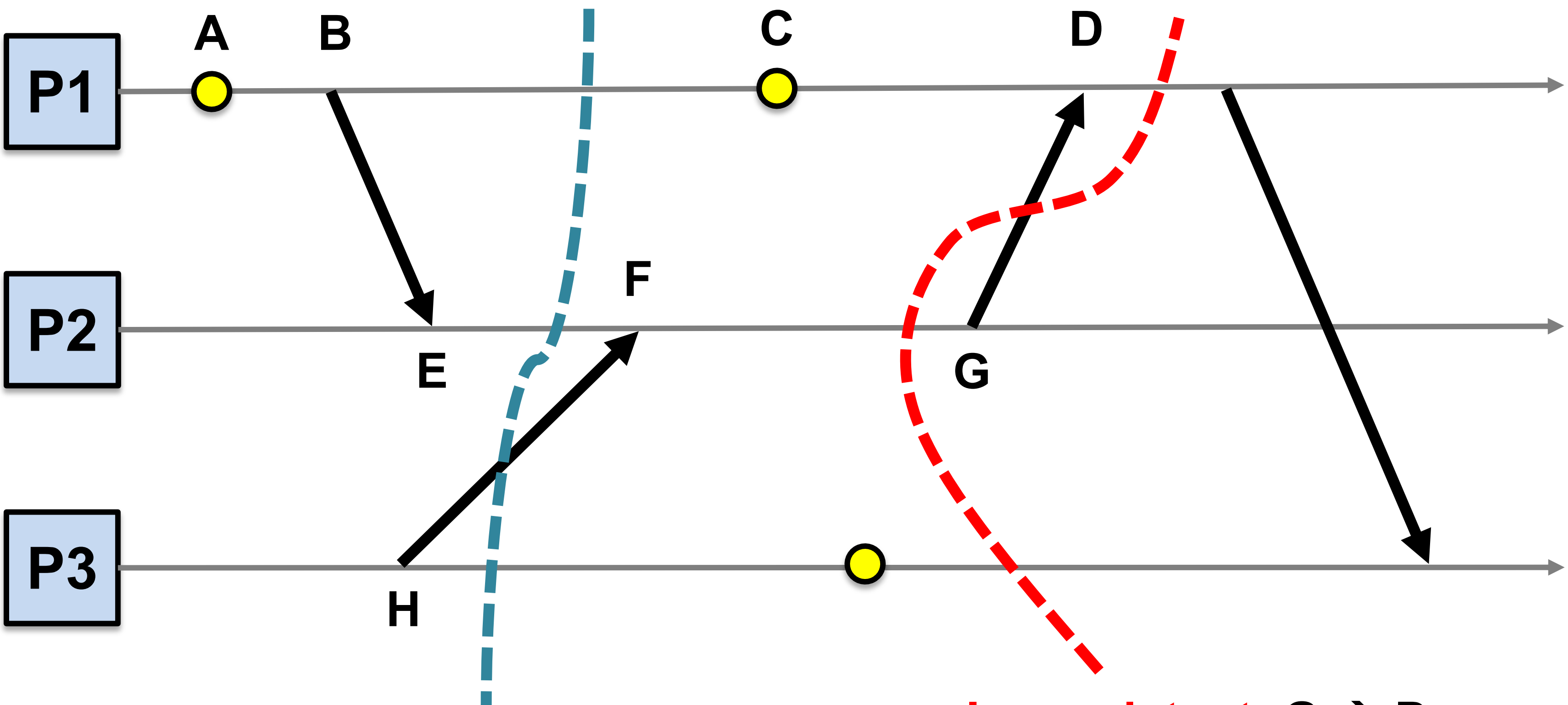
- A **consistent cut** is a cut that **respects causality of events**

- 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**

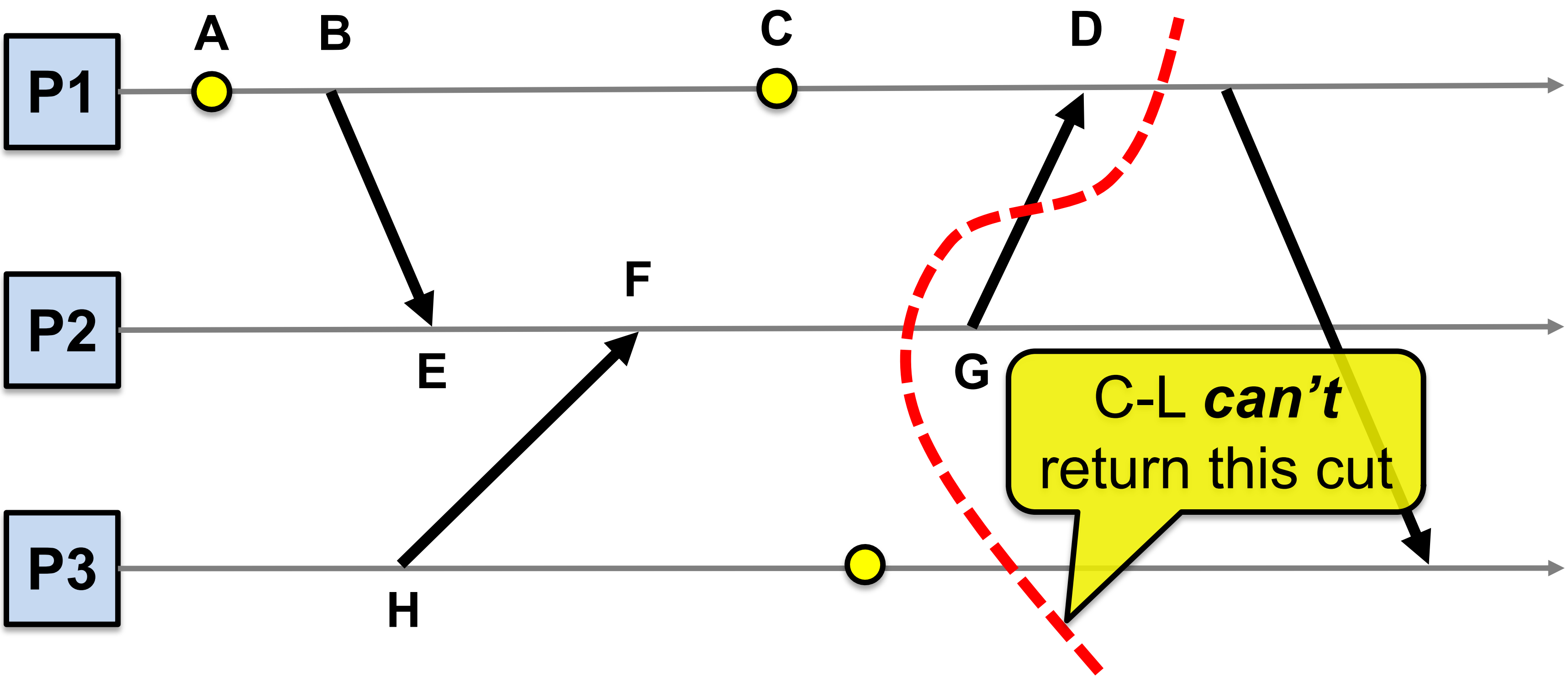
Consistent versus inconsistent cuts



Consistent: H → F
and H in the cut

Inconsistent: G → D
but only D is in the cut

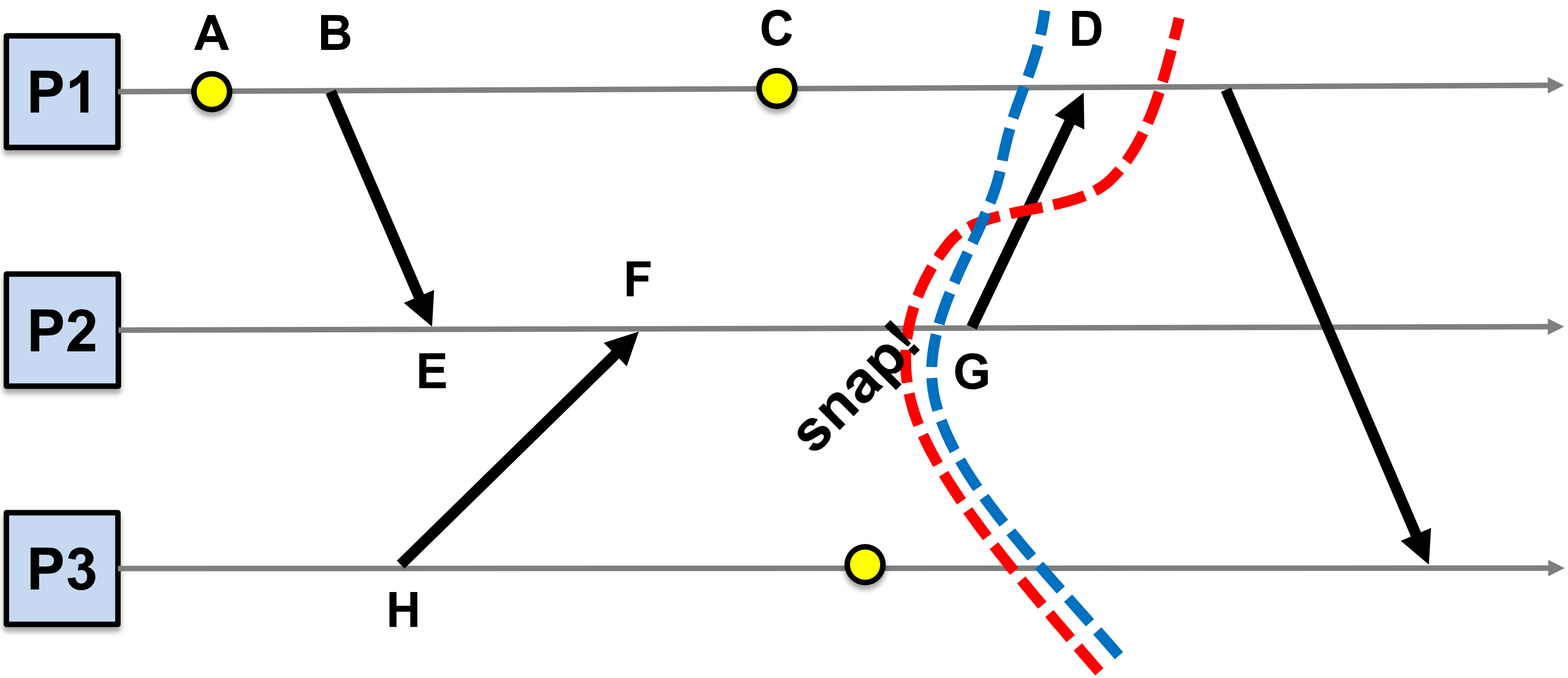
C-L returns a consistent cut



Inconsistent: **G → D**
but only D is in the 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



Take-away points

- Vector Clocks
  - Precisely capture happens-before relationship
- Distributed Global Snapshots
  - FIFO Channels: we can do that!
  - Chandy-Lamport algorithm: use marker messages to coordinate
  - Chandy-Lamport provides a consistent cut

Next Topic:
Eventual Consistency & Bayou