CSE 486/586 Distributed Systems

Global States

Slides by Steve Ko
Computer Sciences and Engineering
University at Buffalo

Last Time

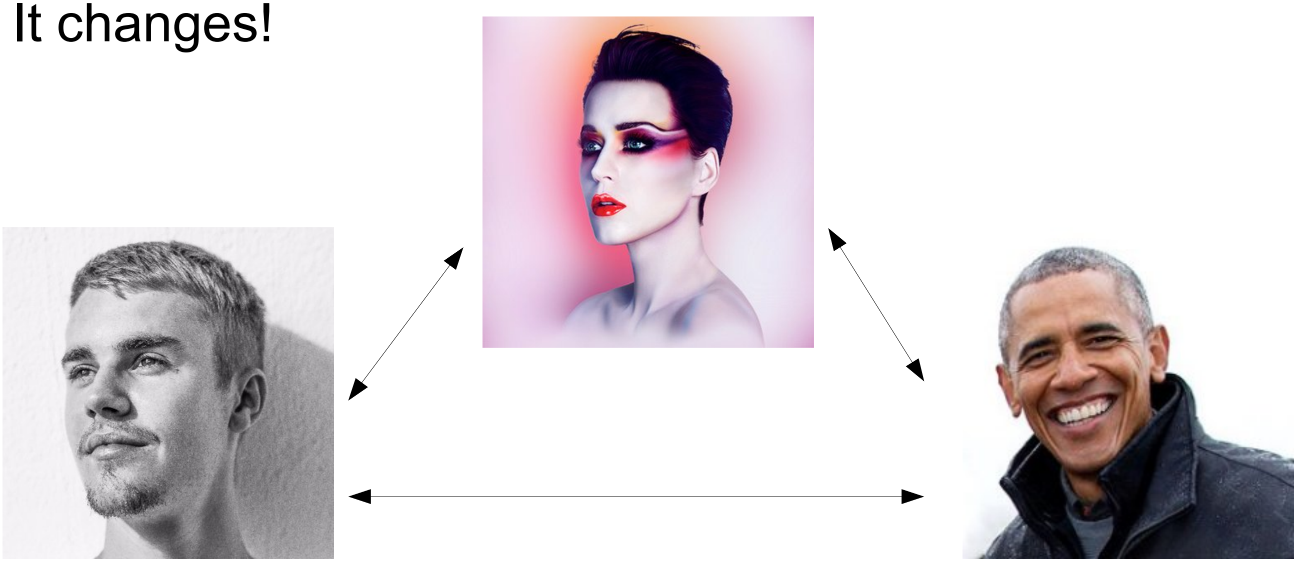
- Ordering of Events
  - Necessary for many applications:
    - Collaborative editing
    - Distributed storage
    - Resource allocation
- Logical time
  - Happens-before and causality
  - Lamport clocks
  - Vector clocks
- Today: Snapshots of global state

Administrivia

- Coding practices
  - Use good practice!
  - Variable naming, comments, structure
  - Loop invariants
- *Debugging other students' code is an AI violation*
  - No other student's code should *ever* be on your machine!

Today's Question

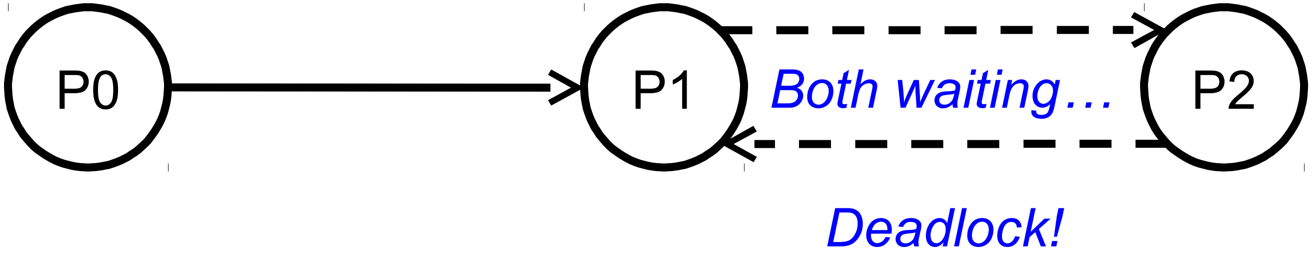
- Example Question: Who has the most Twitter followers?
- Are there challenges to answering this question?
  - It changes!



- What do we need?
  - A snapshot of the social network graph at a particular time

Today's Question

- Distributed debugging

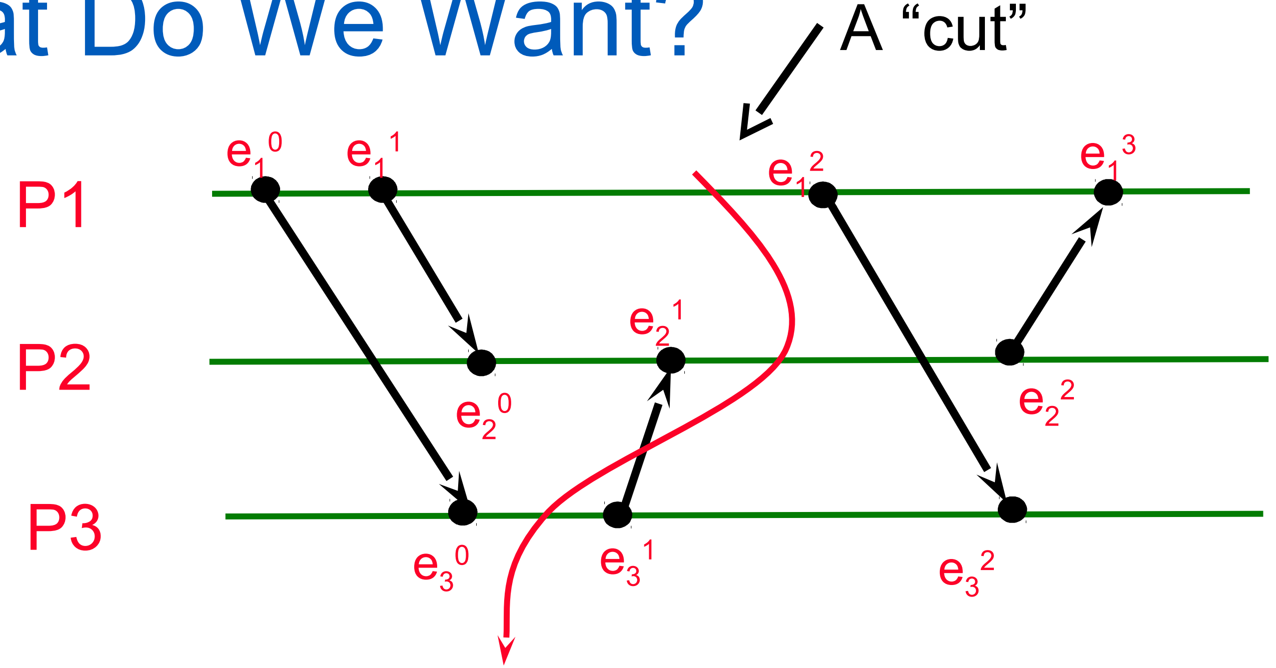


- How do you debug this?
  - Log in to one machine and see what happens
  - Collect logs and see what happens
  - Take a global snapshot!

What is a Snapshot?

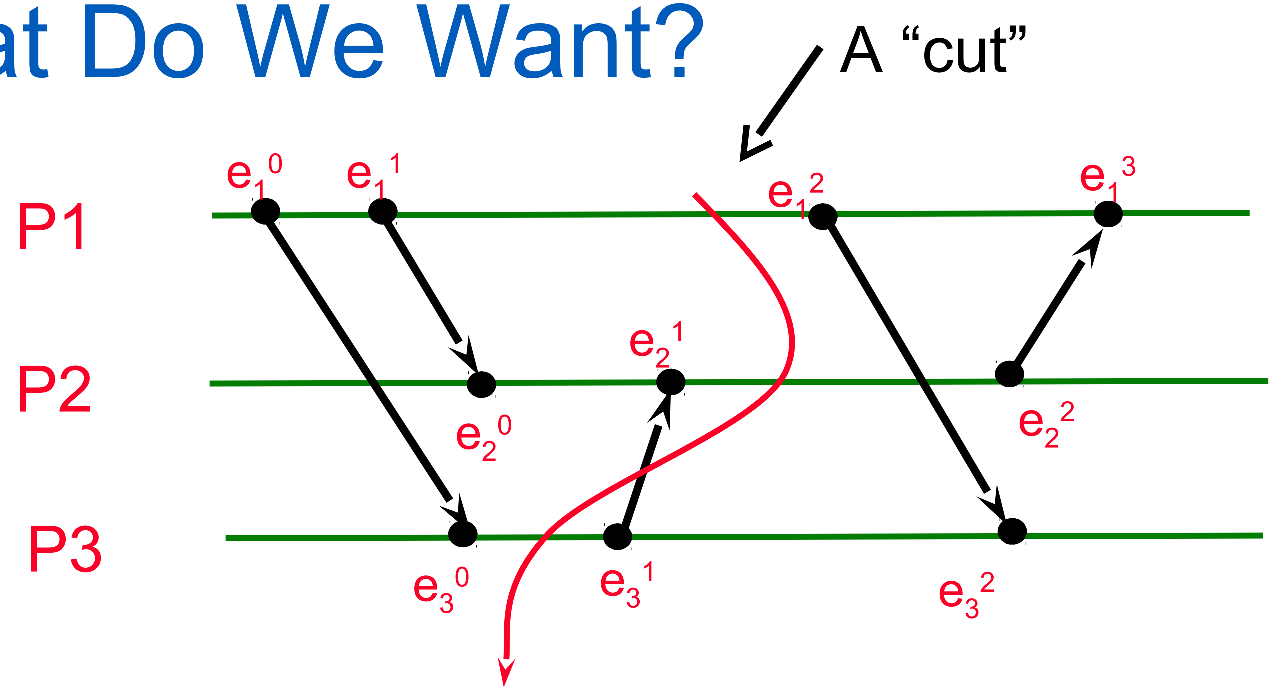
- Single process snapshot
  - A snapshot of local state: *e.g.*, memory dump, stack trace, *etc.*
- Multi-process snapshot
  - Snapshots of all process states
  - Network snapshot
    - All messages in the network

What Do We Want?



- Would you say this is a good snapshot?
  - "Good": we can explain all the causality, including messages
  - No, because e_2^1 might have been caused by e_3^1.

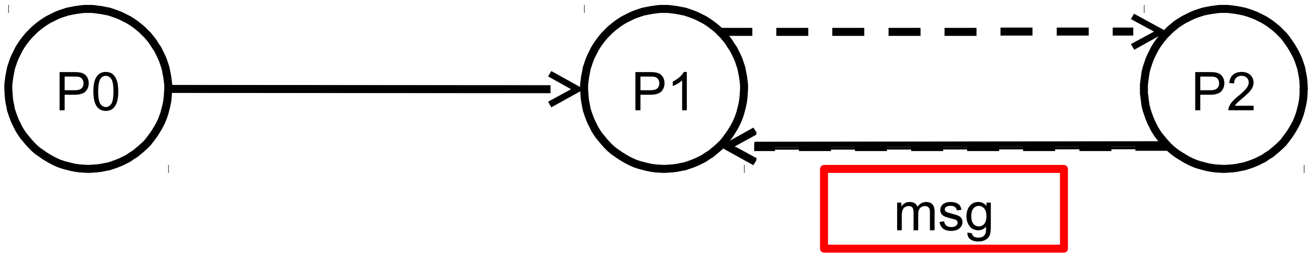
What Do We Want?



- Three things we want.
  - Per-process state
  - Messages that are causally related to each and every local snapshot and in flight
  - All events that happened before each event in the snapshot

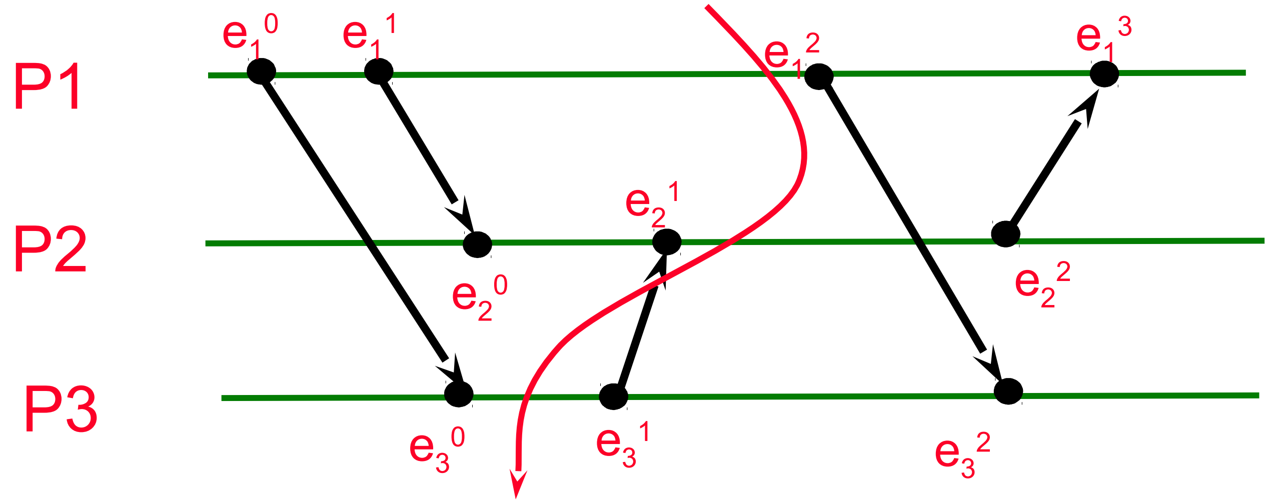
Obvious First Try

- Synchronize clocks of all processes
  - Ask all processes to record their states at known time *t*
- Problems?
  - Only approximate time synchronization is possible
  - Another issue?



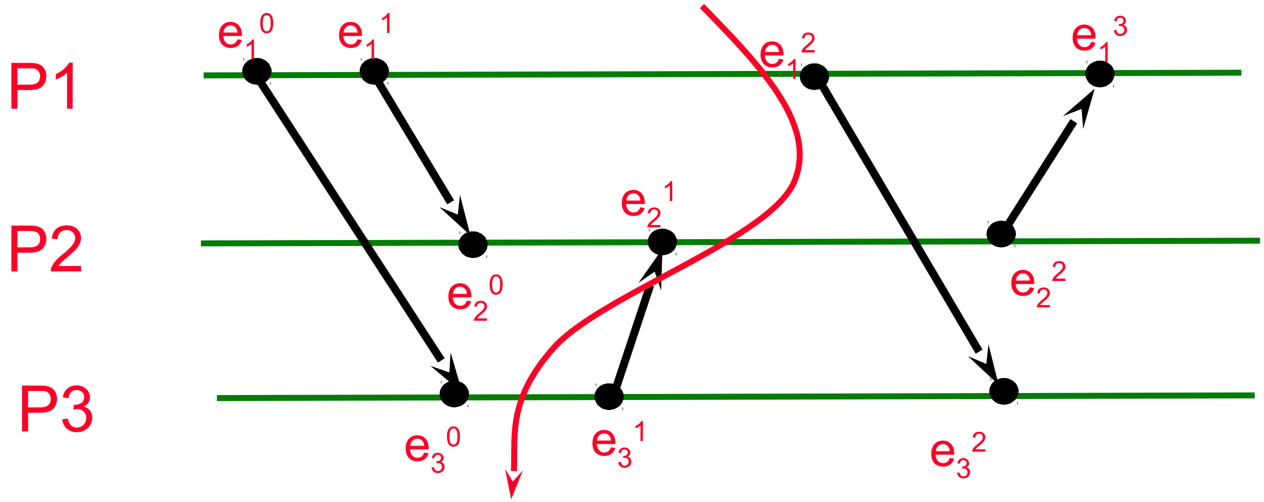
  - Does not record the state of messages in the channels
- Again: causality is sufficient!
- What we need: logical global snapshot
  - The state of each process
  - Messages in transit in all communication channels

How to Do It? Definitions



- For a process P_i, where events $e_i^0, e_i^1, \ldots$ *occur,*
  - *history(P_i)* = $h_i = \langle e_i^0, e_i^1, \ldots \rangle$
  - *prefix history(P_i^k)* = $h_i^k = \langle e_i^0, e_i^1, \ldots, e_i^k \rangle$
  - S_i^k : P_i's state immediately after k[th] event

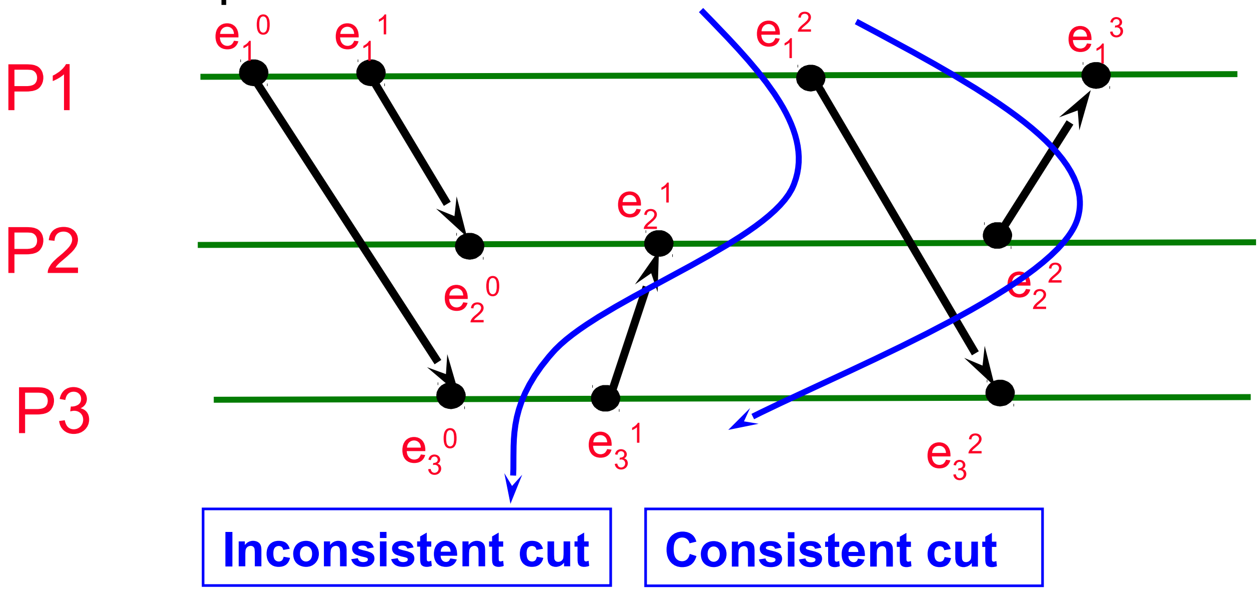
How to Do It? Definitions



- For a set of processes P_1, …,P_i, … :
  - Global history: $H = \cup_i (h_i)$
  - Global state: $S = \cup_i (S_i^{k_i})$
  - A *cut* $C \subseteq H = h_1^{c1} \cup h_2^{c2} \cup \ldots \cup h_n^{cn}$
  - The frontier of $C = \{e_i^{ci}, i = 1,2, \ldots n\}$

Consistent States

- A cut C is consistent if and only if
  - $\forall_{e \in C}$ *(if $f \rightarrow e$ then $f \in C$)*
- A global state S is consistent if and only if
  - it corresponds to a consistent cut



Why Consistent States?

- #1: For each event, you can trace back the causality.
- #2: Consider a state machine
  - The execution of a distributed system as a series of transitions between global states: S0 → S1 → S2 → …
  - …where each transition happens with one single action from a process (*i.e.*, local process instruction, send, and receive)
  - *i.e.*, the clock "ticks" in the logical clocks of last lecture
  - Each state (S0, S1, S2, …) is a consistent state

The Snapshot Algorithm: Assumptions

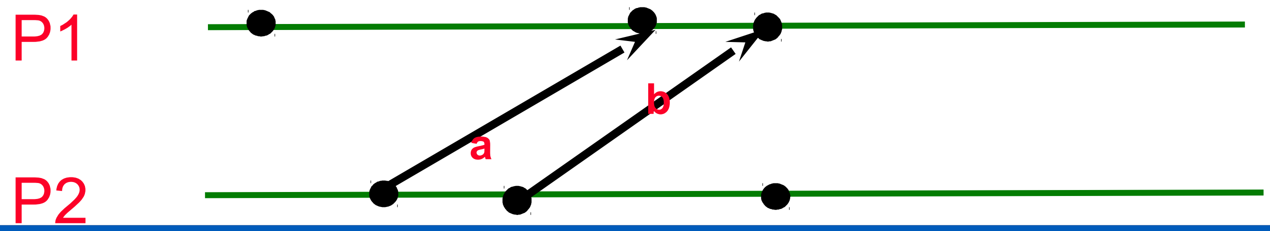
- There is a communication channel between each pair of processes
  - N-1 input and N-1 output channels at each process
- Communication channels are unidirectional and FIFO-ordered (important point)
- No failures, all messages arrive intact and exactly once
- Any process may initiate the snapshot
- Snapshot does not interfere with normal execution
- Each process is able to record its state and the state of its incoming channels (no central collection)

Single Process vs. Multiple Processes

- Single process snapshot
  - A snapshot of local state; *e.g.*, memory dump, stack trace, *etc.*
- Multi-process snapshot
  - Snapshots of all process states
  - Network snapshot: all messages in the network
- Two questions:
  - #1: When should a local snapshot be taken at each process so that the collection of snapshots forms a consistent global state?
  - #2: How are messages in flight captured?

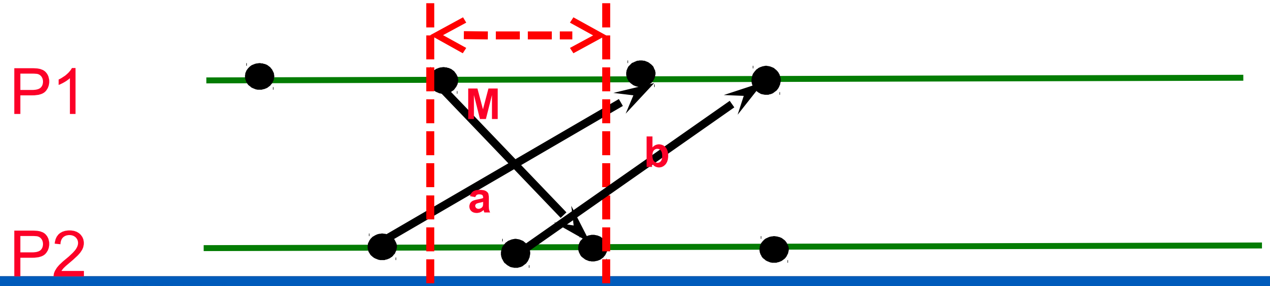
The Snapshot Algorithm

- Clock-synced snapshot (instantaneous snapshot)
- Process snapshots and network messages at time t
- Need to capture:
  - Local snapshots of P1 & P2
  - Messages in the network (message *a*, since message *a* is causally related to P2's snapshot)
- We can't quite do it due to (i) imperfect clock sync and (ii) no help from the network.



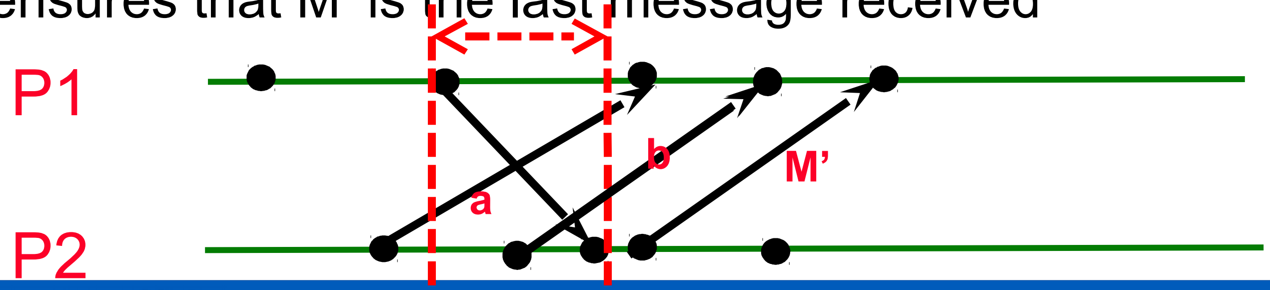
The Snapshot Algorithm [2]

- Logical snapshot (not instantaneous)
  - Goal: capture causality (events and messages)
  - A process initiates the snapshot by sending a message (see the diagram). There is delay in this communication.
  - Need to capture all network messages during the delay (not at an instantaneous moment)
- We need to capture:
  - Local snapshots of P1 & P2 (but now at different times).
  - Messages in flight that are *causally related to each and every local snapshot; e.g.*, messages *a* and *b* for P2's snapshot.
  - How?



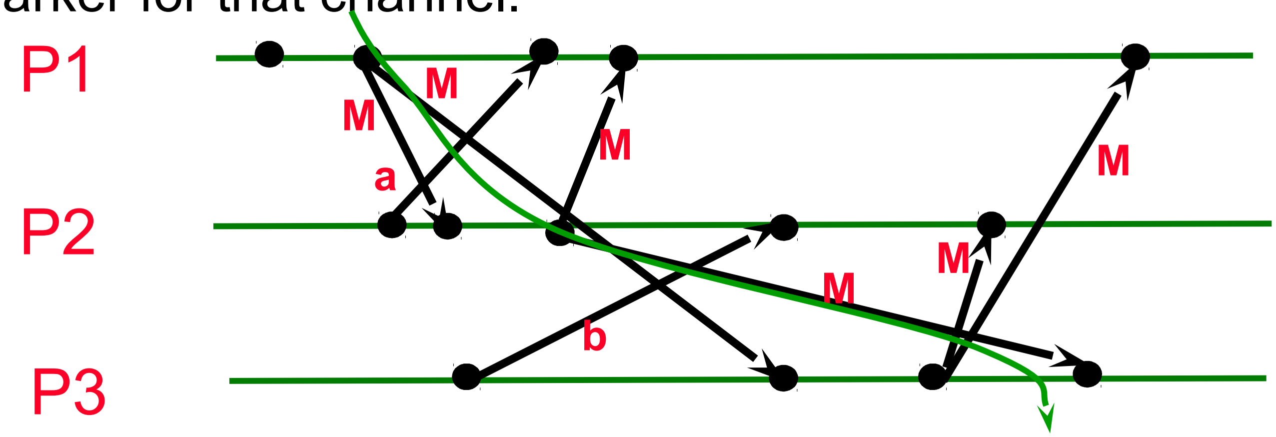
The Snapshot Algorithm [3]

- P1 needs to record all causally-related messages.
  - All the messages already in the network.
  - All the messages sent during the delay.
- For messages already in the network,
  - P1 starts recording as soon as it sends the marker M
  - The messages already in the network will eventually arrive at P1
- For messages sent during the delay,
  - P2 sends a marker M’ to tell P1 that a local snapshot was taken
  - This marks the end of the delay
  - FIFO ensures that M’ is the last message received



The Snapshot Algorithm [4]

- Basic idea: marker broadcast & recording
  - The initiator broadcasts a "marker" message to everyone else
  - If a process receives a marker for the first time, it takes a local snapshot, starts recording all incoming messages, and broadcasts a marker again to everyone else.
  - A process stops recording for each channel when it receives a marker for that channel.



The Snapshot Algorithm [5]

1. Marker sending rule for initiator process P_0
   - After P_0 has recorded its own state
     - for each outgoing channel C, send a marker message on C
2. Marker receiving rule for a process P_k

   on receipt of a marker on channel C:
   - if P_k has not yet recorded its own state
     - record P_k's own state
     - record the state of C as "empty"
     - for each outgoing channel C, send a marker on C
     - turn on recording of messages over other incoming channels
   - else
     - record the state of C as all the messages received over C since P_k saved its own state; stop recording state of C

Chandy and Lamport's Snapshot [1]

Marker receiving rule for process p_i

On p_i's receipt of a marker message over channel *c*:

if (p_i has not yet recorded its state) *it*

records its process state now;

records the state of *c* as the empty set;

turns on recording of messages arriving over other incoming channels;

else

p_i records the state of *c* as the set of messages it has received over *c* since it saved its state.

end if

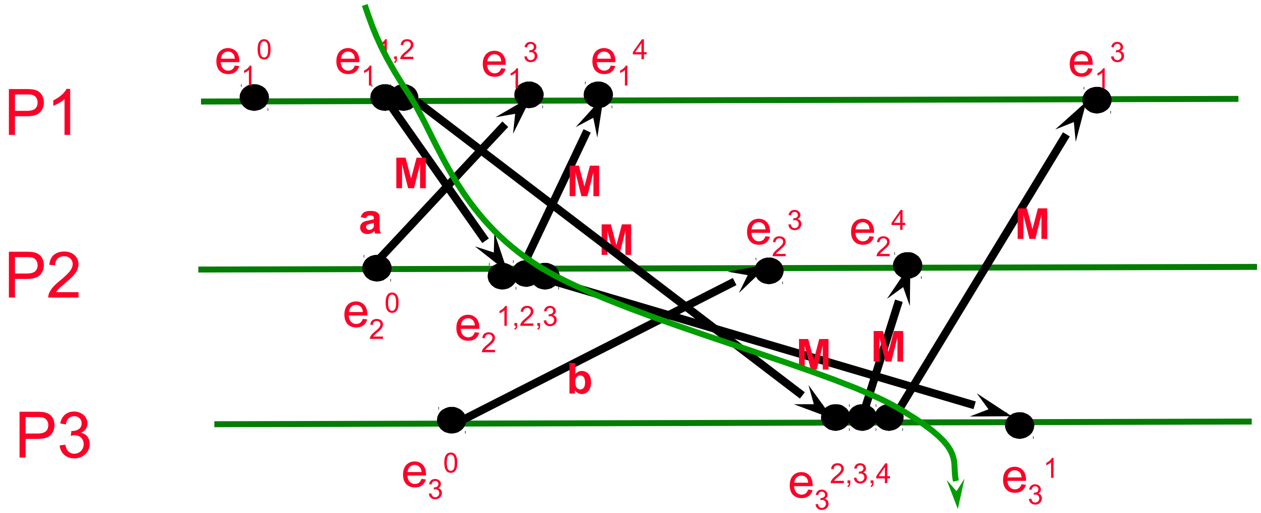
Marker sending rule for process p_i

After p_i has recorded its state, *for each* outgoing channel *c*:

p_i sends one marker message over *c*

(before it sends any other message over *c*).

Exercise



1- P1 initiates snapshot: records its state (S1); sends Markers to P2 & P3; turns on recording for channels C21 and C31

2- P2 receives Marker over C12, records its state (S2), sets state(C12) = {} sends Marker to P1 & P3; turns on recording for channel C32

3- P1 receives Marker over C21, sets state(C21) = {a}

4- P3 receives Marker over C13, records its state (S3), sets state(C13) = {} sends Marker to P1 & P2; turns on recording for channel C23

5- P2 receives Marker over C32, sets state(C32) = {b}

6- P3 receives Marker over C23, sets state(C23) = {}

7- P1 receives Marker over C31, sets state(C31) = {}

One Provable Property

- The snapshot algorithm gives a consistent cut
- Meaning,
  - Suppose e_i is an event in P_i, and e_j is an event in P_j
  - If $e_i \rightarrow e_j$, and e_j is in the cut, then e_i is also in the cut.
- Proof sketch: proof by contradiction
  - Suppose e_j is in the cut, but e_i is not.
  - Since $e_i \rightarrow e_j$, there must be a sequence M of messages that leads to the relation.
  - Since e_i is not in the cut (our assumption), a marker should have been sent before e_i, and also before all of M.
  - Then P_j must have recorded a state before e_j, meaning e_j is not in the cut. (Contradiction)

Summary

- Global state
  - A union of all process states
  - Consistent global state *vs.* inconsistent global state
- The snapshot algorithm
  - Take a snapshot of the local state
  - Broadcast a marker message to tell other processes
  - Start recording all incoming messages for each channel until receiving a marker on that channel
  - Outcome: a consistent global state

References

[1] Leslie Lamport, K. Mani Chandy. *Distributed Snapshots: Determining Global States of a Distributed System*. ACM Transactions on Computer Systems Vol 3 No 1. February 1985. **Required Reading.** http://research.microsoft.com/users/lamport/pubs/chandy.pdf

Acknowledgements

- These slides are by Steve Ko, lightly modified and used with permission by Ethan Blanton
- These slides contain material developed and copyrighted by Indranil Gupta at UIUC.