Computer Science 425
Distributed Systems

CS 425 / CSE 424 / ECE 428

Fall 2012

Indranil Gupta (Indy)

September 13, 2012

Lecture 6

Global Snapshots

Reading: Sections 14.5

Example of a Global Snapshot

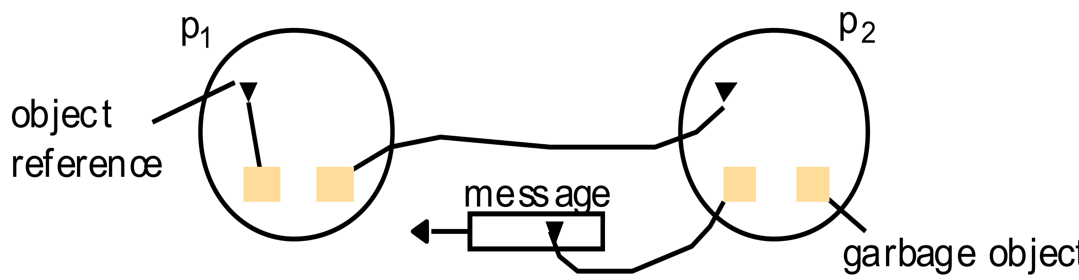
[United Nations photo by Paul Skipworth for Eastman Kodak Company ©1995]

The distributed version is challenging and important

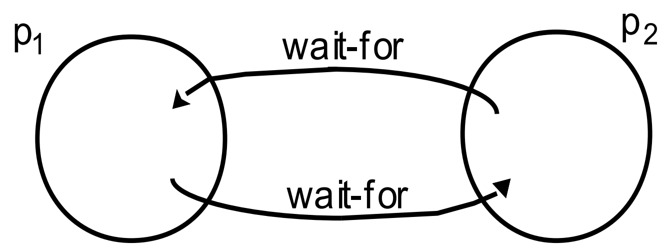
- More often each country's premier were sitting in their respective capital, and sending messages to each other.
- That's the challenge of distributed global snapshots!
- In a cloud: multiple servers (for a service/ application) handling multiple concurrent events and interacting with each other
- The ability to obtain a global photograph of the system is important

Detecting Global Properties

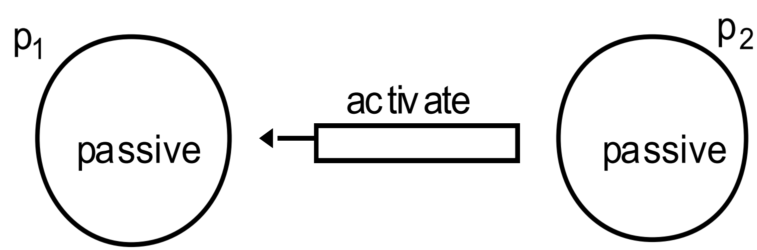
a. Garbage collection



b. Deadlock



c. Termination



Algorithms to Find Global States

- **Why?**
  - (Distributed) garbage collection [think Grid application]
  - (Distributed) deadlock detection, termination [think database transactions]
  - Global states useful for detecting <u>stable predicates</u> : once true always stays true (unless you do something about it)
    - » e.g., once a deadlock, always stays a deadlock
- **What?**
  - Global state=states of all processes + states of all communication channels
  - Capture the instantaneous *state* of <u>each process</u>
  - And the instantaneous *state* of <u>each communication channel</u>, i.e., *messages* in transit on the channels
- **How?**
  - We'll see this lecture!

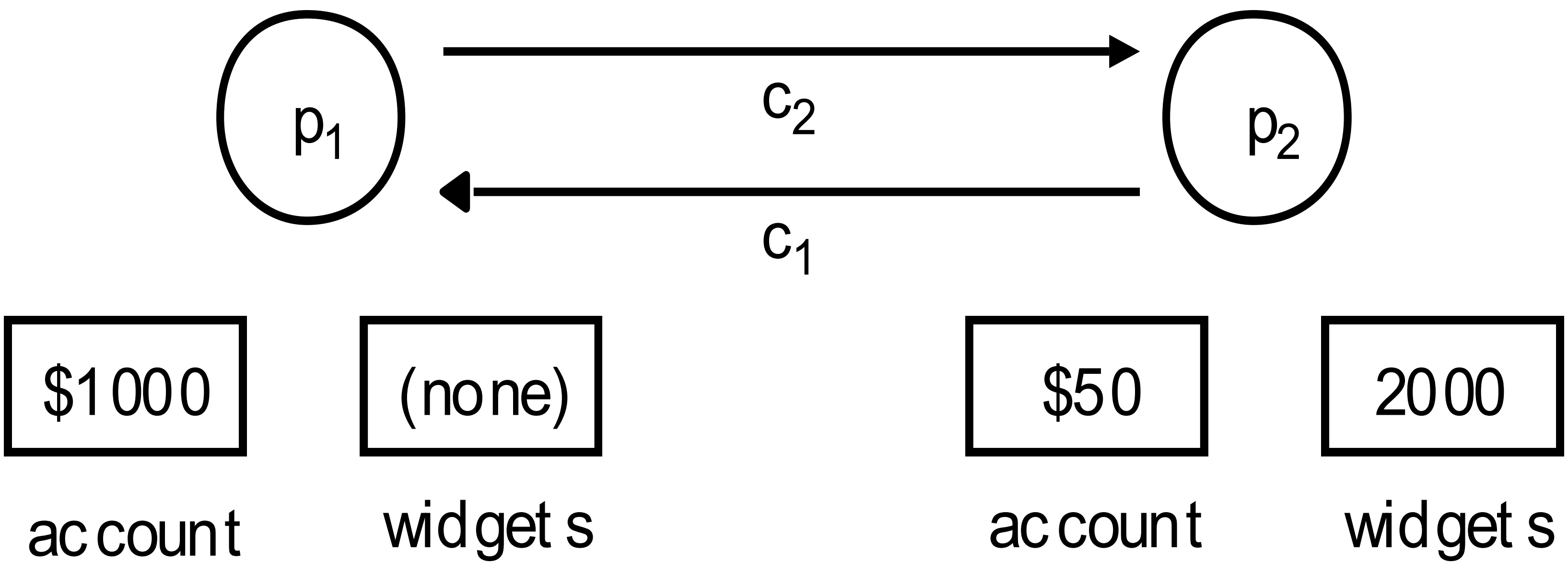
Obvious First Solution…

- Synchronize clocks of all processes
- Ask all processes to record their states at known time t

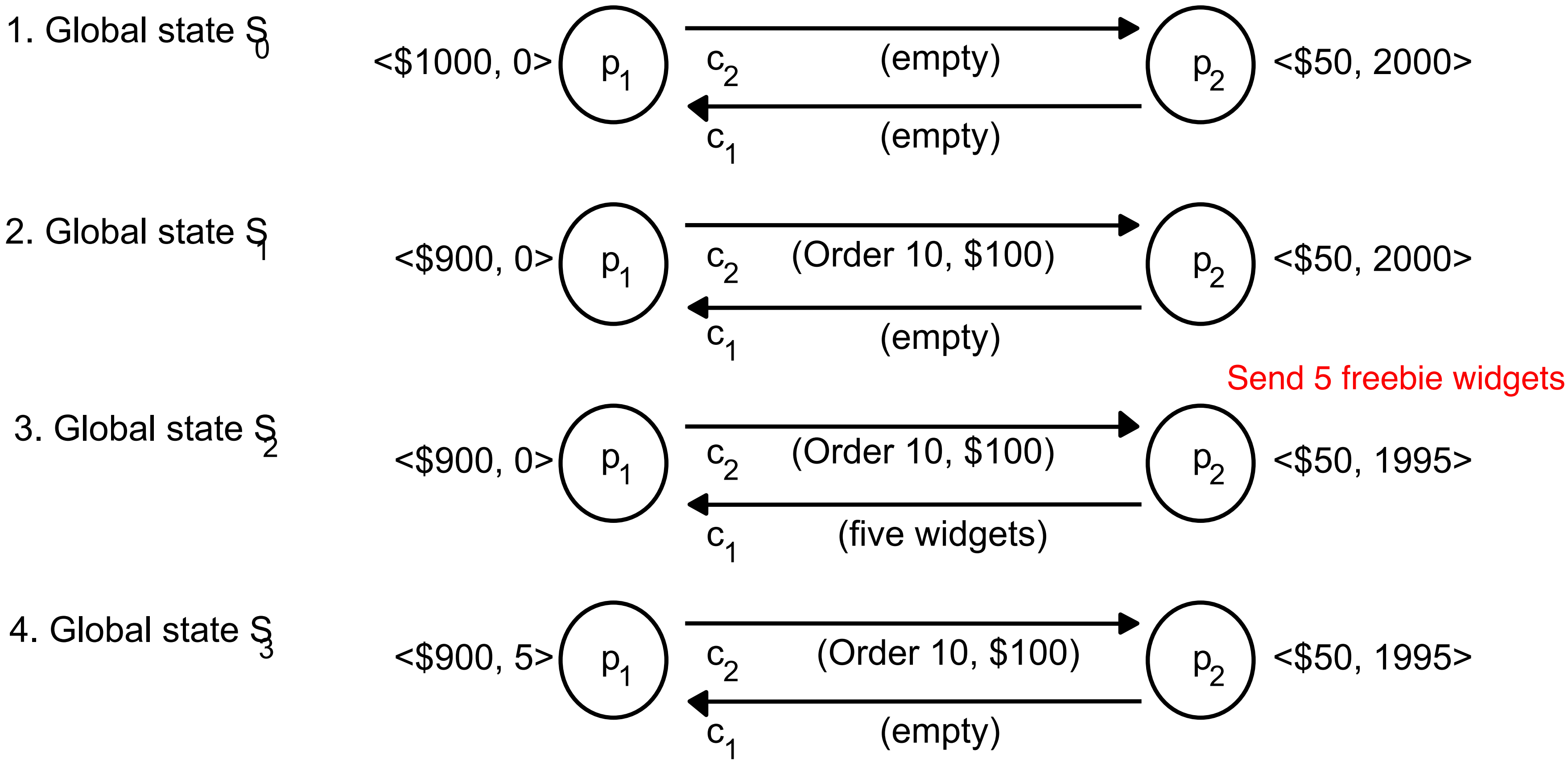
- Problems?
  - Time synchronization possible only approximately (but distributed banking applications cannot take approximations)
  - Does not record the state of messages in the channels

- Again: synchronization not required – causality is enough!

Two Processes and Their Initial States



Execution of the Processes



Process Histories and States

- For a process P_i, where events e_i^0, e_i^1, … *occur:*

  history(P_i) = h_i = $<e_i^0, e_i^1, \ldots >$

  prefix history(P_i^k) = h_i^k = $<e_i^0, e_i^1, \ldots, e_i^k >$

  S_i^k : P_i's state immediately after k^{th} event

- For a set of processes P_1, …,P_i, …. :

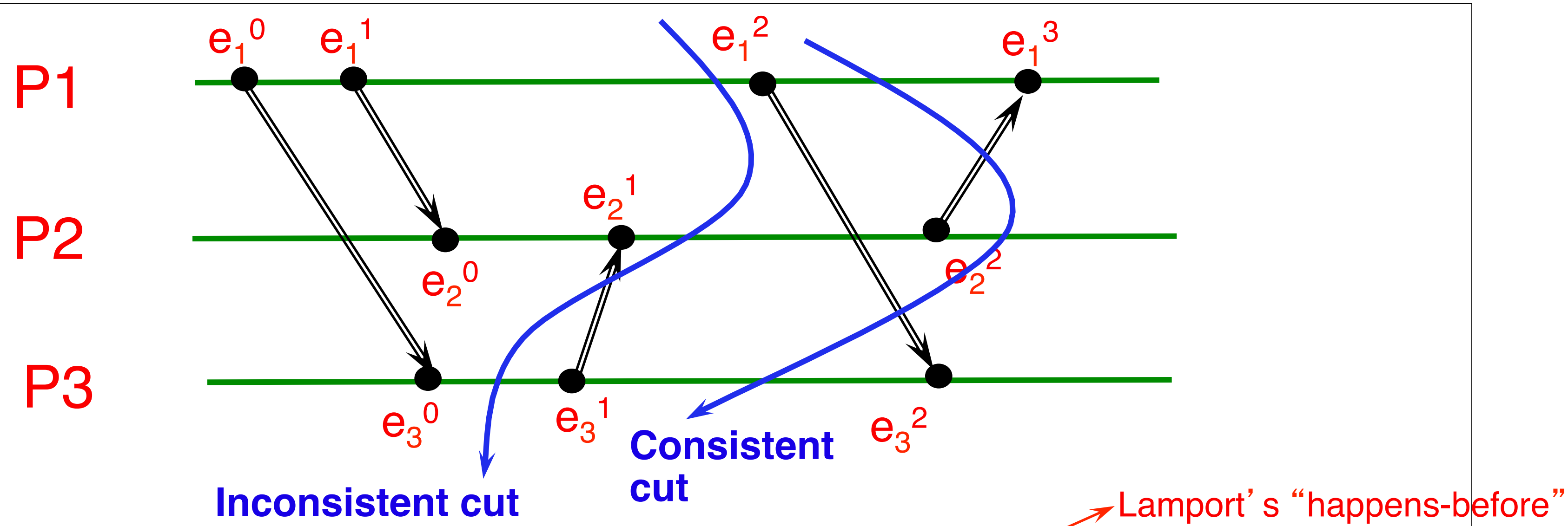
  global history: $H = \cup_i (h_i)$

  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**

$$\forall_{e \in C} \, (\text{if } f \rightarrow e \text{ then } f \in C)$$

- **A global state S is consistent if and only if it corresponds to a consistent cut**
- **A consistent cut == a global snapshot**

The "Snapshot" Algorithm

- **Problem:** Record a set of process and channel states such that the combination is a global snapshot/consistent cut.
- ***System Model:***
  - There is a uni-directional communication channel between each ordered process pair (Pj → Pi and Pi → Pj)
  - Communication channels are FIFO-ordered
  - No failure, all messages arrive intact, exactly once
  - Any process may initiate the snapshot (by sending "Marker" message)
  - 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)

The "Snapshot" Algorithm (2)

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 over channel C**
   - if P_k has not yet received a marker
     - 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' Algorithm

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.

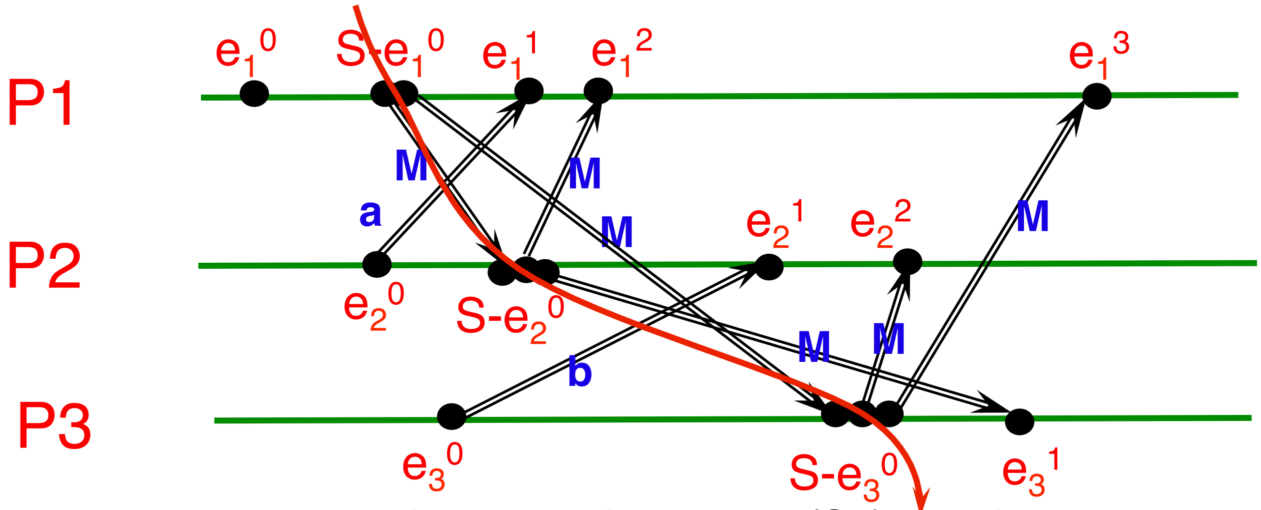
end if

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).

Snapshot Example



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) = {}

Provable Assertion: Chandy-Lamport algo. determines a consistent cut

- Let e_i and e_j be events occurring at p_i and p_j, respectively such that $e_i \rightarrow e_j$
- The snapshot algorithm ensures that

  if e_j is in the cut then e_i is also in the cut.
- if $e_j \rightarrow$ <p_j records its state>, then it must be true that $e_i \rightarrow$ <p_i records its state>.
  - By contradiction, suppose <p_i records its state> $\rightarrow e_i$
  - Consider the path of app messages (through other processes) that go from $e_i \rightarrow e_j$
  - Due to FIFO ordering, markers on above path precede regular app messages
  - Thus, since <p_i records its state> $\rightarrow e_i$, it must be true that p_j received a marker before e_j
  - Thus e_j is not in the cut => contradiction

Global States useful for detecting Global Predicates

- A cut is consistent if and only if it does not violate causality
- A *Run* is a total ordering of events in H that is consistent with each h_i's ordering
- A Linearization is a run consistent with happens-before ($\rightarrow$) relation in H.
- Linearizations pass through consistent global states.
- A global state S_k is reachable from global state S_i, if there is a linearization, L, that passes through S_i and then through S_k.
- The distributed system evolves as a series of transitions between global states S_0 , S_1 , ….

Global State Predicates

- A *global-state-predicate* is a function from the set of global states to {true, false} , e.g., deadlock, termination
- A global state S0 satisfies liveness property P iff:

  liveness($P(S_0)$) $\equiv \exists L_{\in \text{ linearizations from S0}}$ L passes through a S_L & $P(S_L)$ = true
- Ex: P(S) = the computation will terminate from S
- A global state S0 satisfies this safety property P if:

  safety($P(S_0)$) $\equiv \forall S$ reachable from S_0, P(S) = false
- Ex: P(S) = S has a deadlock
- Global states useful for detecting *stable* global-state-predicate: it is one that once it becomes true, it remains true in subsequent global states, e.g., an object O is orphaned, or deadlock
  - A stable predicate may be a safety or liveness predicate

Quick Note – Liveness versus Safety

Can be confusing, but terms are very important:

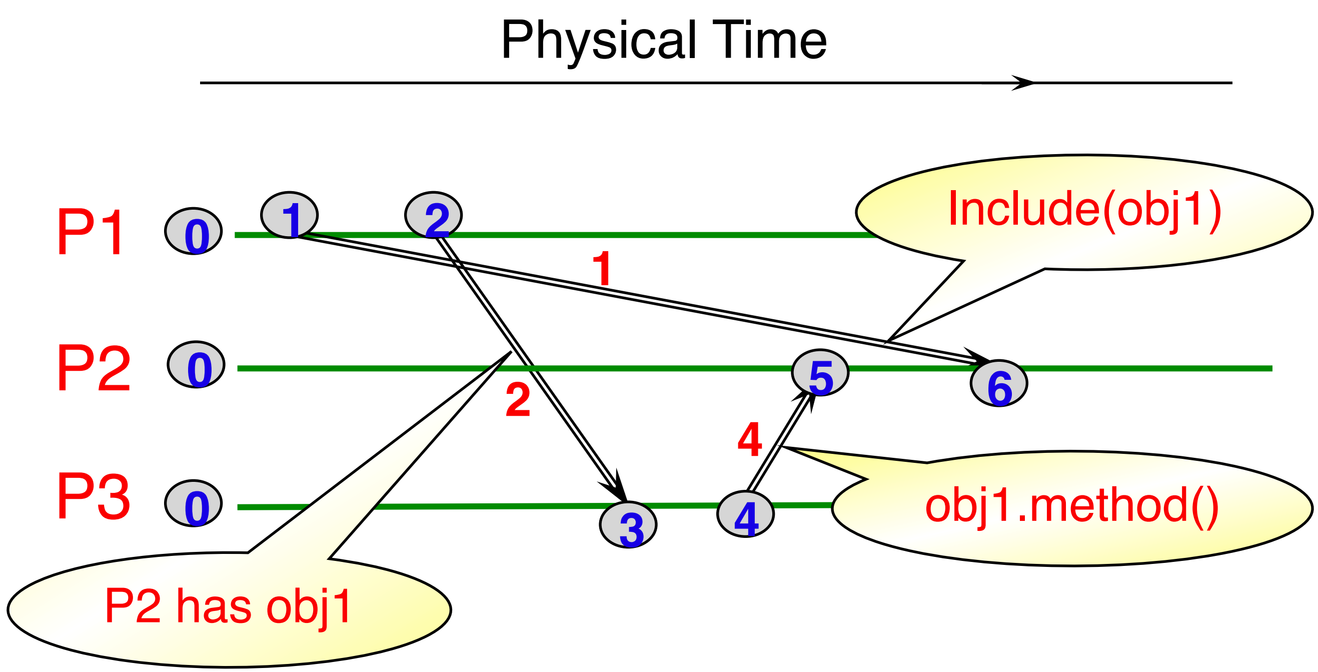
- Liveness=guarantee that something good will happen, eventually
  - "Guarantee of termination" is a liveness property
  - Guarantee that "at least one of the atheletes in the 100m final will win gold" is liveness
  - A criminal will eventually be jailed
  - Completeness in failure detectors
- Safety=guarantee that something bad will never happen
  - Deadlock avoidance algorithms provide safety
  - A peace treaty between two nations provides safety
  - An innocent person will never be jailed
  - Accuracy in failure detectors
- Can be difficult to satisfy both liveness and safety!

Summary, Announcements

- This class: importance of global snapshots, Chandy and Lamport algorithm, violation of causality
- Reading for next week: Sections 15.4, 4.3 (and parts of Chapter 5)
- MP1 due this Sunday at midnight
  - Demos next Tuesday (or Monday)
  - Watch Piazza for signup sheets for demos
- By now you should have a working system, and should have written most tests for it

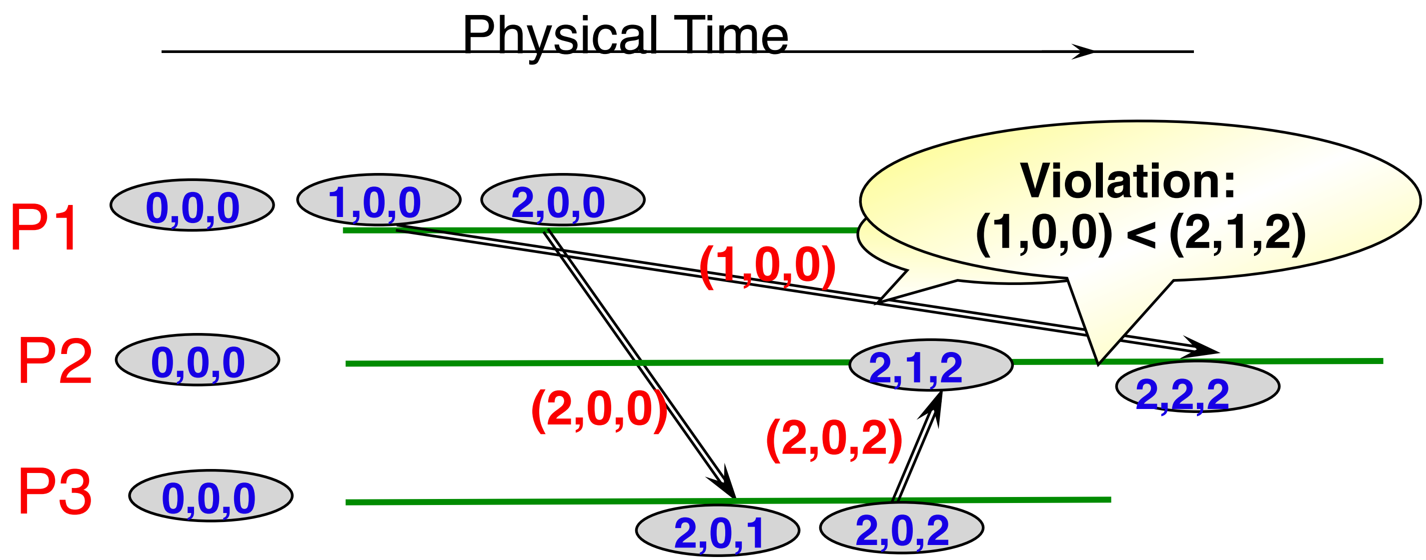
Optional Slides

Side Issue: Causality Violation



- Causality violation occurs when order of messages causes an action based on information that another host has not yet received.
- In designing a DS, potential for causality violation is important

Detecting Causality Violation



- Potential causality violation can be detected by vector timestamps.
- If the vector timestamp of a message is less than the local vector timestamp, on arrival, there is a potential causality violation.