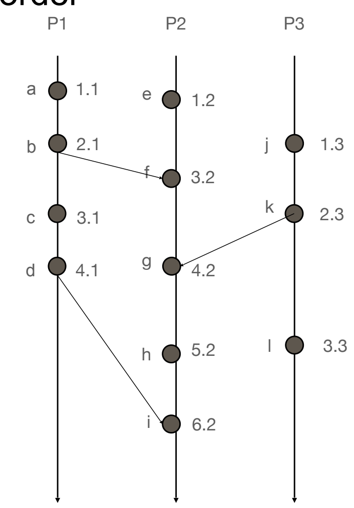
More Classical Problems

- Part 1: Vector Clocks
- Part 2: Distributed Snapshots
- Part 3: Termination Detection
- Part 4: Leader Election

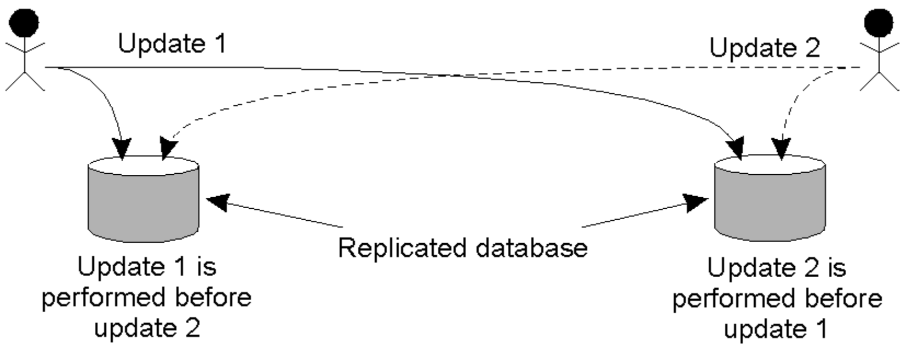
Total Order

- Create total order by attaching process number to an event. If time stamps match, use process # to order



Example: Totally-Ordered Multicasting

• Updating a replicated database and leaving it in an inconsistent state.



Algorithm

- Totally ordered multicasting for banking example
  - Update is timestamped with sender's logical time
  - Update message is multicast (including to sender)
  - When message is received
    - It is put into local queue
    - Ordered according to timestamp,
    - Multicast acknowledgement
  - Message is delivered
    - It is at the head of the queue
    - IT has been acknowledged by all processes
    - P_i sends ACK to P_j if
      - P_i has not made a request
      - P_i update has been processed and P_i's ID > P_j's Id

Causality

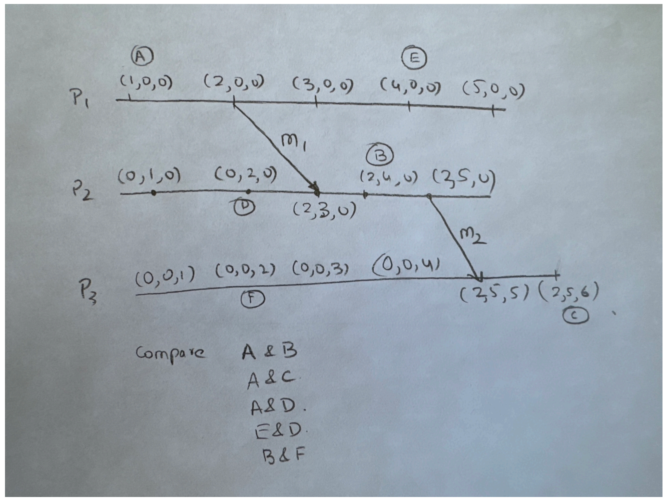
- Lamport's logical clocks
  - If *A -> B* then *C(A) < C(B)*
  - Reverse is not true!!
    - Nothing can be said about events by comparing time-stamps!
    - If *C(A) < C(B),* then ??
- Need to maintain *causality*
  - If a -> b then a is casually related to b
  - *Causal delivery*:If send(m) -> send(n) => deliver(m) -> deliver(n)
  - Capture causal relationships between groups of processes
  - Need a time-stamping mechanism such that:
    - If *T(A) < T(B)* then *A* should have causally preceded *B*

Vector Clocks

- Each process *i* maintains a vector V_i
  - $V_i[i]$: number of events that have occurred at I
  - $V_i[j]$: number of events I knows have occurred at process j
- Update vector clocks as follows
  - Local event: increment V_i[I]
  - Send a message :piggyback entire vector V
  - Receipt of a message: $V_j[k]$ = max($V_j[k], V_i[k]$)
    - Receiver is told about how many events the sender knows occurred at another process *k*
    - Also $V_j[j] = V_j[j]+1$
- *Exercise:* prove that if *V(A)<V(B),* then *A* causally precedes *B* and the other way around.

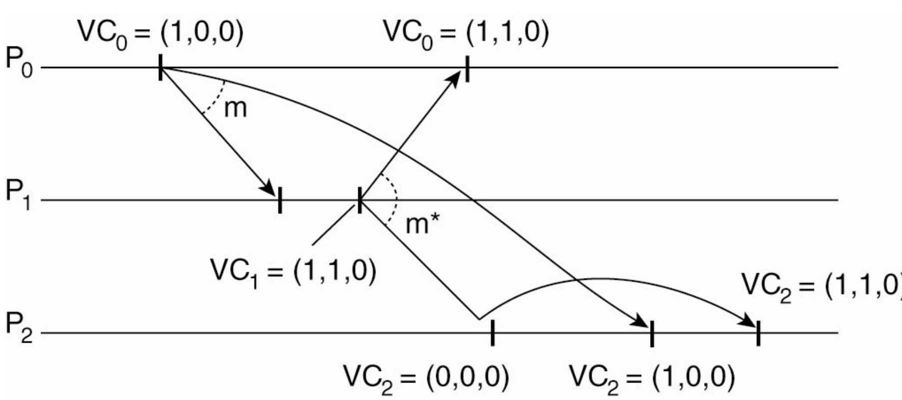
Vector Clock Example

- Vector clocks for three processes



Enforcing Causal Communication

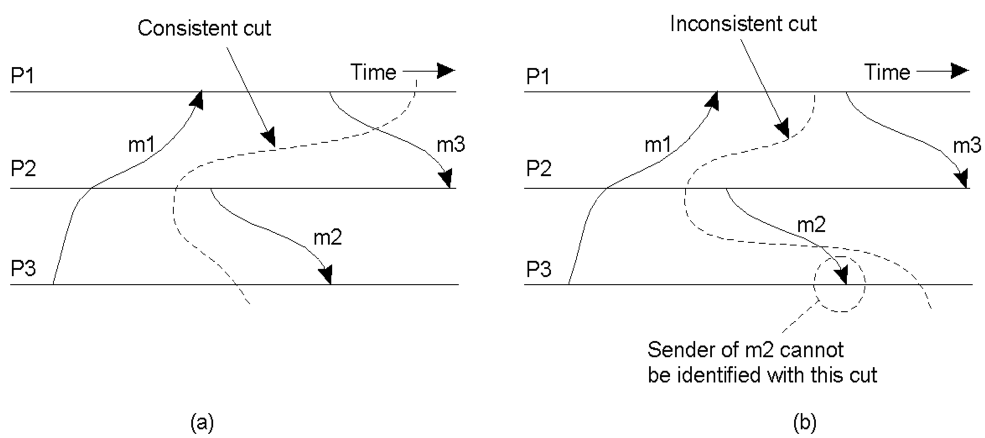
- Figure 6-13. Enforcing causal communication.



Part 2: Global State

- Global state of a distributed system
  - Local state of each process
  - Messages sent but not received (state of the queues)
- Many applications need to know the state of the system
  - Failure recovery, distributed deadlock detection
- Problem: how can you figure out the state of a distributed system?
  - Each process is independent
  - No global clock or synchronization
- Distributed snapshot: a consistent global state

Global State (1)

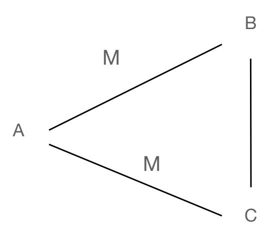


Distributed Snapshot Algorithm

- Assume each process communicates with another process using unidirectional point-to-point channels (e.g, TCP connections)
- Any process can initiate the algorithm
  - Checkpoint local state
  - Send marker on every outgoing channel
- On receiving a marker
  - Checkpoint state if first marker and send marker on outgoing channels, save messages on all other channels until:
  - Subsequent marker on a channel: stop saving state for that channel

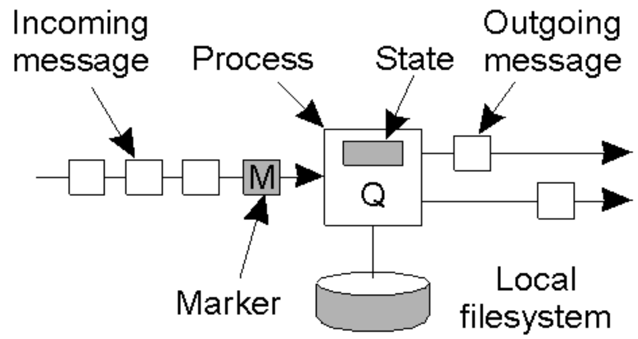
Distributed Snapshot

- A process finishes when
  - It receives a marker on each incoming channel and processes them all
  - State: local state plus state of all channels
  - Send state to initiator
- Any process can initiate snapshot
  - Multiple snapshots may be in progress
    - Each is separate, and each is distinguished by tagging the marker with the initiator ID (and sequence number)



Snapshot Algorithm Example

a) Organization of a process and channels for a distributed snapshot



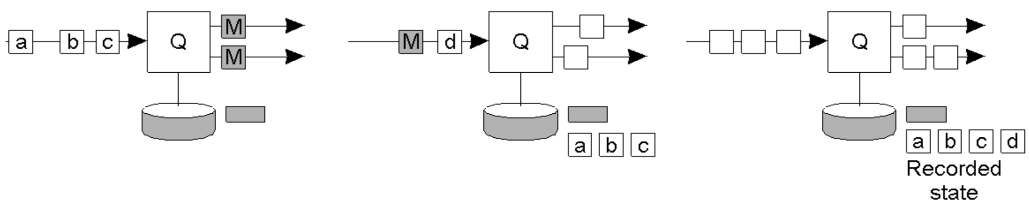
(a)

Snapshot Algorithm Example

b) Process Q receives a marker for the first time and records its local state

c) Q records all incoming message

d) Q receives a marker for its incoming channel and finishes recording the state of the incoming channel



Part 3: Termination Detection

- Detecting the end of a distributed computation
- Notation: let sender be *predecessor*, receiver be *successor*
- Two types of markers: Done and Continue
- After finishing its part of the snapshot, process Q sends a Done or a Continue to its predecessor
- Send a Done only when
  - All of Q's successors send a Done
  - Q has not received any message since it check-pointed its local state and received a marker on all incoming channels
  - Else send a Continue
- Computation has terminated if the initiator receives Done messages from everyone

Part 4: Election Algorithms

- Many distributed algorithms need one process to act as coordinator
  - Doesn't matter which process does the job, just need to pick one
- Election algorithms: technique to pick a unique coordinator (aka *leader election*)
- Examples: take over the role of a failed process, pick a master in Berkeley clock synchronization algorithm
- Types of election algorithms: Bully and Ring algorithms

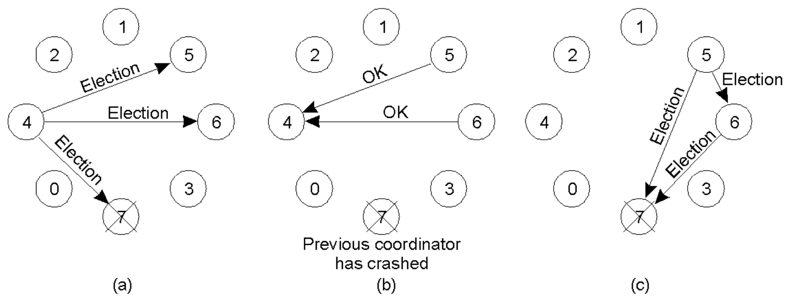
Bully Algorithm

- Each process has a unique numerical ID
- Processes know the Ids and address of every other process
- Communication is assumed reliable
- *Key Idea*: select process with highest ID
- Process initiates election if it just recovered from failure or if coordinator failed
- 3 message types: *election, OK, I won*
- Several processes can initiate an election simultaneously
  - Need consistent result
- $O(n^2)$ messages required with n processes

Bully Algorithm Details

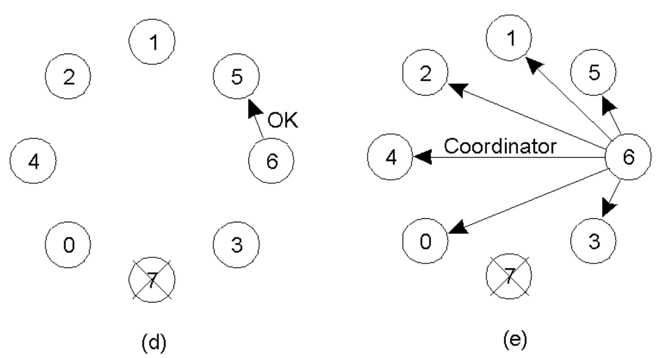
- Any process *P* can initiate an election
- *P* sends *Election* messages to all process with higher Ids and awaits *OK* messages
- If no *OK* messages, *P* becomes coordinator and sends *I won* messages to all process with lower Ids
- If it receives an *OK*, it drops out and waits for an *I won*
- If a process receives an *Election* msg, it returns an *OK* and starts an election
- If a process receives a *I won*, it treats sender an coordinator

Bully Algorithm Example



- The bully election algorithm
- Process 4 holds an election
- Process 5 and 6 respond, telling 4 to stop
- Now 5 and 6 each hold an election

Bully Algorithm Example



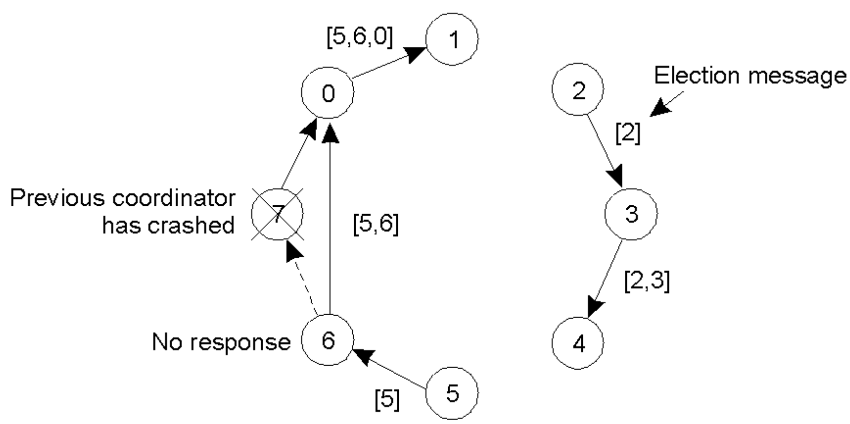
d) Process 6 tells 5 to stop

e) Process 6 wins and tells everyone

Ring-based Election

- Processes have unique Ids and arranged in a logical ring
- Each process knows its neighbors
  - Select process with highest ID
- Begin election if just recovered or coordinator has failed
- Send *Election* to closest downstream node that is alive
  - Sequentially poll each successor until a live node is found
- Each process tags its ID on the message
- Initiator picks node with highest ID and sends a coordinator message
- Multiple elections can be in progress
  - Wastes network bandwidth but does no harm

A Ring Algorithm



- Election algorithm using a ring.

Comparison

- Assume *n* processes and one election in progress

- Bully algorithm
  - Worst case: initiator is node with lowest ID
    - Triggers n-2 elections at higher ranked nodes: $O(n^2)$ msgs
  - Best case: immediate election: n-2 messages
- Ring
  - 2 (n-1) messages always