# Last Class: Naming

- Naming
  - Distributed naming
  - DNS
  - LDAP
-



# Today: Classical Problems in Distributed Systems

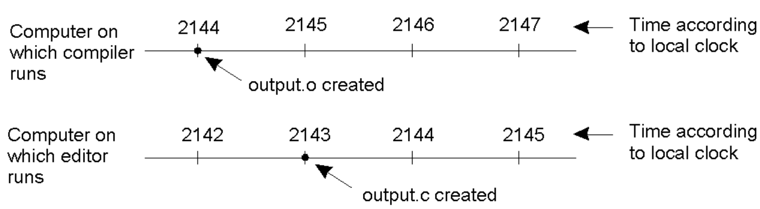
- Time ordering and clock synchronization (today)

Next few classes:

- Leader election
- Mutual exclusion
- Distributed transactions
- Deadlock detection
- CAP Theorem

# Clock Synchronization

- Time in unambiguous in centralized systems
  - System clock keeps time, all entities use this for time
- Distributed systems: each node has own system clock
  - Crystal-based clocks are less accurate (1 part in million)
  - *Problem:* An event that occurred after another may be assigned an earlier time



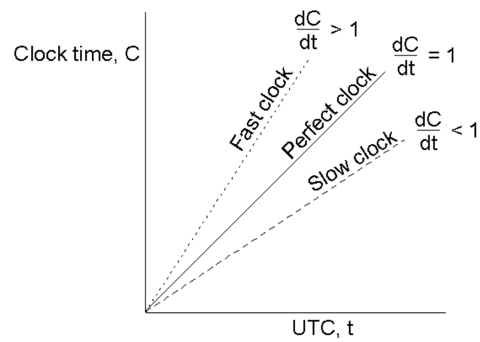



# Physical Clocks: A Primer

- How do you tell time?
  - Use astronomical metrics (solar day)
- Accurate clocks are atomic oscillators (one part in $10^{13}$)
- Coordinated universal time *(UTC)* – international standard based on atomic time
  - Add leap seconds to be consistent with astronomical time
  - UTC broadcast on radio (satellite and earth)
  - Receivers accurate to 0.1 – 10 ms
- Most clocks are less accurate (e.g., mechanical watches)
  - Computers use crystal-based blocks (one part in million)
  - Results in *clock drift*
- Need to synchronize machines with a master or with one another

# Clock Synchronization

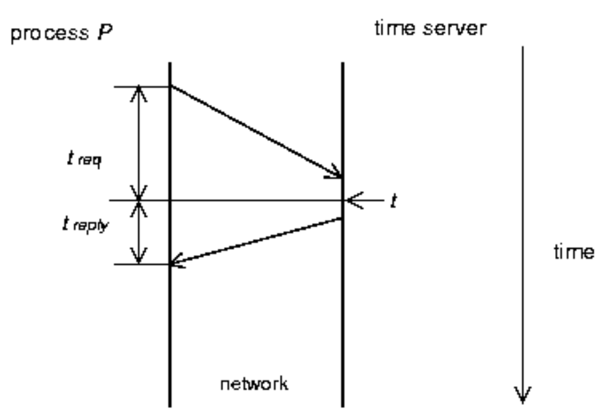
- Each clock has a maximum drift rate ρ
    - 1-ρ <= dC/dt <= 1+ρ
  - Two clocks may drift by 2ρ Δt in time Δt
  - To limit drift to δ => resynchronize every δ/2ρ seconds






# Cristian's Algorithm

- Synchronize machines to a *time server* with a UTC receiver
- Machine P requests time from server every δ/2ρ seconds
  - Receives time $t$ from server, P sets clock to $t+t_{reply}$ where $t_{reply}$ is the time to send reply to P
  - Use $(t_{req}+t_{reply})/2$ as an estimate of $t_{reply}$
  - Improve accuracy by making a series of measurements

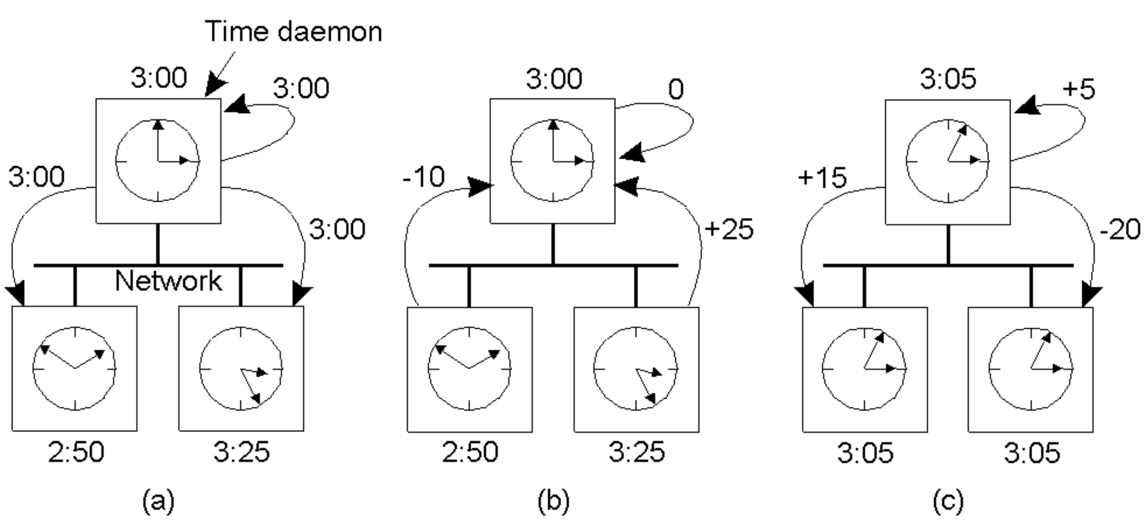


# Berkeley Algorithm

- Used in systems without UTC receiver
  - Keep clocks synchronized with one another
  - One computer is *master*, other are *slaves*
  - Master periodically polls slaves for their times
    - Average times and return differences to slaves
    - Communication delays compensated as in Cristian's algo
  - Failure of master => election of a new master



# Berkeley Algorithm



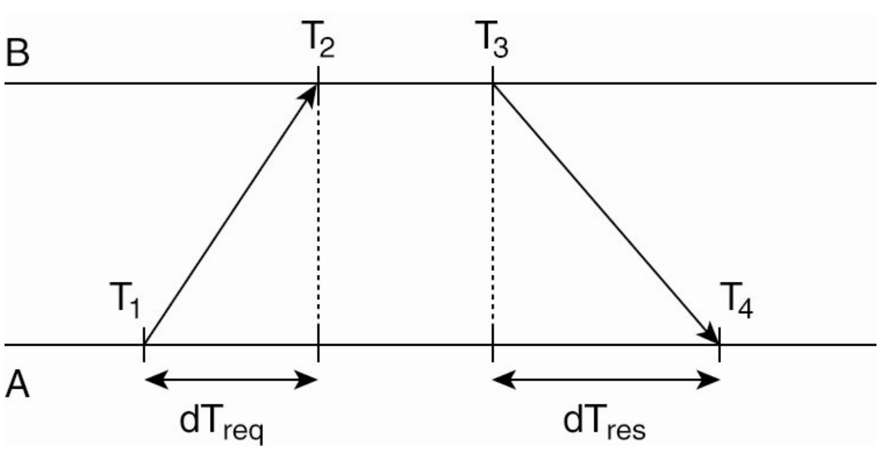

a) The time daemon asks all the other machines for their clock values
b) The machines answer
c) The time daemon tells everyone how to adjust their clock

# Distributed Approaches

- Both approaches studied thus far are centralized
- Decentralized algorithms: use resync intervals
  - Broadcast time at the start of the interval
  - Collect all other broadcast that arrive in a period *S*
  - Use average value of all reported times
  - Can throw away few highest and lowest values
- Approaches in use today
  - *rdate*: synchronizes a machine with a specified machine
  - Network Time Protocol (NTP) - discussed in next slide
    - Uses advanced techniques for accuracies of 1-50 ms

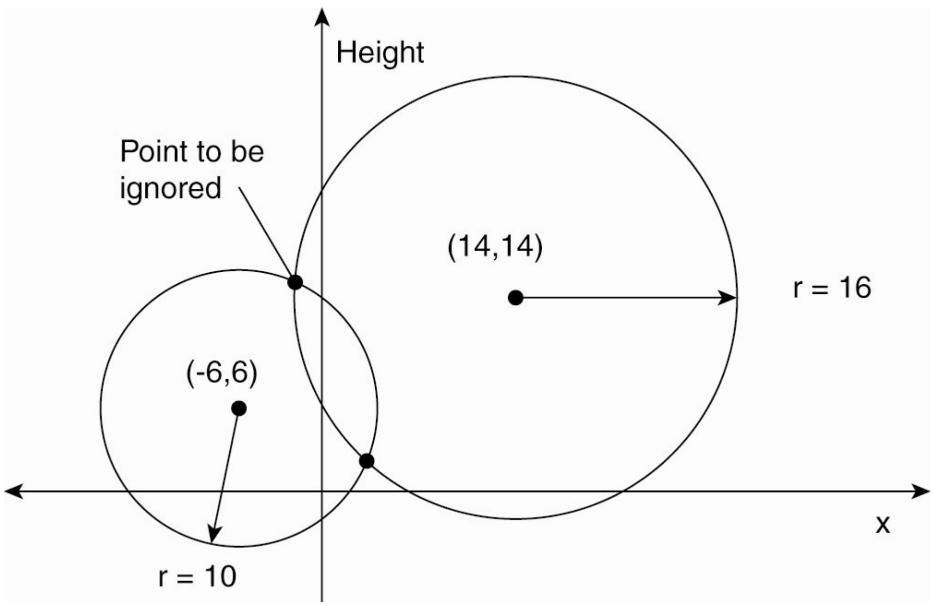


# Network Time Protocol




- Widely used standard - based on Cristian's algo
  - Uses eight pairs of delays from A to B and B to A.
- Hierarchical – uses notion of stratum
- **Clock can not go backward**

# Global Positioning System




- Computing a position in a two-dimensional space.



# Global Positioning System

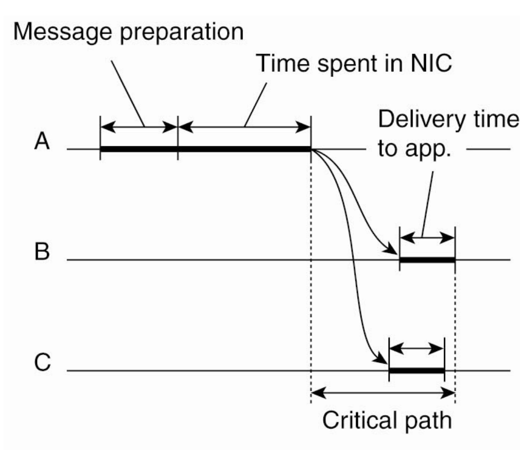
- Real world facts that complicate GPS
- It takes a while before data on a satellite's position reaches the receiver.
- The receiver's clock is generally not in synch with that of a satellite.

# GPS Basics

- $D_r$ – deviation of receiver from actual time
- Beacon with timestamp $T_i$ received at $T_{now}$
  - Delay $D_i = (T_{now} - T_i) + D_r$
  - Distance $d_i = c\,(T_{now} - T_i)$
  - Also $d_i = \text{sqrt}[\,(x_i-x_r)^2 + (y_i-y_r)^2 + (z_i-z_r)^2\,]$
- Four unknowns, need 4 satellites.
-



# Clock Synchronization in Wireless Networks




(b)

- Reference broadcast sync (RBS): receivers synchronize with one another using RB server
  - Mutual offset = $T_{i,s} - T_{j,s}$ (can average over multiple readings)

# Logical Clocks

- For many problems, internal consistency of clocks is important
  - Absolute time is less important
  - Use *logical* clocks
- Key idea:
  - Clock synchronization need not be absolute
  - If two machines do not interact, no need to synchronize them
  - More importantly, processes need to agree on the *order* in which events occur rather than the *time* at which they occurred



# Event Ordering

- *Problem:* define a total ordering of all events that occur in a system
- Events in a single processor machine are totally ordered
- In a distributed system:
  - No global clock, local clocks may be unsynchronized
  - Can not order events on different machines using local times
- Key idea [Lamport ]
  - Processes exchange messages
  - Message must be sent before received
  - Send/receive used to order events (and synchronize clocks)

# Happened Before Relation

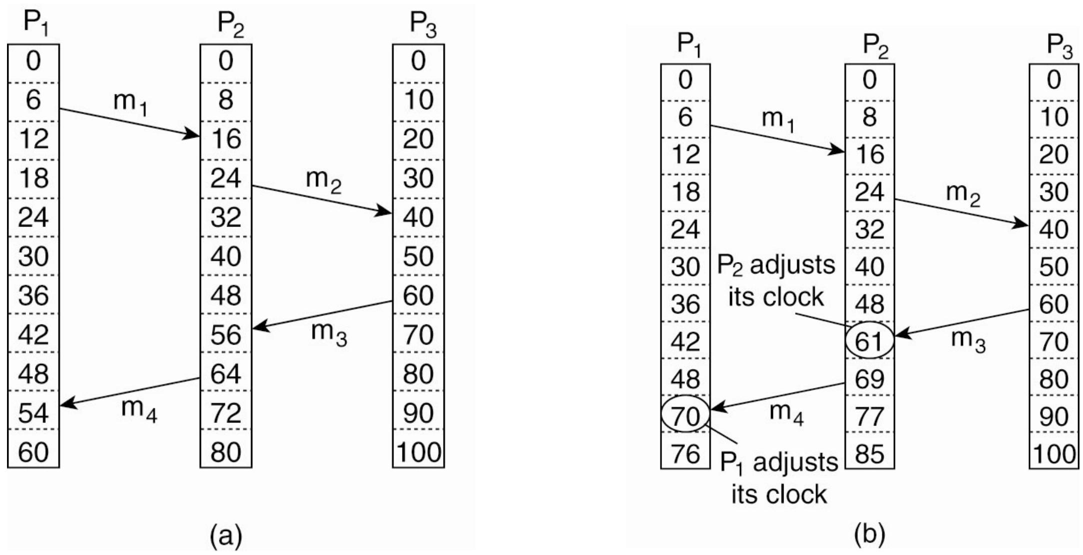
- If *A* and *B* are events in the same process and *A* executed before *B*, then *A* -> *B*
- If A represents sending of a message and B is the receipt of this message, then A -> B
- Relation is transitive:
  - A -> B and B -> C => A -> C
- Relation is undefined across processes that do not exchange messages
  - Partial ordering on events



# Event Ordering Using *HB*

- Goal: define the notion of time of an event such that
  - If A-> B then C(A) < C(B)
  - If A and B are concurrent, then C(A) <, = or > C(B)
- Solution:
  - Each processor maintains a logical clock $LC_i$
  - Whenever an event occurs locally at I, $LC_i = LC_i+1$
  - When *i* sends message to *j*, piggyback $Lc_i$
  - When *j* receives message from *i*
    - If $LC_j < LC_i$ then $LC_j = LC_i +1$ else do nothing
  - Claim: this algorithm meets the above goals

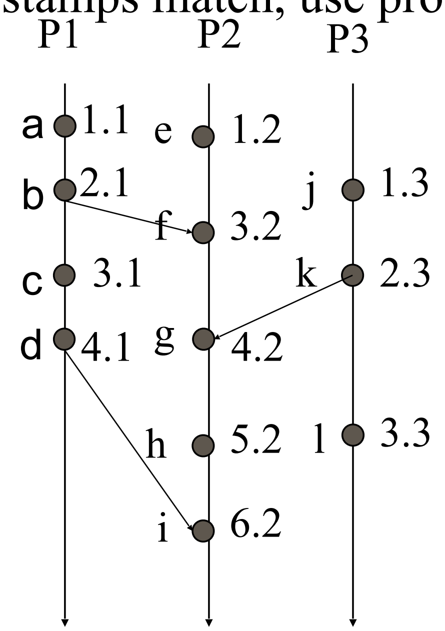
# Lamport's Logical Clocks



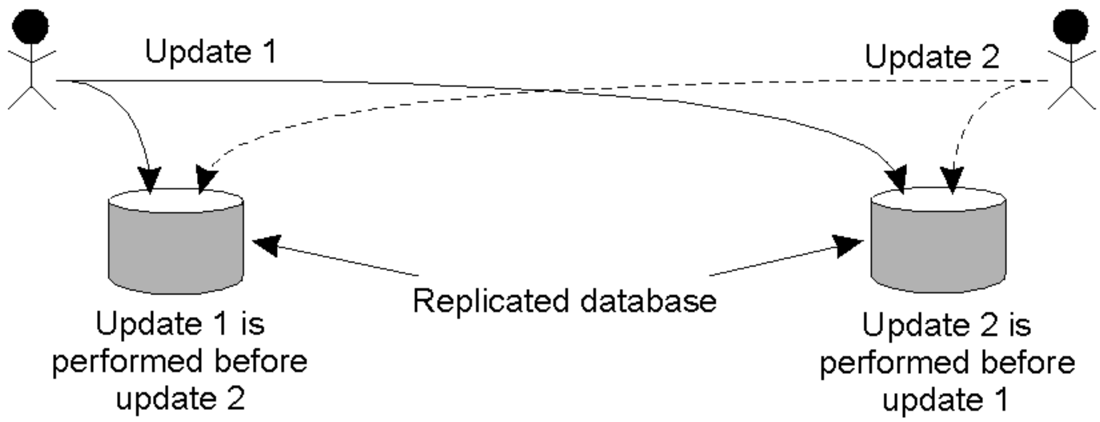



# Total Order

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



# Example: Totally-Ordered Multicasting






# 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

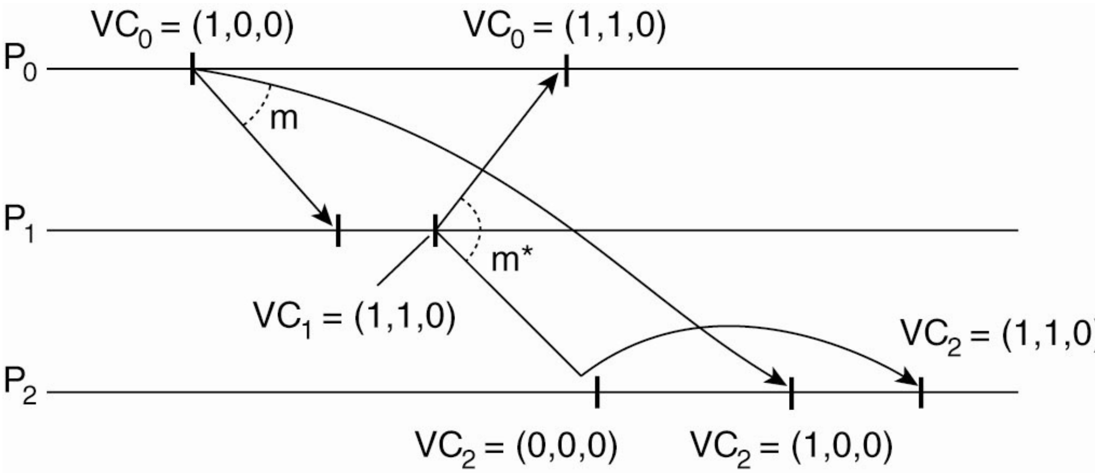
- Lamport's logical clocks
  - If *A -> B* then *C(A) < C(B)*
  - Reverse is not true!!
    - Nothing can be said about events by comparing time-stamps!
    - If *C(A) < C(B),* then ??
- Need to maintain *causality*
  - If a -> b then a is casually related to b
  - *Causal delivery*:If send(m) -> send(n) => 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 $V_j[i] = V_j[i]+1$
- *Exercise:* prove that if *V(A)<V(B),* then *A* causally precedes *B* and the other way around.

# Enforcing Causal Communication

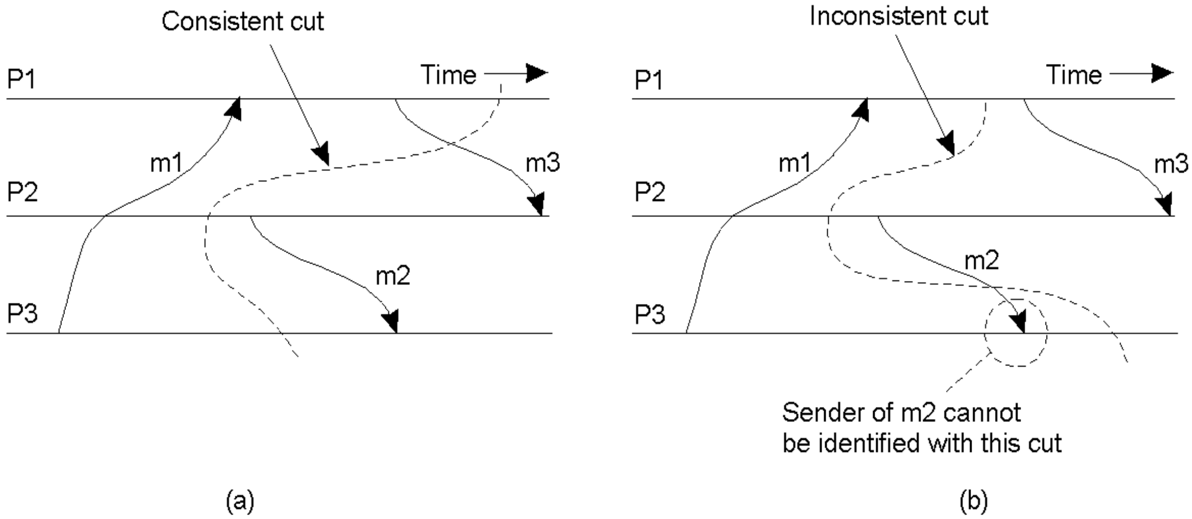
- 




# 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) A consistent cut
b) An inconsistent cut

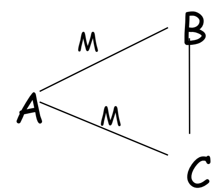


# Distributed Snapshot Algorithm

- 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

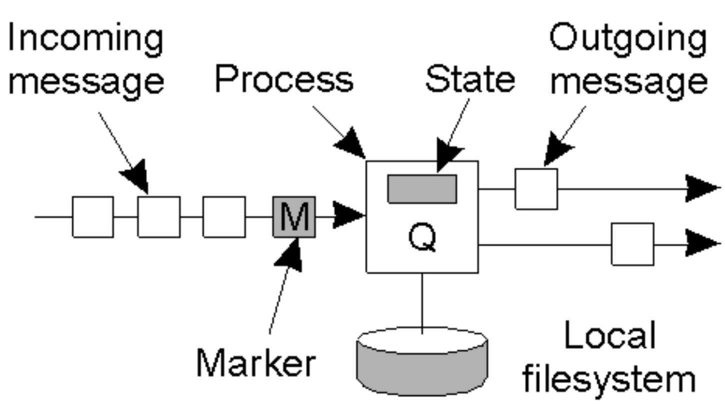
# Distributed Snapshot

- 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)





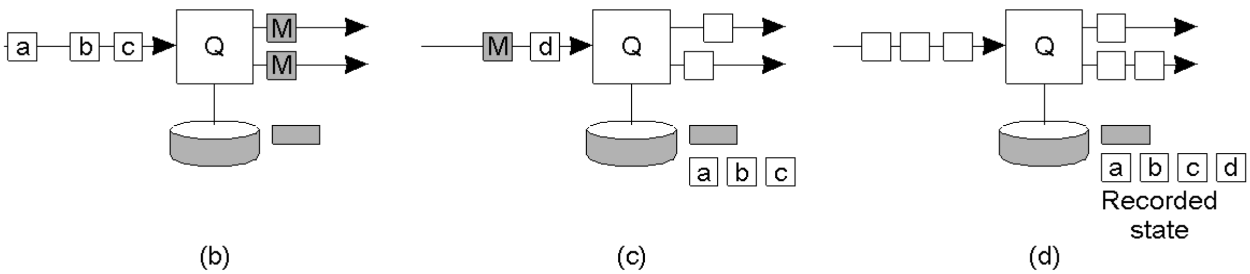

# Snapshot Algorithm Example




(a)

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