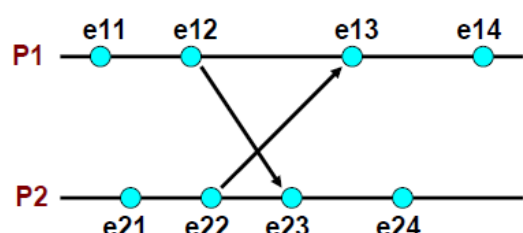


## Lecture 6: Logical Time

1. Question from reviews
  - a.
2. Key problem: how do you keep track of the order of events.
  - a. Examples: did a file get deleted before or after I ran that program?
  - b. Did this computers crash after I sent it a message?
  - c. QUESTION: Why is this a problem?
    - i. Clocks may be different on different machines
      1. E.g. processors in a multiprocessor system
      2. Machines in a cluster
    - ii. QUESTION: How different do they have to be?
      1. More than the minimum time to send a message (1 ms), which is not much
    - iii. Relativity: given different computers executing simultaneously and sending messages asynchronously, how can you tell?
  - d. QUESTION: what do we really care about?
    - i. If one thing happened at time X, and another at time X+delta, and they never communicate, does it matter?
    - ii. Focus on “happens before” relationship
    - iii. Don’t need real clocks for many uses; since we are more interested in the **order of events** then in when the actually happened
  - e. Examples:
    - i. What kind of clock is good for security logs?
      1. Wall clock – want to correlate with human-scale events
      2. Absolute time – coordinate with outside world
    - ii. What kind of clock is good for figuring out which machines communicated and when?
      1. Logical clock: want to be able to order the communication from different machines (relative order)
  - f. QUESTION: Is there an application to computer games?
    - i. E.g. in a distributed environment, you can tell where another player is logically?
3. CONTEXT FOR SOLUTION
  - a. General approach of theoretical papers: strip out all practical concerns not relevant to the problem, as they can be layered on afterwards if you get the basics right

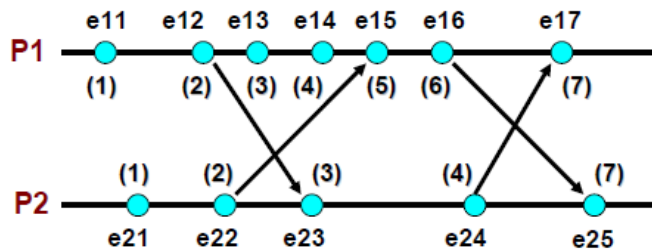
- b. Example: ignore message loss, reordering on a link
      - i. Easy to solve with TCP/IP
    - c. Example: Ignore process/link failure
      - i. Hard to solve, but need a separate protocol and this system works fine between times
    - d. QUESTION: Why?
      - i. Addressing all these concerns is orthogonal to the problem in many cases, clutters paper
      - ii. Note: real clocks and message delay are relevant, so they are included
  - 4. Happens before
    - a. Intuitive idea:
      - i. Events in a single process are ordered (they are sequential)
      - ii. A message send always precedes the receipt of that message (no speculation!)
    - b. For two events  $a, b$ ,  $a$  happens before  $b$  ( $a \rightarrow b$ ) if:
      - i.  $a$  and  $b$  are events in the same process and  $a$  occurred before  $b$ , or
      - ii.  $a$  is a send event of a message  $m$  and  $b$  is the corresponding receive event at the destination process, or
      - iii.  $a \rightarrow c$  and  $c \rightarrow b$  for some event  $c$  (transitive)
    - c. Indicates causal relationship;  $a$  can affect  $b$
  - 5. Concurrent events:
    - a. Not  $a \rightarrow b$  and not  $b \rightarrow a$
- 

$e_{11}$  and  $e_{21}$  are concurrent

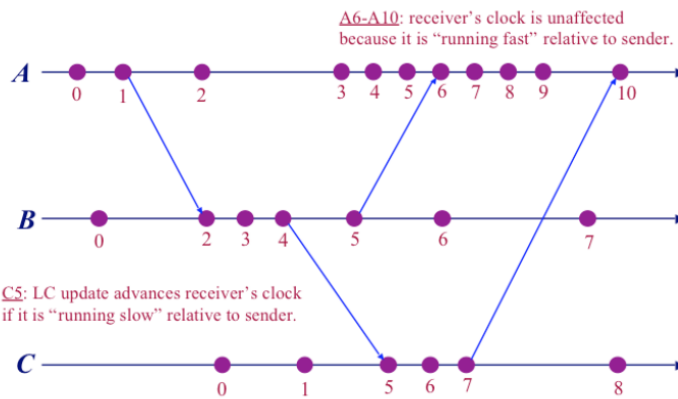
$e_{14}$  and  $e_{23}$  are concurrent

$e_{22}$  causally affects  $e_{14}$
- b.
    - c. Space time diagrams: time moves left, space is vertical (rotated from paper)
    - d. Note: this is a partial order
      - i. Not all events are ordered, some are before others (or after), but some are not.
      - ii. QUESTION: in a distributed system, do you need a complete order or a partial order?
  - 6. Logical clock: any counter that assigns times to events such that
    - a. Clock condition:  $A \rightarrow B$  implies  $C(a) < C(b)$
  - 7. Lamport Logical Clocks
    - a. Each process  $P_i$  maintains a register (counter)  $C$

- b. Each event  $a$  in  $P_i$  is timestamped  $C_i(a)$ , the value of  $C$  when  $a$  occurred
- c. IR1:  $C_i$  is incremented by 1 for each event in  $P_i$
- d. IR2: If  $a$  is the send of a message  $m$  from process  $P_i$  to  $P_j$ , then on receive of  $m$ :
  - i.  $C_j = \max(C_j, C_i(a)+1)$



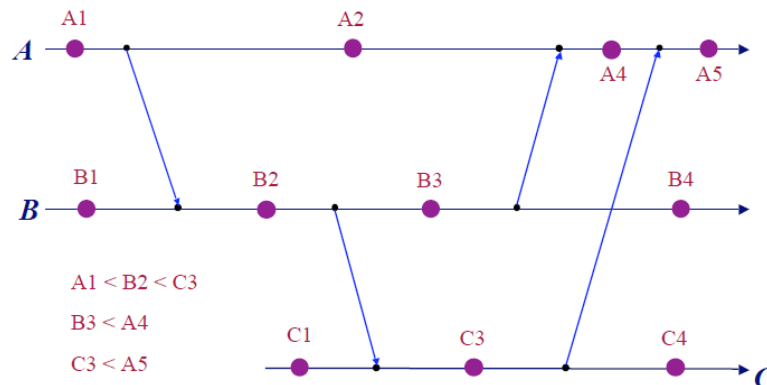
e.



f.

DRAW TICK LINES

(connect zeroes, 1s, etc)



g.

8. Notes on logical clocks:

- a. It provides the guarantee that  $a \rightarrow b$  implies  $C(a) < C(b)$
- b. But,  $C(a) < C(b)$  does not imply  $a \rightarrow b$ : see events  $e_{24}$  and  $e_{15}$  above
- c.  $C(a) == C(b)$  implies  $a$  and  $b$  are concurrent, but not vice versa (see  $e_{24}$ ,  $e_{14}$ )

9. IN LOOKING AT TICK LINES:

- a. Must be line between two concurrent events
- b. Must be line between send and receipt of a message

10. QUESTION: What happens with failures? How does that affect ordering

11. Total order

- a. What if you need to agree on a total order for events?
- b. Use logical clocks and break ties deterministically: using process ID or node ID as a tie breaker
- c. QUESTION: is this really a total order?
  - i. Real thing: an agreed upon order consistent with reality for happens-before
- d. QUESTION: What happens with failures?

12. Use of logical clocks

- a. Suppose everybody broadcasts updates
- b. How do you impose a fixed order on updates?
- c. Do them in logical time order (assuming you wait forever...)

13. BIG QUESTION:

- a. How useful is this?
  - i. When you care about order?
  - ii. When you don't have synchronized time
    - 1. Sensors
    - 2. Loosely coupled machines
  - iii. When you cannot afford a common time base
    - 1. Multiprocessors

14. Physical clock extensions

- a. Similar rule, but advance time according to clock received + minimum possible delay
- b. Need clock to be monotonic increasing
- c. Is the basis for NTP – send multiple messages to learn the minimum delay in each direction, use that to sync clocks to bounds tighter than delay

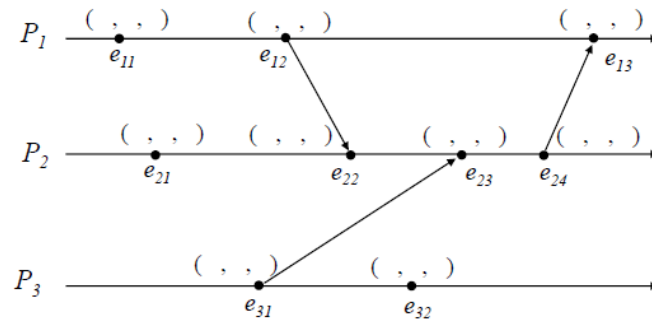
15. Vector clocks (also used as Version Vectors)

- a. Extension of logical clocks to capture more information
- b. Suppose A sends to B, D at time 2 (A changes object, sends it out)
  - i. Time of B is 3
  - ii. Time of D is 3
  - iii. D then sends to B
    - 1. At B: has D seen A's message yet? Does the copy of the object from D include A's change?
    - 2. Cannot answer with logical clocks
      - a.  $C(D \text{ send}) > C(A \text{ send})$  does not imply D send logically occurs after A sends

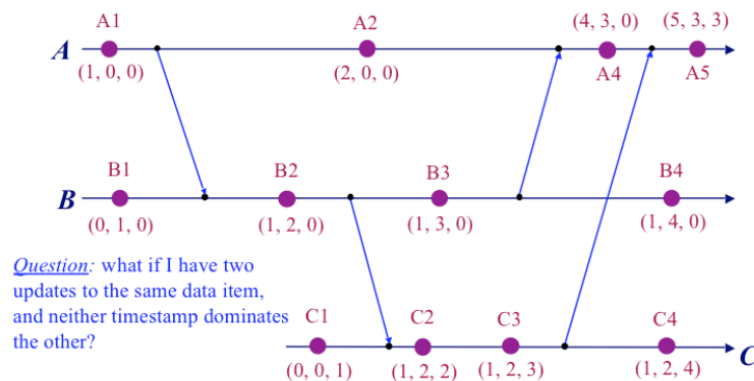
c. Solution: “vector clocks”

- i. Keep one logical clock **per process**, only incremented with local events
- ii. Maintain a local **vector clock** tracking received timestamps
- iii. Transmit all logical clock values you have seen
- iv. Set local vector clock to pairwise max(received vector, local vector)
- v. So:
  1.  $C_i[i]$  =  $P_i$ 's own logical clock
  2.  $C_i[j]$  =  $P_i$ 's best guess of logical time at  $P_j$ 
    - a. Or: latest thing that  $P_j$  did that  $P_i$  knows about directly or indirectly
- vi. Implementation rules:
  1. Events A and B in the same process:  $C_i[i]$  for a =  $C_i[i]$  for b + delta
  2. Send vector clock  $T_m$  on all messages  $M$
  3. If A is sending and B is receiving of a message  $M$  from  $P_i$  to  $P_j$ :
    - a. For all  $K$ ,  $C_j[k] = \max(C_j[k], T_m[k])$

vii. Example:



viii.



ix.

d. Rules for comparison:

- Vector timestamps can be compared in the obvious way:

- $t^a = t^b$  iff  $\forall i, t^a[i] = t^b[i]$
- $t^a \neq t^b$  iff  $\exists i, t^a[i] \neq t^b[i]$
- $t^a \leq t^b$  iff  $\forall i, t^a[i] \leq t^b[i]$
- $t^a < t^b$  iff  $(t^a \leq t^b \wedge t^a \neq t^b)$

- Important observation:

- $\forall i, \forall j : C_i[i] \geq C_j[i]$

i.

ii. So:

1. Equal if all elements equal
2. Not equal if at least one element not equal
3.  $Ta \leq Tb$  if all elements less or equal
4.  $Ta < Tb$  if  $Ta \leq Tb$  and  $Ta \neq Tb$ 
  - a. Means must be at least one element where  $Ta[k] < Tb[k]$

iii. Causally related events with vector clocks:

1.  $A \rightarrow B$  if and only if  $Ta < Tb$

iv. Concurrent with vector clocks:

1.  $Ta \not\leq Tb$  and  $Tb \not\leq Ta$
2. Consider past example: (A changes an object, sends it out)
  - a. Suppose A sends to B, D at time 2
    - i. Time of B is 3
    - ii. Time of D is 3
    - iii. D then sends to B
      1. At B: has D seen A's message yet?
      2. Cannot answer with logical clocks
        - a.  $C(D \text{ send}) > C(A \text{ send})$  does not imply D send logically occurs after A sends
  - b. A (1,0,0) sends to B and D
  - c. B receives at (0,3,0), sets clock to (1,3,0)
  - d. D receives at (0,0,2), sets clock to (1,0,3)
  - e. D sends to B at (1,0,4)
  - f. B receives when clock is (1,4,0)
    - i. B knows that D has received A's message, because it has a 1 for A's clock

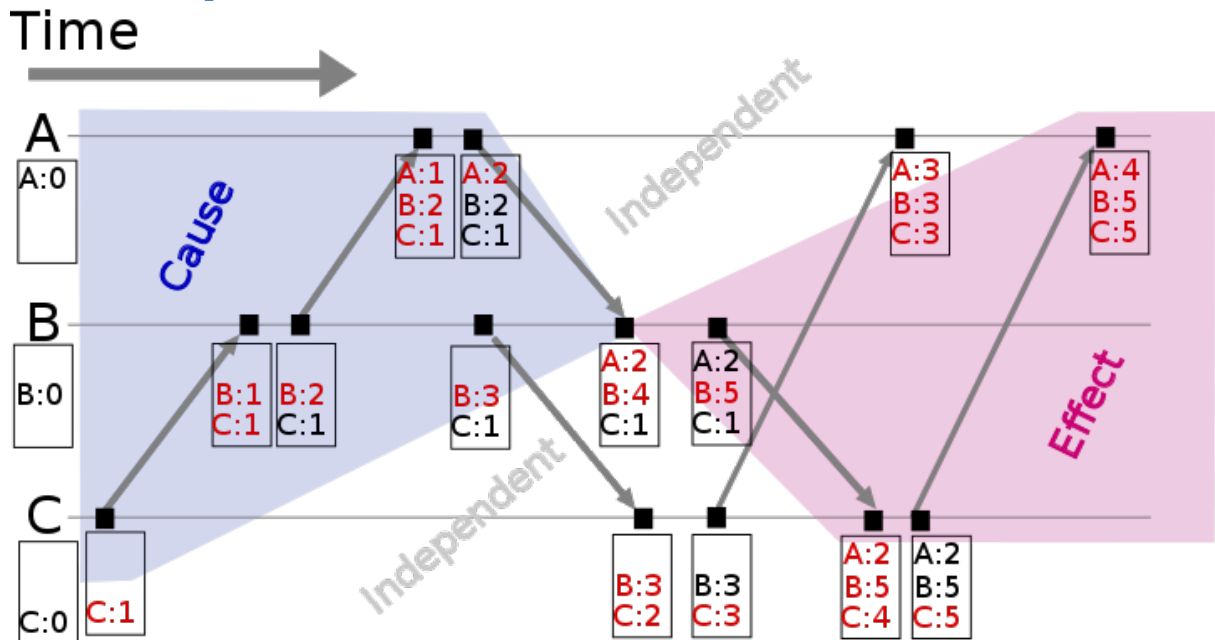
e. Issues with vector clocks

i. How big are vectors?

1. Same size as the number of machines

- ii. What if the set of machines changes? Can you get rid of elements
  1. Only if you are sure it will never come back
- iii. When used?
  1. Good for replication (multiple copies of an object)
    - a. Can modify at multiple points
    - b. Can exchange updates pairwise
    - c. Want to know if the other side saw an update you saw

### Vector clock example



1.

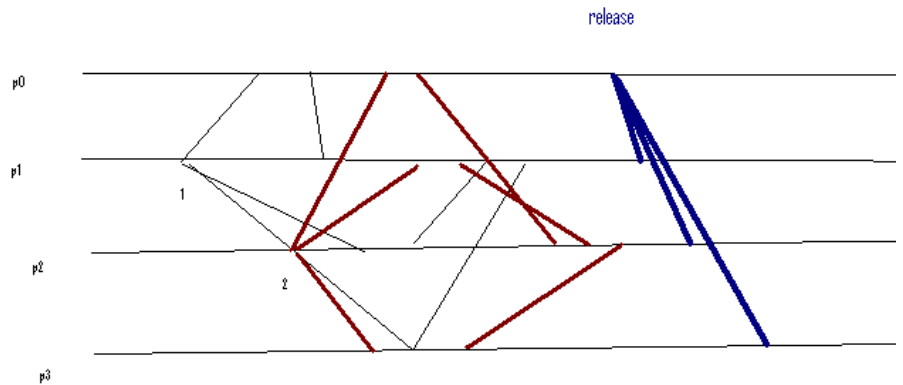
### Replicated state machine: Using logical clocks:

- 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  $P_j$  receives a release message, it removes  $T_m:Pi$ , it removes any  $T_m:Pi$  request resource message from its queue
    - a. Note: this must be after the request and after the ack
  6. Process  $P_i$  is granted the resource when:
    - a. There is a  $T_m:Pi$  **RequestResource** message in its queue when  $T_m <$  any other  $T_m$  (assuming a total order for messages)
    - b.  $P_i$  has received a message from every other process with a time  $> T_m$
- v. Why works?
1. Condition b in part 6 above ( $P_i$  has received messages) ensures that  $P_i$  would have heard about any other request from any other process with a timestamp  $< T_m$
  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  $P_i$  knows of their requests when they see the ack
- viii. Example:
1. For processes:  $P_0, P_1, P_2, P_3$
  2.  $P_1, P_2$  send "request messages",  $P_1$  at local time 1,  $P_2$  at local time 2
  3.  $P_0-P_3$  put  $P_1:1$  and  $P_2:2$  in their queue and ack
  4.  $P_0$  sends release message
  5.  $P_1$  takes over. When done, sends release
  6.  $P_2$  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

## 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 snapshot 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 snapshot" – 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)



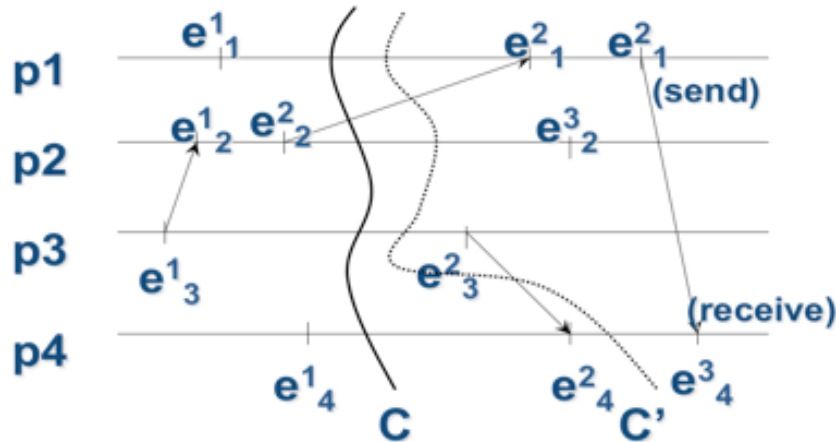
- iv.
- c. What are snapshots used for?
  - i. Stable properties: if property P of a global state S becomes true, it is true for all states reachable from S
  - ii. E.g.: deadlock
  - iii. E.g. termination of a distributed algorithm (all processes waiting for another process to send a message to work 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 ( $s \rightarrow s'$ ) 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  $S_i$  to  $S_{i+1}$
    - 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'

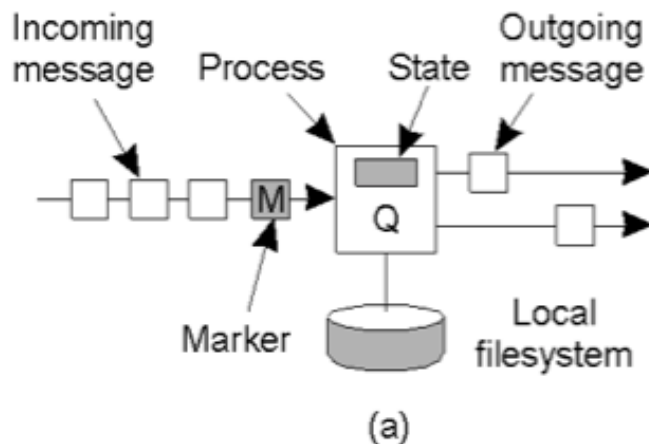
10. How do you snapshot?

- a. Given space-time diagram (event  $e$  in  $C$ , everything after event  $e$  is also in  $C$ )

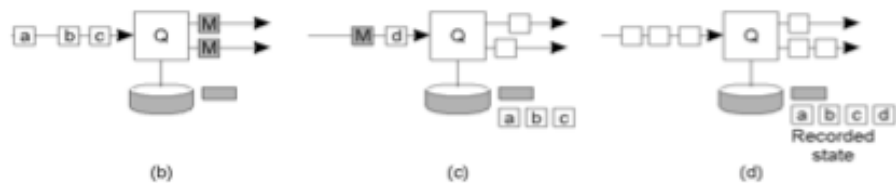
Finding  $C$  such that  $(e \in C) \wedge (e' \rightarrow e) \Rightarrow e' \in C$



- 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  $\text{snap}_i$  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  $\text{snap}_i$ .
    - 3. Thereafter, records all messages arriving on each incoming channel, up to the marker.



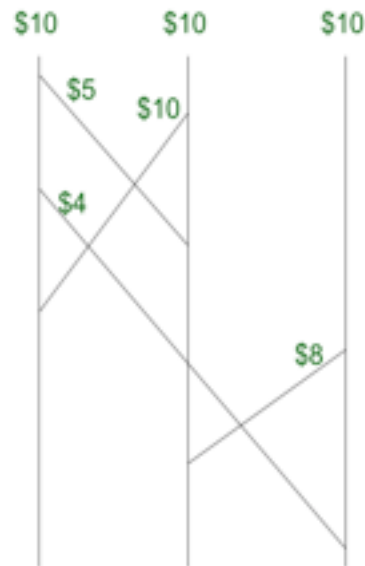
- 4.
- iii. When process  $i$  receives marker message without having received  $\text{snap}_i$ :
1. Snaps  $A_i$ 's state, sends out markers, and begins recording messages as before.
  2. Channel on which it got the marker is recorded as empty.



- 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

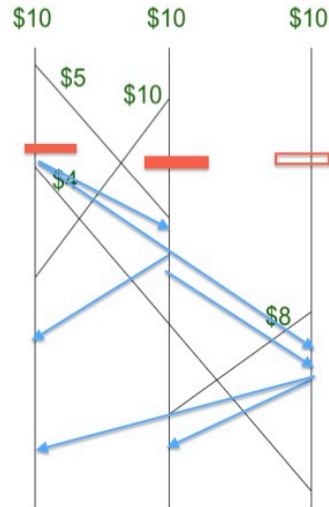
- a. Message to request/release/ack
  - 3. HOW DO YOU DETECT DEADLOCK
    - a. Circular graph of nodes holding locks and requests for other locks.
  - ii. E.g. total money in a bank system – see below
    - 1. Add up money in each process + money in channels
- h. Why it works:
  - i. No message sent after marker 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)
  - iii. Proof that it is a legitimate state between two global states
    - 1. Can swap concurrent events in the real sequence to get to the recorded state and from the recorded state to a real state
    - 2. Swapping order has no impact because they are concurrent
    - 3. Swap prerecording events with post recording events
      - a. cannot be on same node or with communication between nodes or
- 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.
- k. In picture below, start snap at first bar:
  - i. Node 1 has \$5
  - ii. Node 2 has \$0
  - iii. Node 3 has \$10
  - iv. Channel 2->1 has \$10
  - v. Channel 1->2 has \$5

- 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).



- l.
- m. In Chandy-Lamport snapshot:
  - i. Node 1 records \$5
  - ii. Node 2 records \$5
  - iii. Node 3 records \$2
  - iv. Node 1 records 2->1: \$10
  - v. Node 2 records 3->2 \$8
- n. 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. EXAMPLE:
    1. Real order:
      - a. 1 sends 2 \$5 - PRE
      - b. 2 sends 1 \$5 - PRE
      - c. 1 sends 3 \$4 - POST
      - d. 2 receives \$5 from 1 - PRE
      - e. 1 receives \$10 from 2 - POST
      - f. 3 sends \$8 to 2 - PRE
      - g. 2 receives \$8 from 3 - POST



- h. 3 receives \$4 from 1 - POST
  - 2. So can reorder
    - a. Move up d, f – could happen at any time
    - b. REDRAW!
- viii. 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!
- o. Effectively picks a "virtual time" for snapshot, moves all events to be before or after that event by stretching/compressing timelines
  - i.

#### 11. FLAWS:

- a. State external to the system not captured (e.g. clients of a distributed service)

#### 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)