

Last Class: Clock Synchronization

- Physical clocks
- Clock synchronization algorithms
 - Cristian's algorithm
 - Berkeley algorithm
- Logical clocks

Today: More Canonical Problems

- Causality
 - Vector timestamps
- Global state and termination detection
- Election algorithms

Causality

- Lamport's logical clocks
 - If $A \rightarrow B$ 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*
 - Causal delivery: If $\text{send}(m) \rightarrow \text{send}(n) \Rightarrow \text{deliver}(m) \rightarrow \text{deliver}(n)$
 - 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

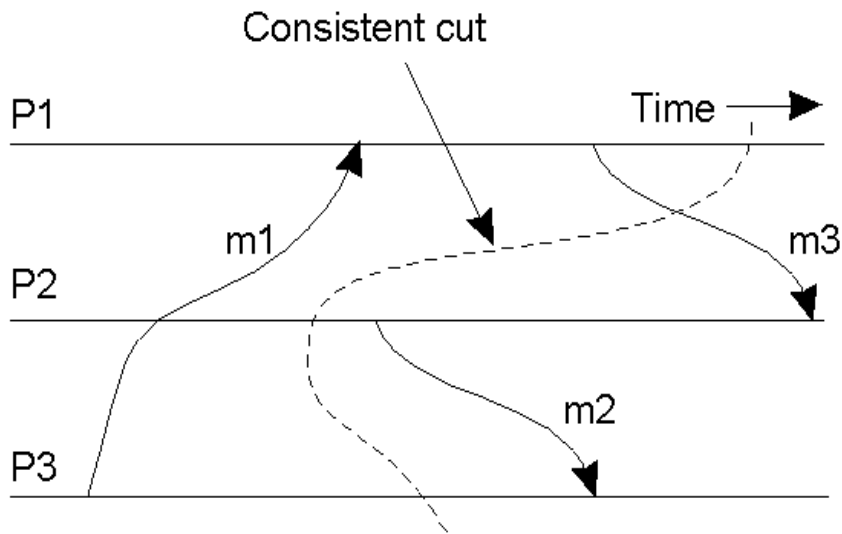
Vector Clocks

- 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 $V_j[i] = V_j[i] + 1$
- *Homework*: convince yourself that if $V(A) < V(B)$, then A causally precedes B

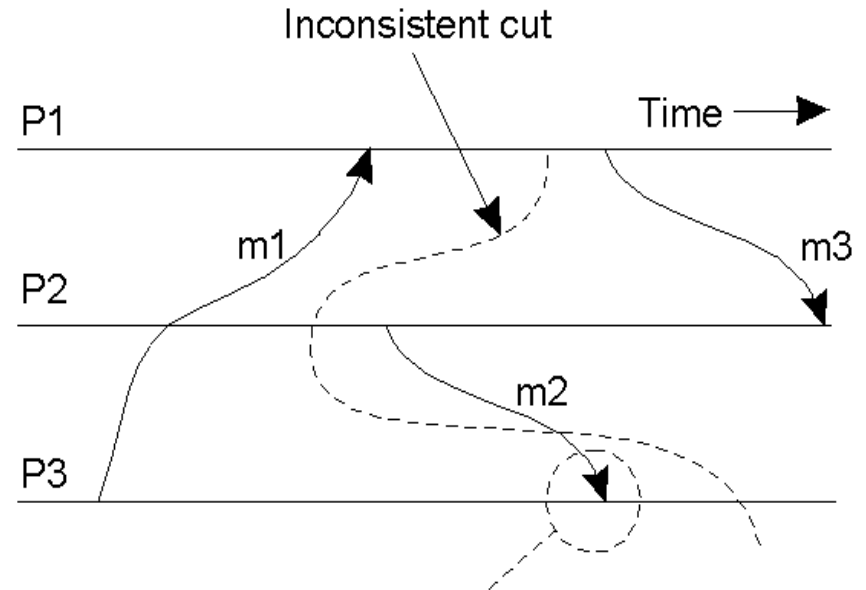
Global State

- 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)



(a)



(b)

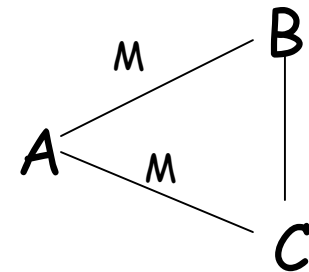
- a) A consistent cut
- b) An inconsistent cut

Distributed Snapshot Algorithm

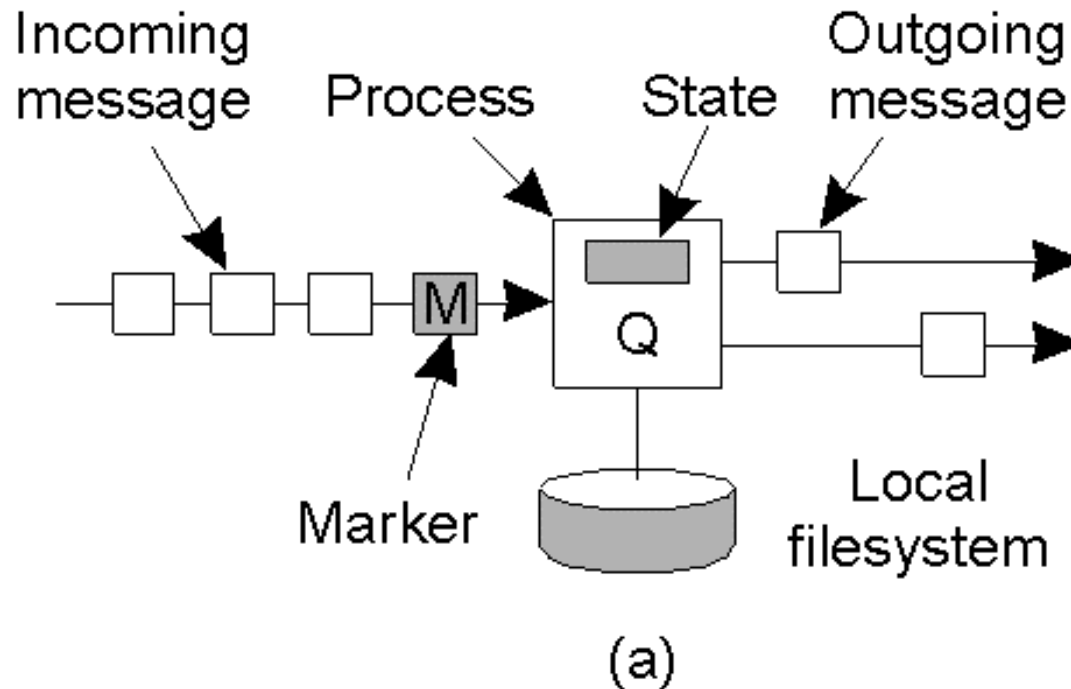
- Assume each process communicates with another process using unidirectional point-to-point channels (e.g, TCP connections)
- Any process can initiate the algorithm
 - Checkpoint local state
 - Send marker on every outgoing channel
- On receiving a marker
 - Checkpoint state if first marker and send 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)

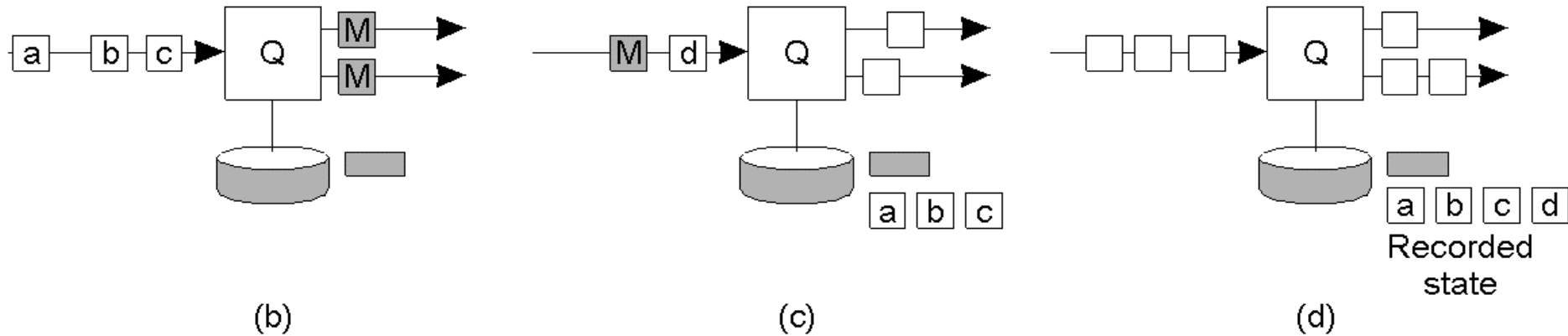


Snapshot Algorithm Example



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

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

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

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

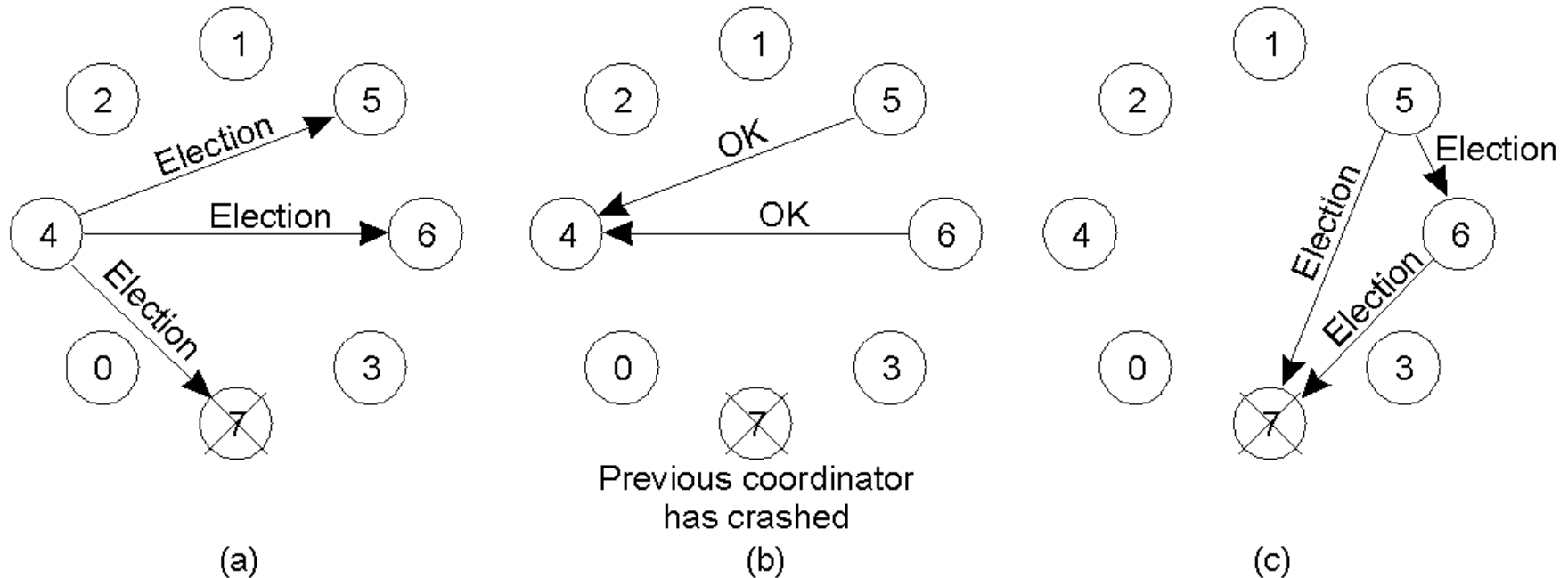
Bully Algorithm

- 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

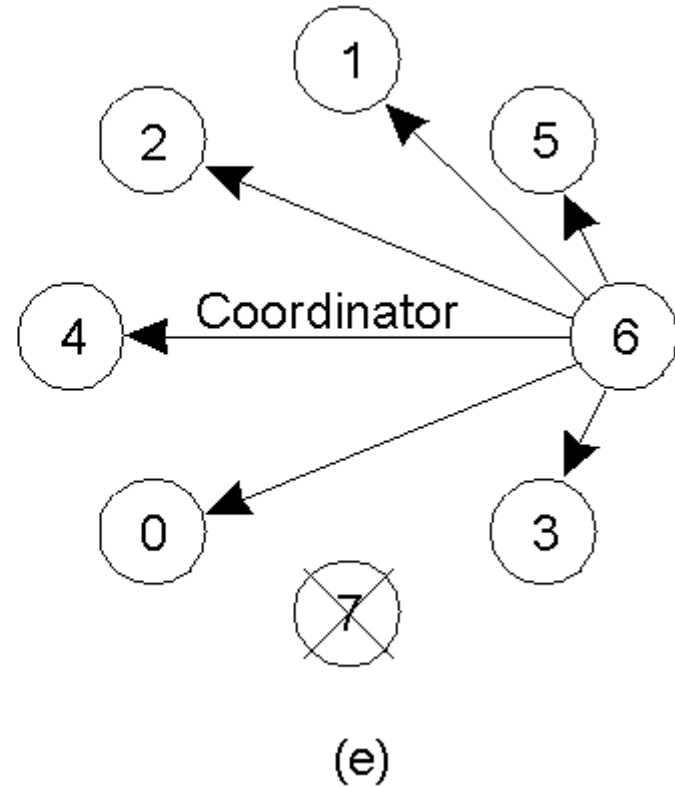
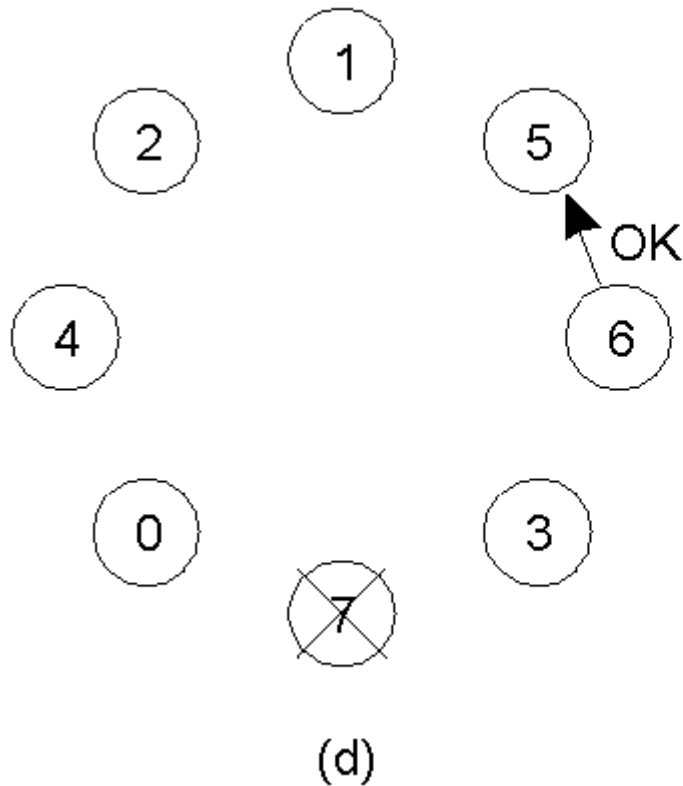
- Any process P can initiate an election
- P sends *Election* messages to all process with higher Ids and awaits *OK* messages
- If no *OK* messages, P becomes coordinator and sends *I won* messages to all process with lower Ids
- If it receives an *OK*, it drops out and waits for an *I won*
- If a process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