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?
• A "cut"

  P1
  e_1^0 e_1^1 e_1^2 e_1^3

  P2
  e_2^0 e_2^1 e_2^2

  P3
  e_3^0 e_3^1 e_3^2

• Would you say this is a good snapshot?
  – No because e_2^1 might have been caused by e_1^3.
• 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?

  P0
  msg

  P1

  P2

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

  Deadlock!

• 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
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)

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 \rightarrow S1 \rightarrow S2 \rightarrow ...
  - ...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.

Consistent States
- A cut C is consistent if and only if
  \[ \forall e \in C \left( f \rightarrow e \text{ then } e \in C \right) \]
- 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: When to capture messages in flight 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 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.

Marker sending rule for process P0
1. P0 initiates snapshot: records its state (S0); sends markers to P1 & P2; turns on recording for channel C1.
2. P0 receives marker over C12, records its state (S1), sets state(C12) = {}.
3. P0 receives marker over C21, sets state(C21) = {a}.
4. P0 receives marker over C32, sets state(C32) = {b}.
5. P0 receives marker over C23, sets state(C23) = {}.
6. P0 receives marker over C31, sets state(C31) = {}.
7. After P0 has recorded its own state, for each outgoing channel C, P0 sends one marker message over C (before it sends any other message over C).

Marker receiving rule for process P1
- On P1’s receipt of a marker over channel C:
  - If (P1 has not yet recorded its state) it records its process state now; turns on recording of messages arriving over other incoming channels;
  - Else (P1 has recorded its state) it records the state of C as the set of messages it has received over C since it saved its state.

Proof sketch: proof by contradiction
- Suppose e1, e2, and e3 is the cut; meaning, e1 is in the cut, then e2 is also in the cut.
- Suppose e1 is in the cut, but e3 is not.
- Since e1 → e3, there must be a sequence M of messages that leads to the relation.
- Since e1 is not in the cut (our assumption), a marker should’ve been sent before e1, and also before all of M.
- Then P0 must’ve recorded a state before e1, meaning, e1 is not in the cut. (Contradiction)

Chandy and Lamport’s Snapshot

Marker receiving rule for process Pk
- On Pk’s receipt of a marker over channel C:
  - If (Pk has not yet recorded its state) it records its process state now; turns on recording of messages arriving over other incoming channels;
  - Else (Pk has recorded its state) it records the state of C as “empty”.

Marker sending rule for process Pk
- After Pk has recorded its state, for each outgoing channel C:
  - Pk sends one marker message over C (before it sends any other message over C).

Exercise

One Provable Property

• The snapshot algorithm gives a consistent cut
  - Suppose e1 → e2, and e2 is in the cut, then e1 is also in the cut.

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 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.