CSE 486/586 Distributed Systems
Global States

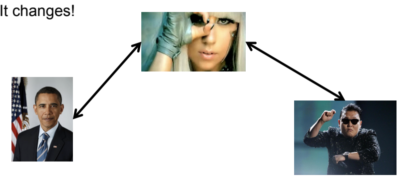
Steve Ko
Computer Sciences and Engineering
University at Buffalo

Last Time

- Ordering of events
  - Many applications need it, e.g., collaborative editing, distributed storage, etc.
- Logical time
  - Lamport clock: single counter
  - Vector clock: one counter per process
  - Happens-before relation shows causality of events

Today's Question

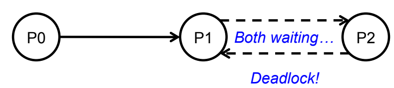
- Example question: who has the most friends on Facebook?
- Challenges to answering this question?
  - It changes!



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

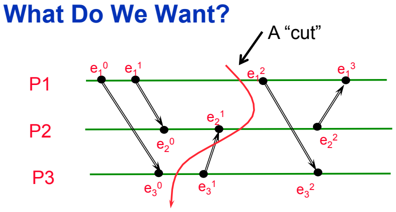
Today's Question

- Distributed debugging



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

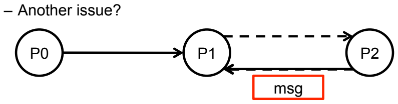
What Do We Want?



- Would you say this is a good snapshot?
  - No because e_2^1 might have been caused by e_3^1.
- Three things we want.
  - Per-process state
  - Messages in flight
  - All events that happened before each event in the snapshot

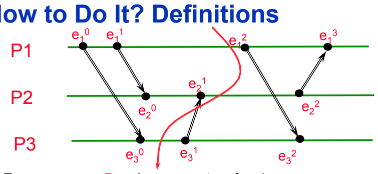
Obvious First Try

- Synchronize clocks of all processes
  - Ask all processes to record their states at known time *t*
- Problems?
  - Time synchronization possible only approximately
  - Another issue?



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

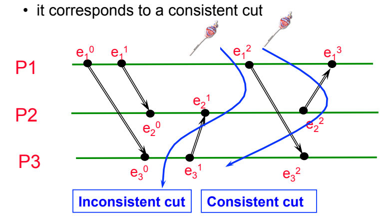
How to Do It? Definitions



- 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
- For a set of processes $P_1, \ldots, P_i, \ldots$:
  - 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: Back to the state machine (from the last lecture)
  - 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 event, send, and receive)
  - Each state (S0, S1, S2, …) is a consistent state.

CSE 486/586 Administrivia

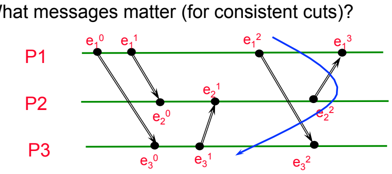
- PA2-A deadline: This Friday
- Please come and ask questions during office hours.

The "Snapshot" Algorithm: Assumptions

- There is a communication channel between each pair of processes (@each process: N-1 in and N-1 out)
- Communication channels are unidirectional and FIFO-ordered
- No failure, all messages arrive intact, 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
  - Just a snapshot of the 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 messages matter (for consistent cuts)?



Single Process vs. Multiple Processes

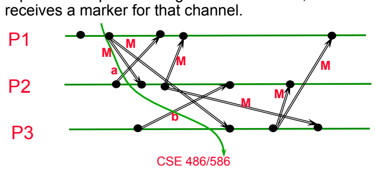
- For each local snapshot, we want to record all messages in the network that are a result of a send event reflected in the snapshot.
- How?
  - Each sender can record it, but probably with extra overhead.
  - Alternatively, each receiver can record it---we need to know when to start and when to stop.
  - As soon as a process takes a local snapshot, it starts recording incoming messages.
  - For each process pair, a process stops recording when another process takes a snapshot.

The "Snapshot" Algorithm

- Goal: records a set of process and channel states such that the combination is a consistent global state.
- Two questions:
  - #1: When to take a local snapshot at each process so that the collection of them can form a consistent global state? (**Process snapshot**)
  - #2: How to capture messages in flight sent before each local snapshot? (**Network snapshot**)
- Brief answer for #1
  - The initiator broadcasts a "marker" message to everyone else ("hey, take a local snapshot now")
- Brief answer for #2
  - 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. ("hey, I've sent all my messages before my local snapshot to you, so stop recording my messages.")
  - A process stops recording, when it receives a marker for each channel.

The "Snapshot" Algorithm

- Basic idea: marker broadcast & recording
  - The initiator broadcasts a "marker" message to everyone else ("hey, take a local snapshot now")
  - 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. ("hey, I've sent all my messages before my local snapshot to you, so stop recording my messages.")
  - A process stops recording for each channel, when it receives a marker for that channel.



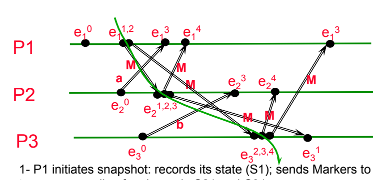
The "Snapshot" Algorithm

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 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

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've been sent before e_i, and also before all of M.
  - Then P_j must've recorded a state before e_j, meaning, e_j is not in the cut. (Contradiction)

Another Provable Property

- Can we evaluate a stable predicate?
  - Predicate: a function: (a global state) → {true, false}
  - Stable predicate: once it's true, it stays true the rest of the execution, e.g., a deadlock.
- A stable predicate that is true in S-snap must also be true in S-final
  - S-snap: the recorded global state
  - S-final: the global state immediately after the final state-recording action.
- Proof sketch
  - The necessity for a proof: S-snap is a snapshot that may or may not correspond to a snapshot from the real execution.
  - Strategy: prove that it's part of what could have happened.
  - Take the actual execution as a linearization
  - Re-order the events to get another linearization that passes through S-snap.

Related Properties

- Liveness (of a predicate): guarantee that something good will happen eventually
  - For any linearization starting from the initial state, there is a reachable state where the predicate becomes true.
  - "Guarantee of termination" is a liveness property
- Safety (of a predicate): guarantee that something bad will never happen
  - For any state reachable from the initial state, the predicate is false.
  - Deadlock avoidance algorithms provide safety
- Liveness and safety are used in many other CS contexts.

Summary

- Global states
  - 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" msg to tell other processes to record
  - Start recording all msgs coming in for each channel until receiving a "marker"
  - Outcome: a consistent global state

Acknowledgements

- These slides contain material developed and copyrighted by Indranil Gupta at UIUC.