Distributed Snapshots ("Global States of Distributed Systems")

Why do we want global snapshots?

- Checkpoint / restart.
  - if you do a long-running distributed computation and you want to be resilient to failure, one possibility is to periodically checkpoint the global state, and recover to that state after failure
  - what about non-determinism?
    - message receive timing, in this model
    - external message arrival (outside of the model)
    - "box" picture

    - checkpoint can capture non-deterministic side-effects (e.g., a particle detector fires)
    - resume causes execution to proceed down a feasible execution path, but not necessarily the same execution path as before

- Stable predicate detection
  - definition: if a system is ever in a state in which a stable property holds, it will always hold from that point onwards, even given non-determinism
    - deadlock – why?
    - algorithm has terminated – why?
    - token has been lost – why?
  - note that many interesting properties are not stable
    - underload/overload/balanced-load, for example

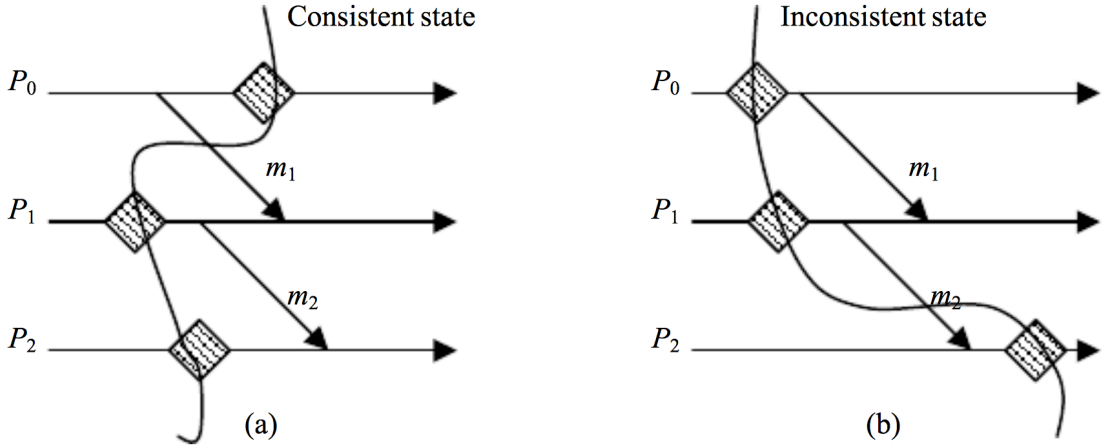
Problem: find a way to coordinate processes in a distributed system so that they all record their states **at approximately the same time.** Let's try to pin down what is needed for these snapshots to be "consistent."

- if a process's checkpoint reflects the receipt of a message, then the send of that message needs to also be in a checkpoint
  - more so: if a channel's checkpoint reflects a message, then the send of that message needs to also be in a checkpoint
  - basically causal consistency:
    - if a → b, and b is in some checkpoint, then so is a.

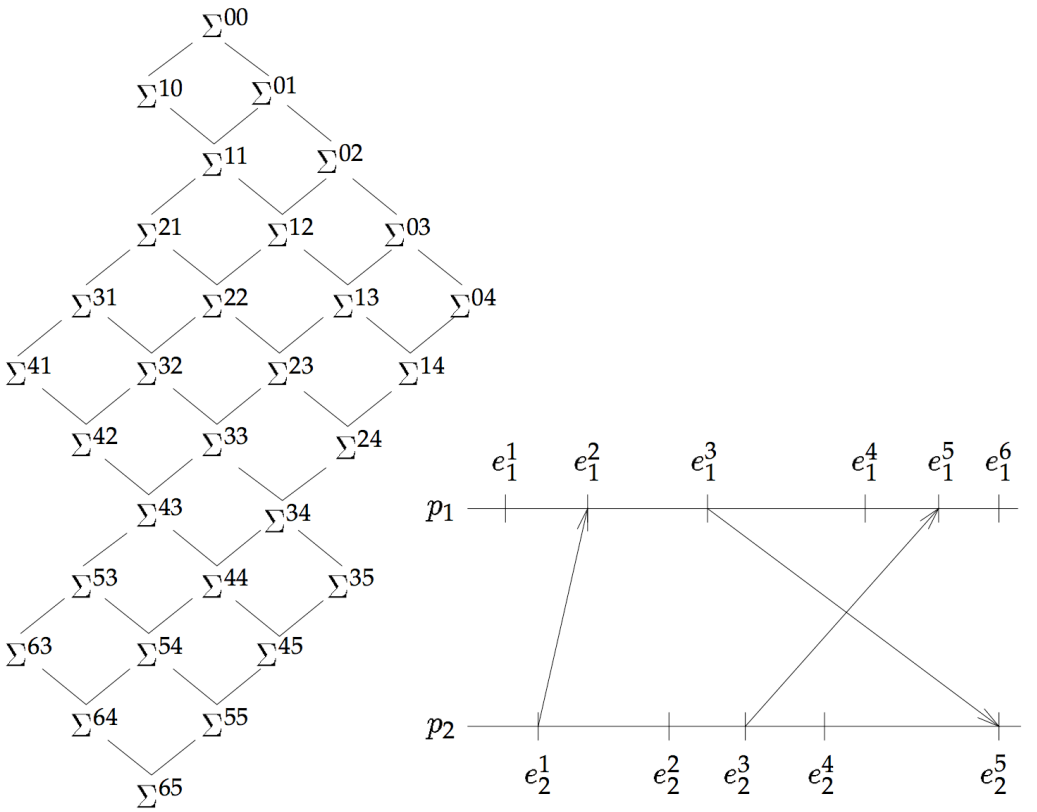
- if a process's checkpoint reflects the send of a message, then that message needs to be in either a channel checkpoint or the recipient's checkpoint
  - basic model assumes reliable communication, so can't afford to lose messages. could relax model to get rid of this requirement.

- turns out this is enough to give you the stable predicate property

One way to think about distributed checkpoints



Leads to a notion of a lattice of execution states



Go over, in particular why message send/receive constrain the lattice, and the notion of "leads to"

Consistent global snapshot

From the point of view of the lattice:

- snapshot algorithm begins with computation at lattice point So
- snapshot algorithm terminates with computation at lattice point Se
- the snapshot itself captures lattice point S*
- proof in the paper that:
    - So "leads to" S* "leads to" Se
    - even though actual computation could have taken a different path that doesn't go through S*

Alternatives to Chandy/Lamport

- freeze, dump, start
    - snapshotter sends a "suspend" message to all processes
    - processes send "flush" message on all outgoing channels
    - processes wait for "flush" on all incoming channels
    - processes take snapshot, send to snapshotter
    - snapshotter sends a "resume" message to all processes
- snapshot at wall-clock time X
    - doesn't deal with state in channels
    - doesn't necessarily preserve causality, if clock skew exists
    - if known clock sync bound, hybrid "freeze for epsilon then snapshot" so that causality is preserved; still problems with message channels

Chandy/Lamport algorithm

Assumptions:

- finite set of processes (why?)
- finite set of channels (why?)
- strongly connected directed graph between processes (why?)
- channels
    - infinite buffers [why? no deadlock]
    - error free [why? liveness and correctness]
    - in-order delivery [why? correctness]
    - finite but aribtrary delay [why? liveness]
- processes
    - deterministic state machine
    - correct and prompt

Start:

- some processor P0 sends itself "take a snapshot" token

Progress:

- the first time Pi receives a "take a snapshot" token (from Pj)
    - Pi records its local state
    - Pi sends "take a snapshot" along all outgoing channels
    - Pi records that channel from Pj is "empty"
    - Pi begins recording messages from all of its other channels
- from then on, when Pi receives "take a snapshot" from Pk
    - Pi stops recording the channel from Pk

Termination:

- when Pi has received "take a snapshot" on all incoming channels, it sends its recorded state to P0 and stops

Why does this work?

- tokens are basically an event horizon separating causal "now" from causal "after"
- if a Pi records state that includes receipt of a message from Pj
    - know that Pj records state that includes sending that message

Q: why not just drop all messages in communication channel, since TCP or application-level retransmit is good enough?

- Only OK if retransmit preserves ordering [TCP does, if you consider messages at the transport level. Application level may not.]
    - If not preserve order, not good enough to preserve stable properties. Ordering of messages in channel matters [e.g., imagine lock acquire/free requests are reversed in a deadlocked dependency graph].
- Carefully think where boundary between channel and process is. If drop messages, need to capture TCP/IP stack.

Q: can you extend global snapshot to broadcast or multicast communications channels? How?

Q: what happens if channels are not infinite capacity? (Process needs to block.) If so, can deadlock occur? Yes, if all processes are sending and all channels are blocked. Can fix this? Don't know, haven't figured it out.