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

What Do We Want?

- Would you say this is a good snapshot?
  - No because $e_2$ might have been caused by $e_1$.
- 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

Today’s Question

- Distributed debugging
  - P0
  - P1
  - P2
  - Deadlock!
- How do you debug this?
  - Log in to one machine and see what happens
  - Collect logs and see what happens
  - Taking a global snapshot

P0
P1
P2
How to Do It? Definitions

- For a process \( P_i \), where events \( e_i^0, e_i^1, \ldots \) occur,
  - history\( (P_i) = h_i = <e_i^0, e_i^1, \ldots> \)
  - prefix history\( (P_i^k) = h_i^k = <e_i^0, e_i^1, \ldots, e_i^k> \)
- For a set of processes \( P_1, \ldots, P_i, \ldots \):
  - Global history: \( H = \bigcup_i (h_i) \)
  - Global state: \( S = \bigcup_i (S_i^k) \)
  - A cut \( C \subseteq H = h_1^{c_1} \cup h_2^{c_2} \cup \ldots \cup h_n^{c_n} \)
  - The frontier of \( C = \{ e_i^{c_i}, i = 1, 2, \ldots, n \} \)

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: \( S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \ldots \)
  - where each transition happens with one single action from a process (i.e., local process event, send, and receive)
  - Each state \( (S_0, S_1, S_2, \ldots) \) is a consistent state.

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)

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

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?
  - #2: How to capture messages in flight sent before each local 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, 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.

### Chandy and Lamport’s Snapshot

**Marker receiving rule for process p**

On p's receipt of a marker message over channel c:
- If (p has not yet recorded its state) it records its process state now;
- turns on recording of messages arriving over other incoming channels;
- p records the state of c as the set of messages it has received over c since it saved its state.

**Marker sending rule for process p**

After p has recorded its state, for each outgoing channel c:
- p sends one marker message over c (before it sends any other message over c).

### One Provable Property

- The snapshot algorithm gives a consistent cut
- Meaning,
  - Suppose e<sub>i</sub> is an event in P<sub>i</sub> and e<sub>j</sub> is an event in P<sub>j</sub>,
  - If e<sub>i</sub> → e<sub>j</sub> and e<sub>j</sub> is in the cut, then e<sub>i</sub> is also in the cut.
- Proof sketch: proof by contradiction
  - Suppose e<sub>i</sub> is in the cut, but e<sub>j</sub> is not.
  - Since e<sub>i</sub> → e<sub>j</sub> there must be a sequence M of messages that leads to the relation.
  - Since e<sub>i</sub> is not in the cut (our assumption), a marker should’ve been sent before e<sub>i</sub> and also before all of M.
  - Then P must’ve recorded a state before e<sub>i</sub>, meaning, e<sub>i</sub> 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.