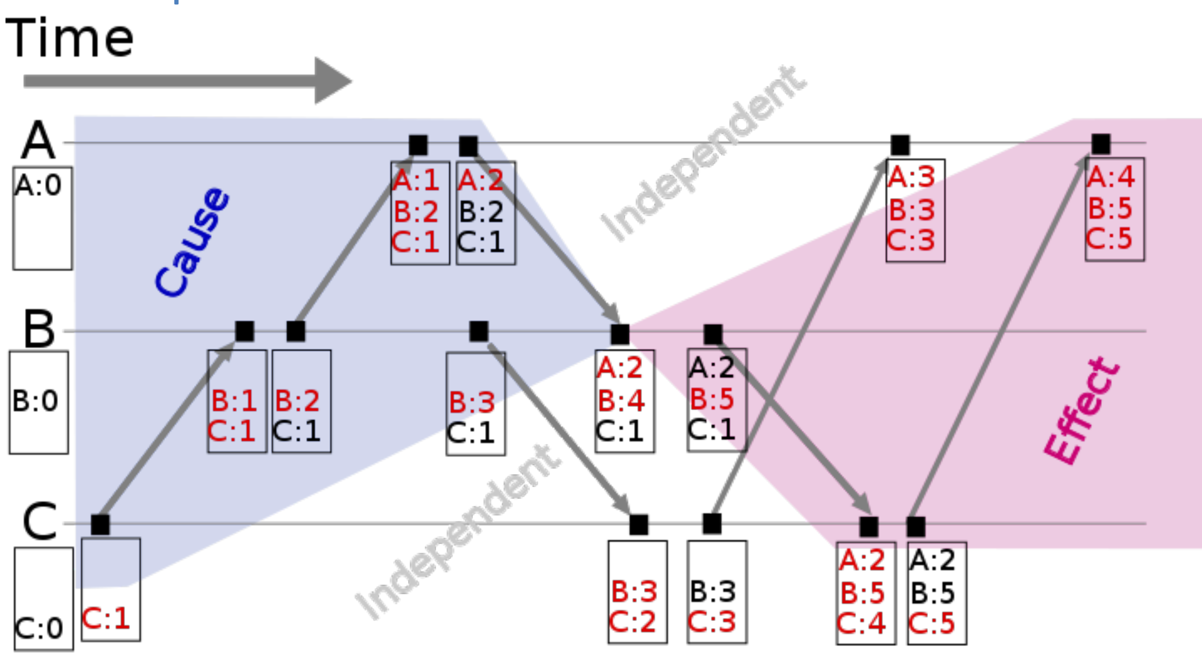
Lecture 8 – Consistent Distributed Snapshots

Vector clock example



1.

Replicated state machine: Using logical clocks:

a. Real problem: want a set of nodes to see same set of state transitions
    i. E.g. lock requests, acquires, releases.
b. Problem:
    i. Want to have a group of nodes perform the same set of actions on a set of messages
    ii. General approach: each node implements a state machine
        1. Has local state
        2. Receives messages causing it to update state, send reply message
        3. In some cases, must receive messages in same order at every node
        4. Or, states must be commutative (can receive out of order without changing outcome)
    iii. For example: a distribute service storing your bank balance
        1. Send messages to deposit/withdraw to multiple copies, want outcomes to be the same

iv. For example: decide who gets to modify a shared object (e.g. access shared storage)
   1. Send request to access to all nodes
   2. All nodes agree on an order of who gets to access next
   3. When it is your turn, do the access
   4. When done, send message to release access

c. How it works for mutual exclusion:
   i. Rules we want to implement:
      1. A process granted the resource must release it before anyone else can access it (safety)
      2. Grants of the resource are made in the order the requests are made
      3. If every grant is eventually release, then every request eventually granted (liveness)
   ii. What if we use a central scheduler? (assuming asynchronous messages)
      1. P0 has resource
      2. P1 sends a message to P1 requesting resource, then P2
      3. P2 receives P1's message, then sends a request to P0 asking for resource
      4. P0 receives P2's request before P1s (violation condition 2)
   iii. Assume:
      1. P0 starts with resource
      2. FIFO channels
      3. Eventual delivery (no failures)
   iv. Solution:
      1. Each process maintains a local **request queue** initialized to T0P0 (because P0 requests resource at time T0)
      2. To request the resource, process Pi sends a **RequestResource** message Tm:Pi to all other processes and places it in its own request queue
      3. When process Pj receives a request resource message, it places it in its request queue and sends a (timestamped) ack message back to Pi
      4. To release a resource, Pi remove the **RequestResource** message for Pi from its own queue and sends a **Tm:Pi Release Resource** message to all other processes (old Tm:Pi)
      5. When process Pj receives a release message, it removes Tm:Pi, it removes any Tm:Pi request resource message from its queue
         a. Note: this must be after the request and after the ack

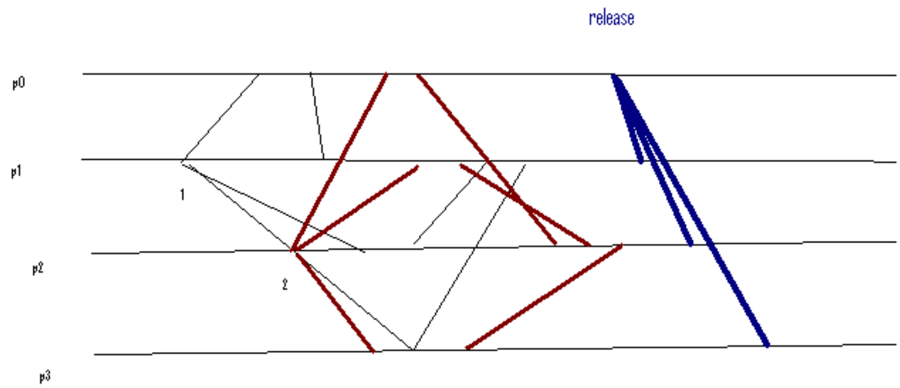
6. Process Pi is granted the resource when:
   a. There is a Tm:Pi **RequestResource** message in its queue when Tm < any other Tm (assuming a total order for messages)
   b. Pi has received a message from every other process with a time > Tm

v. Why works?
   1. Condition b in part 6 above (Pi has received messages) ensures that Pi would have heard about any other request from any other process with a timestamp < Tm
   2. Messages not deleted until granter sends a release message, so it will be in everyone's queue
   3. Overall, don't take resource until everyone else ACKs and you know you are the least. On release resource, as soon as you get a release, you can go next, because you know everybody else agrees you will go next

vi. QUESTION: What happens if there is a failure (message lost, time out etc)?
   1. Need to retry on a link-to-link basis

vii. NOTE: relies on common knowledge
   1. When you get the acks from everyone else, a process has common knowledge that everyone knows of its request, and they know that Pi knows of their requests when they see the ack

viii. Example:
   1. For processes: P0, P1, P2, P3
   2. P1, P2 send "request messages", P1 at local time 1, P2 at local time 2
   3. P0-P3 put P1:1 and P2:2 in their queue and ack
   4. P0 sends release message
   5. P1 takes over. When done, sends release
   6. P2 takes over



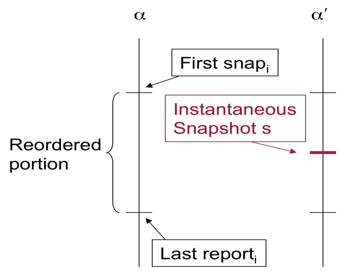
7.

2. Benefits of state machine approach
    a. **Everybody decides on right thing to do locally, knows everybody else will make the same decision (common knowledge)**
    b. If everybody has the same initial state (e.g. lock release at low time) and sees the same sequence of messages in the same order, they will compute the same result in a distributed fashion
        i. Basis for lots of mechanisms – replication
    c. Note: Given protocol pretty unrealistic – it really is an example of how it could work
    d. But basics of protocol are used – e.g. chubby lock servers use similar replicated state machines

Distributed Snapshots

3. Questions from Reviews
    a. N squared complexity?
4. Context
    a. Last lecture: talked about how global time wasn't that meaningful, couldn't talk about what happens at one particular time.
    b. Now: what if you want to know the state of a system? How do you know the state
    c. Problem:
        i. State of system =
            1. State of processes +
            2. State of network (channels
        ii. Cannot capture all simultaneously (no global time with this accuracy)
        iii. QUESTION: How many network channels are there?
            1. What does this imply about the number of messages you need?
    d. Need to tell each process what to record and when
    e. Need to record contents of channels properly
        i. Cannot ignore channels or deliver all messages

        ii. Delivery a message can trigger more sends, which would have to be delivered, which …
    f. Cannot pause entire system
        i. This makes it too easy, or causes too much performance loss
    g. Would like to be able to test properties of the state
        i. We'll call them "stable properties" – once true, are always true.
5. When are snapshots useful?
    a. Deadlock detection: is there a circular waits-for graph?
    b. Debugging: has an invariant been violated
        i. E.g. sum of the tokens in a system = n
    c. Checkpoint: can save state and resume later
    d. QUESTION: What if the state you want to check is not stable – it can vary over time
        i. Is there anyway to snaphot in an asynchronous system that will capture it?
        ii. Do you need consistency in that sense?
        iii. So you see the property is true/false at an instant in time – then what?
            1. Is this meaningful?
6. Assumptions
    a. Fifo channels
    b. Processes form a strongly connected graph (path from every node to every other node)
    c. Messages delivered in finite time
        i. QUESTION: Why? Needed for liveness to algorithm finishes
    d. No outside world
        i. So can capture complete state
7. What kinds of snapshots are there?
    a. "instantaneous snaphot" – global state of everything at some point (real world time)
        i. But cannot do – each process can only see local state
        ii. Have random network delays preventing tight synchronization
        iii. QUESTION: What is it good for?
            1. Loads on system, transient effects like delays
    b. "Consistent snapshot" – looks like an instantaneous snapshot (could have happened legally), but not at one time
        i. Good enough in some cases
        ii. Is same as real snapshot up to start of snapshot, and after termination of snapshot
        iii. Snapshot is state at some point in of a legitimate execution during the snapshot (but may not have actually occurred)

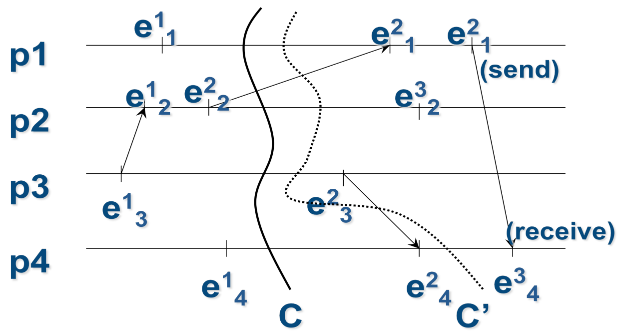
iv. 

c. What are snapshots used for?
   i. Stable properties: if property P of a global state S becomes true, it is true for all states reachable from S
   ii. E.g.: deadlock
   iii. E.g. termination of a distributed algorithm (all processes waiting for another process to send a message to work on)

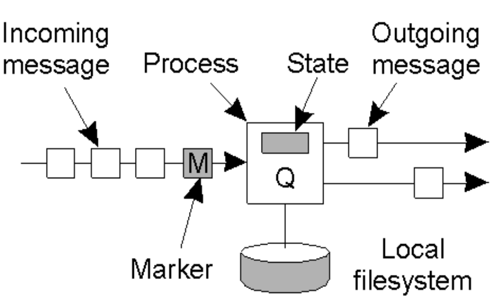
8. Models/definitions:
   a. "causally consistent global state" – no even in state caused by something not in state
      i. cannot have receipt without send being captured
      ii. Cannot have event j captured in a process without event k, k < j
   b. System model:
      i. Local state = each process
         1. Processes move between states (s -> s') on events
         2. Events are sending message, receiving message, internal event
         3. Receiving pops message off queue, send pushes message on queue
         4. Events advance state of process Si to Si+1
      ii. Global state advances on event in one process at a time
         1. Event e = (p,s,s',c,m) = processes p was in state s and is now in state s' having sent message m on channel c (outgoing c) or received message m on channel c (incoming c)
         2. Can execute an event if a process p is in state s and has a message m at the **head of the queue** for channel c (or message M, channel c are NULL)
         3. Can have nondeterminism: multiple next events could happen
            a. One of two processes can go next
            b. Process can do internal event or receive a message
         4. BUT: sequence has a total order (unlike Lamport clock model)
   c. How does this relate to other models?
      i. COMPARE to Lamport partial order
         1. Instead has total order of global states
      ii. Assumes reliable network, fifo delivery (unlike Lamport clocks)

9. Terminology
   a. CUT = line through each process separating each one into a PAST and a FUTURE
   b. CONSISTENT CUT = line such that
      i. No future messages received in past

ii. Preserves causal order: future can not have causal effect on past
iii. SHOW EXAMPLE OF CONSISTENT AND INCONSISTENT CUT from below – C and C'

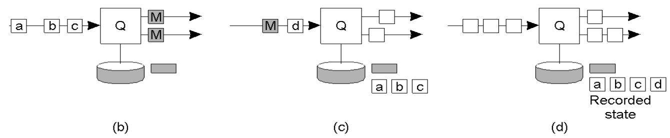
10. How do you snapshot?
   a. Given space-time diagram (event e in C, everything after event e is also in C)
      Finding C such that $(e \in C) \wedge (e' \rightarrow e) \Rightarrow e' \in C$
   b. 

   c. Key idea: nodes take snapshots, record incoming messages as channel state
      i. Use markers to indicate beginning/end of snapshot process
   d. PROBLEMS TO SOLVE:
      i. When should a process save its state?
      ii. What messages should it store as channel state?
         1. Any message sent before snapshot must be recorded either in process state (as received) or channel state (as in flight)
         2. Any message sent after snapshot must not be recorded in either way
   e. Algorithm:
      i. General model: a diffusion algorithm
         1. Send message out to all nodes (like flooding) until everybody has received it
      ii. When uninvolved process i receives $snap_i$ input:
         1. Snaps A_i's state.
         2. Sends marker on each outgoing channel, thus marking the boundary between messages sent before and after the $snap_i$.
         3. Thereafter, records all messages arriving on each incoming channel, up to the marker.



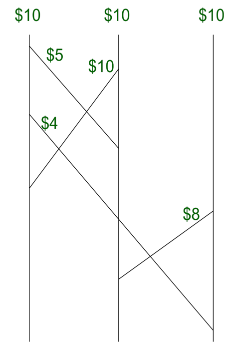
(a)

         4.

    iii. When process i receives marker message without having received $snap_i$:
        1. Snaps A_i's state, sends out markers, and begins recording messages as before.
        2. Channel on which it got the marker is recorded as empty.
        3. 

    iv. So:
        1. Initiator saves its state, then saves messages received along each channel until it receives a marker back
            a. Ensures messages sent after one node snaps but before other are captured as channel state
        2. When receive a marker, don't need to record anything on that channel, but must record other channels until get a marker back.
    v. QUESTION: what if a process delays between snapping and sending markers?
f. Terminates:
    i. Strongly connected, so will eventually reach all nodes, and will receive marker along all channels
    ii. Finite delivery time ensures finite termination for finite network
g. QUESTION: How do you use the snapshot state to detect a stable property?
    i. E.g. deadlock
        1. QUESTION: What is state?
            a. Look at Lamport locks
            b. Queue of messages at each node
            c. Internal state of who holds each lock
        2. QUESTION: What is channel state
            a. Message to request/release/ack
        3. HOW DO YOU DETECT DEADLOCK
            a. Circular graph of nodes holding locks and requests for other locks.
    ii. E.g. total money in a bank system – see below
        1. Add up money in each process + money in channels
h. Why it works:
    i. No message sent after maker on a channel will be recorded; marker makes the cut
    ii. When a process receives a message that precedes the marker:
        1. If it has not taken the snapshot, the message is processed and is part of its state
        2. If it has taken a snapshot, then the message is recorded as being inflight and part of channel state (the cut crosses the send/receive of the message)
i. Example:

- Distributed bank, money sent in reliable messages.
- Audit problem:
  - Count the total money in the bank.
  - While money continues to flow around.
  - Assume total amount of money is conserved (no deposits or withdrawals).



j.

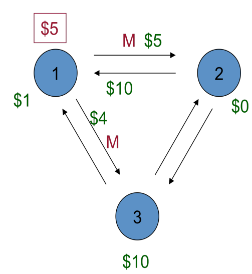
- Nodes 1,2,3 start with $10 apiece.

- Node 1 sends $5 to node 2.
- Node 2 sends $10 to node 1.
- Node 1 sends $4 to node 3.
- Node 2 receives $5 from node 1.
- Node 1 receives $10 from node 2.
- Node 3 sends $8 to node 2.
- Node 2 receives $8 from node 3.
- Node 3 receives $4 from node 1.

k. • Count the money?

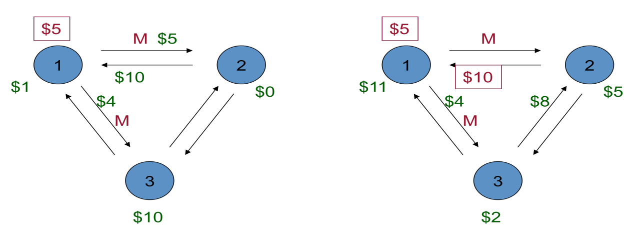
l. Assume snap input after node 1 sends $5 to node 2

- Node 1 sends $5 to node 2.
- Node 2 sends $10 to node 1.
- Node 1 receives snap input, takes a snapshot, records state of A_1 as $5, sends markers.
- Node 1 sends $4 to node 3.



m.

- Node 2 receives $5 from node 1.
- Node 1 receives $10 from node 2, accumulates it in its count for $C_{2,1}$.
- Node 3 sends $8 to node 2.

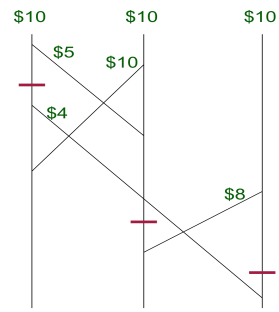


n.

o. If just snapshot node state without channels:

- Nodes 1,2,3 start with $10 apiece.

- Node 1 sends $5 to node 2.
- Node 2 sends $10 to node 1.
- Node 1 snaps.
- Node 1 sends $4 to node 3.
- Node 2 receives $5 from node 1.
- Node 1 receives $10 from node 2.
- Node 3 sends $8 to node 2.
- Node 2 snaps.
- Node 2 receives $8 from node 3.
- Node 3 snaps.
- Node 3 receives $4 from node 1.



p.
    i. NOTE: money recorded is $5 at node 1, $5 at node 2, and $2 at node 3
    ii. NOTE: Missing channel state: $18 dollars

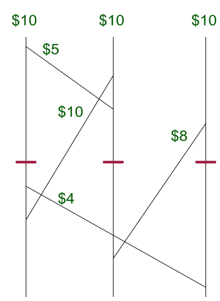
q. Look at what was recorded: with Chandy-Lamport protocol:
    i. Node 1 sends marker to nodes 2 and 3, arrives at snapshot times
    ii. Node 2 sends to node 1,3
    iii. Node 3 sends to node 1,2
    iv. Node 1 records channel state of $10 from node 2 (between snap and marker) node 2 records channel state of $8 from node 3 (between snap and marker from node 3)

- Nodes 1,2,3 start with $10 apiece.

- Node 1 sends $5 to node 2.
- Node 2 sends $10 to node 1.
- Node 2 receives $5 from node 1.
- Node 3 sends $8 to node 2.

- Everyone snaps.

- Node 1 sends $4 to node 3.
- Node 1 receives $10 from node 2.
- Node 2 receives $8 from node 3.
- Node 3 receives $4 from node 1.



r.

s. Why is this reordering correct?
    i. Problem: process could change state asynchronously (internal events) before the markers it sends are received by other sites
    ii. Has same events, can get from to this state with same events (in different order) from input
    iii. Can get from this state to same output event with same events (in different order)
    iv. Key idea:
        1. Reorder events in total order so that all pre-snapshot events happen, then snapshot, then post-snapshot events
    v. Notion:
        1. Actual states = global states that occurred
        2. Feasible states = states that could occur according to local state machine at each process
    vi. Based on logical time: can reorder logically concurrent events in the total order and get an equivalent output
    vii. Suppose we could not reorder:

1. Means there is a "happens before" relationship between the things being reordered
2. Implies either
   a. They are in the same process -> but not reordering anything in a single process
   b. There is a line of causal communication between them
3. If causal communication, then must have been a message
   a. Would have an earlier (but post-snapshot) event followed by a later (but pre-snapshot) event with communication
   b. But by rule, always send marker after snapshot, so recipient (pre-snapshot) would have had to snapshot,
   c. CONTRADICTION!

t. Effectively picks a "virtual time" for snapshot, moves all events to be before or after that event by stretching/compressing timelines

11. Unreliable networks
    a. What if the network is unreliable?
    b. ANSWER: use a protocol to make it reliable, like TCP/IP.
       i. This guarantees that if marker is received, all messages before it will be received
       ii.
12. Using snapshots
    a. Still useful today?
       i. We have synchronized clocks, but networks are much faster.
          1. In 1 ms of skew, could have 1-10 megabits (100k-1mb data)
    b. Use in bank balance:
       i. Can detect invariants (is the amount of money constant)
          1. Sum balances + in-flight transfers
          2. Only one node should hold a lock at a time
       ii. Can detect deadlock
          1. See what each process is waiting for
          2. Look at what "wake up" message have been sent
          3. If circular waiting and no wake-up message after waiting, then will deadlock
    c. What about non-stable properties?
       i. Can detect them, but may be false positives (as would be true perhaps in any system), as they could go away
13. FLAWS:
    a. State external to the system not captured (e.g. clients of a distributed service)