Distributed Systems Lecture 6

Professor Louise E. Moser

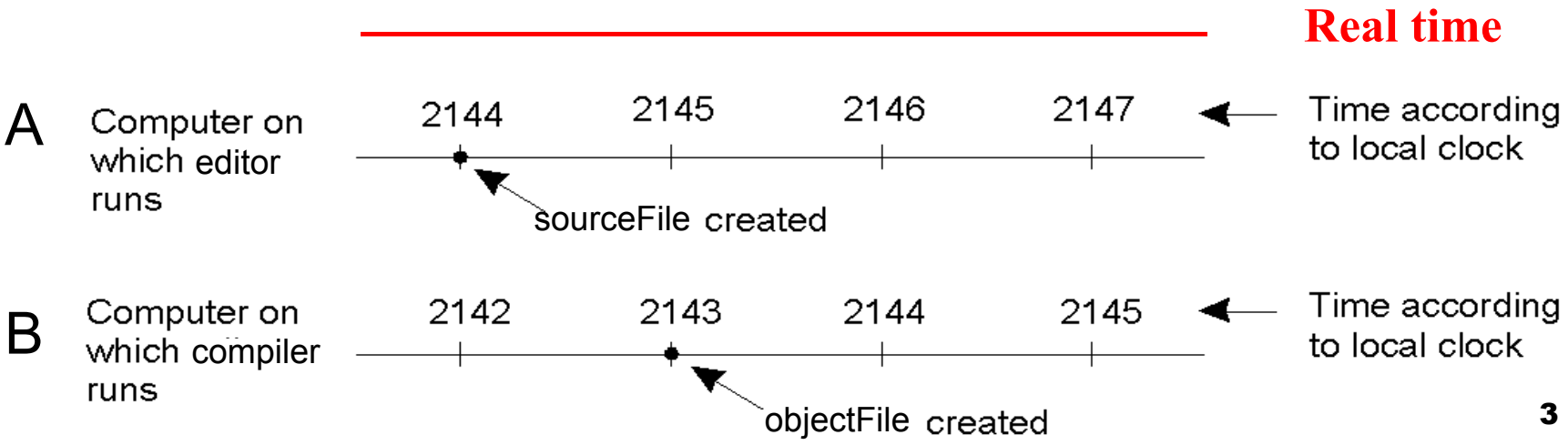
Spring 2014

Clocks

- Physical clocks
- Logical clocks
- Vector clocks

Clock Synchronization

- **Problem:** Each computer in a distributed system has its own local physical hardware clock; synchronizing the clocks exactly is difficult, if not impossible
- In real time, an event (editing) on computer A occurs before an event (compiling) on computer B, but the clocks on A and B indicate that compling occurred before editing

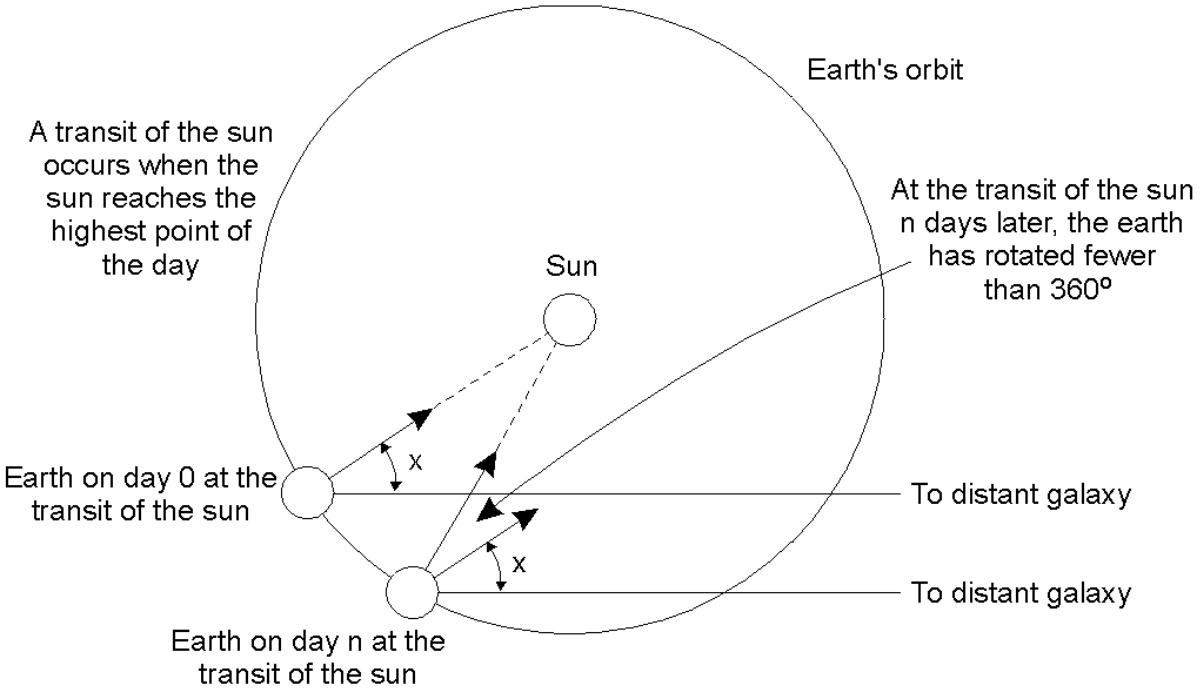


Physical Clocks

- **Problem:** In some (real-time) distributed systems, we need an exact time, not just an ordering of events
- **Possible Solution:** **Universal Coordinated Time (UTC)**
  - **UTC gives the time of day**
  - **Based on the number of transitions per second of the cesium 133 atom, which is quite accurate**
  - **The real time is taken as the average of 50 cesium clocks around the world**
  - **Introduce a leap second from time-to-time to compensate for days that are getting longer due to tidal and atmospheric drag**
  - **UTC is broadcast through shortwave radio or satellite**
    - Satellites can give an accuracy of about 0.5ms
- **Questions:** Does UTC solve the problem?
  Don't we now have a global timing mechanism?

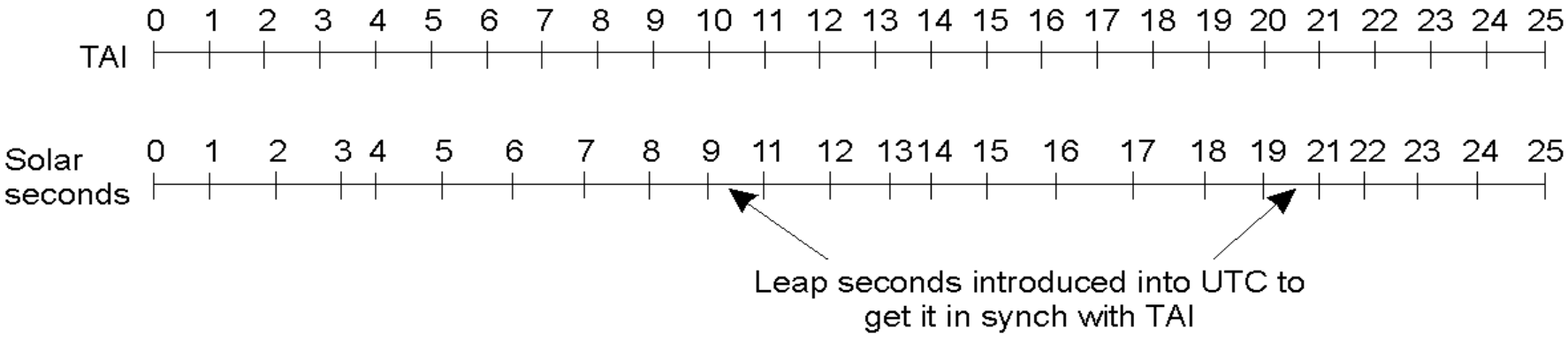
Physical Clocks

- Computation of the mean **solar day**
  - **The solar day is slightly shorter (about 4 minutes) than the distant galaxy day (the Andromeda day)**
  - **One solar day corresponds to about 359° rotation of the Earth about its axis and about 1° rotation of the Earth about the sun**



Physical Clocks

- **International Atomic Time (TAI)** seconds, obtained from cesium atoms, are nearly constant in length, unlike solar seconds
- Leap seconds are introduced, when necessary, to keep in phase with the sun

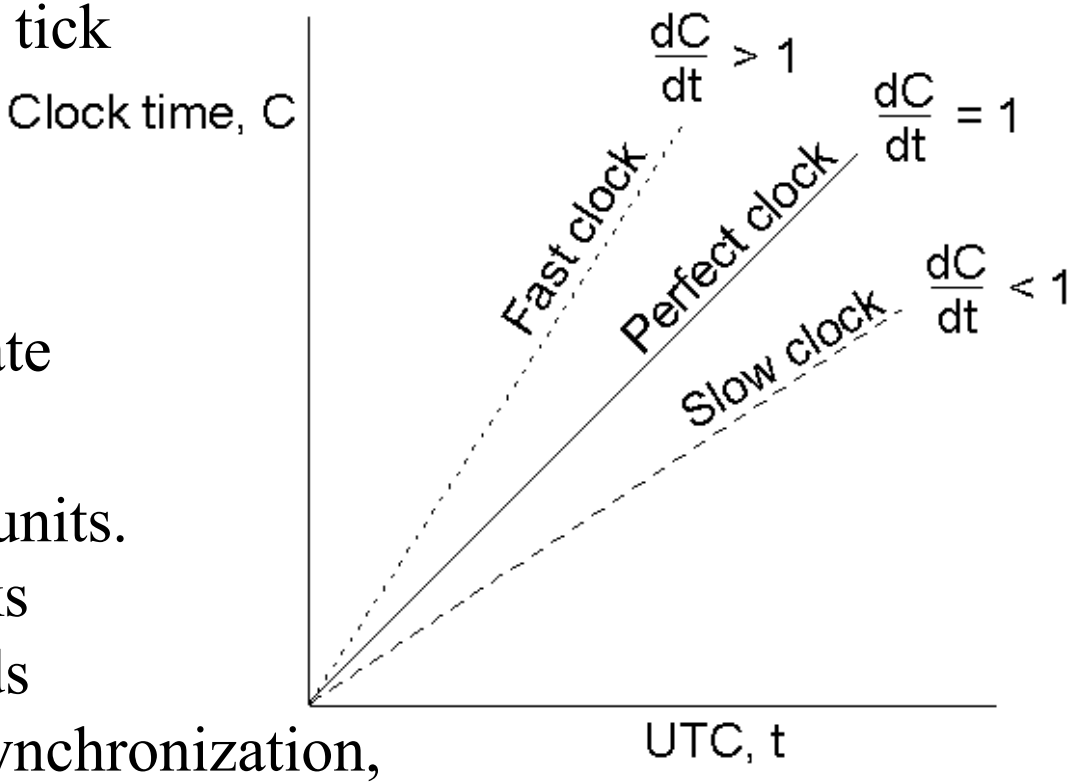


Physical Clocks

- **Problem:** Suppose we have a distributed system with a UTC receiver somewhere within it
  - **We still have to distribute the UTC time *t* to each machine (processor) in the distributed system**
- **Basic idea**
  - **Each processor *p* has a timer that generates an interrupt *h* times per second**
  - **The clock in processor *p* ticks on each timer interrupt**
    - Denote the value of that clock by *Cp(t)*, where *t* is UTC time
  - **Ideally, we have that, for each processor *p*, *Cp(t)* = *t* and, thus, *dCp/dt* = 1**

Clock Synchronization Algorithms

- The relation between clock time and UTC time when clocks tick at different rates
- In practice:
  $1 - \rho < dC/dt < 1 + \rho$
  where ρ is the clock drift rate
- **Goal:** Never let two clocks differ by more than δ time units. Then synchronize the clocks at least every $\delta/(2\rho)$ seconds
  Why? Δt time units after synchronization, two clocks can be at most $2\rho\Delta t$ apart. Thus, $2\rho\Delta t \leq \delta$ or $\Delta t \leq \delta/(2\rho)$

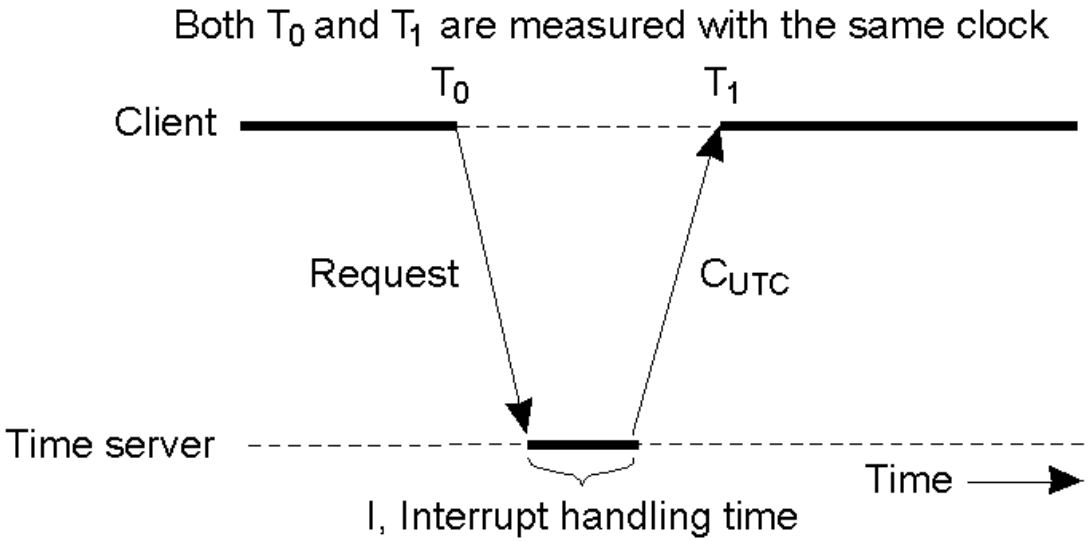


Clock Synchronization

- **Solution I:** **Each machine queries a time server** for the accurate time at least once every $\delta/(2\rho)$ seconds
  - **OK, but you need an accurate measurement of round-trip delay, including interrupt handling and processing of incoming messages**
- **Solution II:** **The time server queries all machines periodically**, calculates an average, and informs each machine how it should adjust its time relative to its current time
  - **OK, you'll probably get the machines more or less in sync**
  - **Alternatively, the time server could distribute UTC time, which it gets from the Internet or a GPS receiver**
- **Fundamental Problem:** Time must **never** be set backwards. Instead, change the clock rate, i.e., slow down the fast clocks

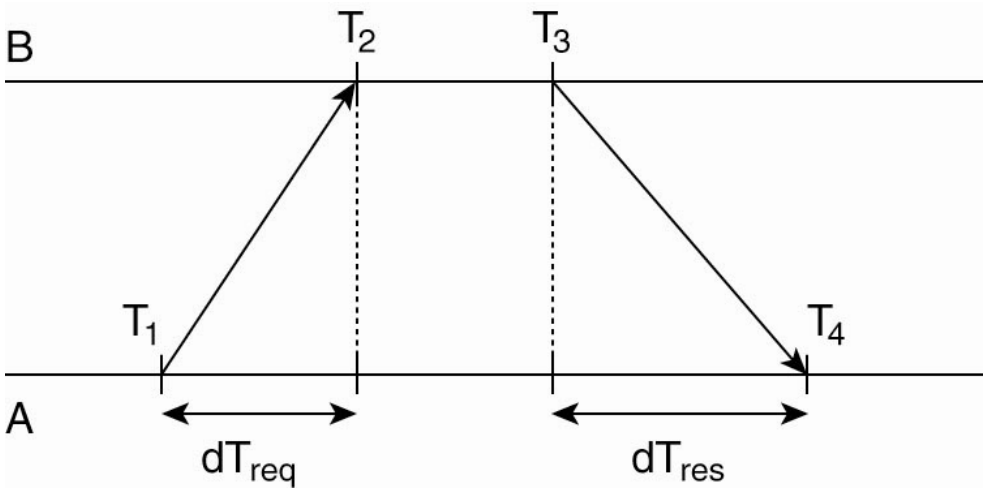
Cristian's Algorithm

- Get the current time from a time server
- The communication introduces a round-trip delay, which can be highly variable
- Make multiple requests to the time server and choose the **time in the response with the smallest round-trip delay**



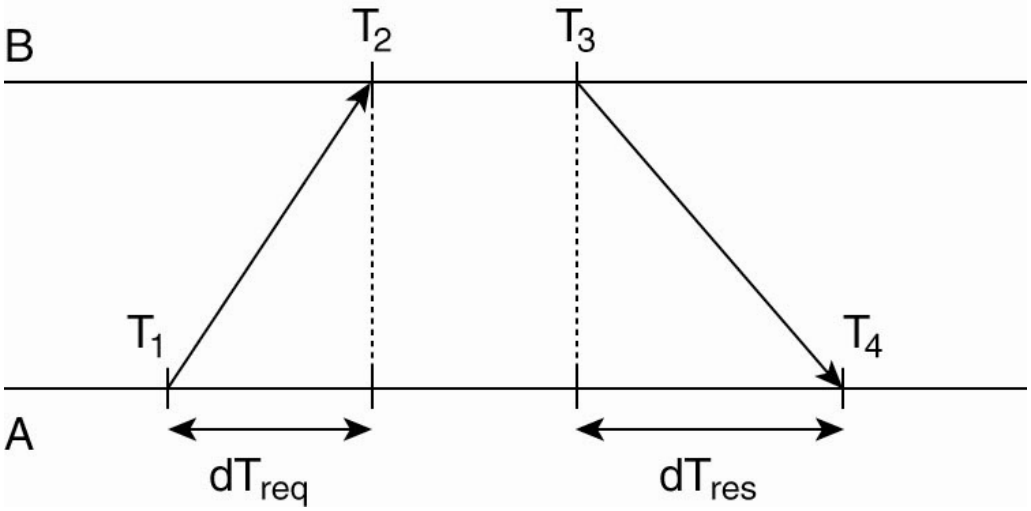
Network Time Protocol (NTP)

- A requests the current time from the time server B
- B responds to A with T2 and T3
- A calculates an **offset θ** (A's time T4 relative to B's time T3 + average one-way propagation delay) as follows:
  $\theta = T3 + ((T2-T1) + (T4-T3))/2 - T4 = (T2-T1)+(T3-T4))/2$
- If $\theta < 0$, A would set its clock backward which is not allowed;
  in this case, A must reduce its clock gradually by reducing its clock rate

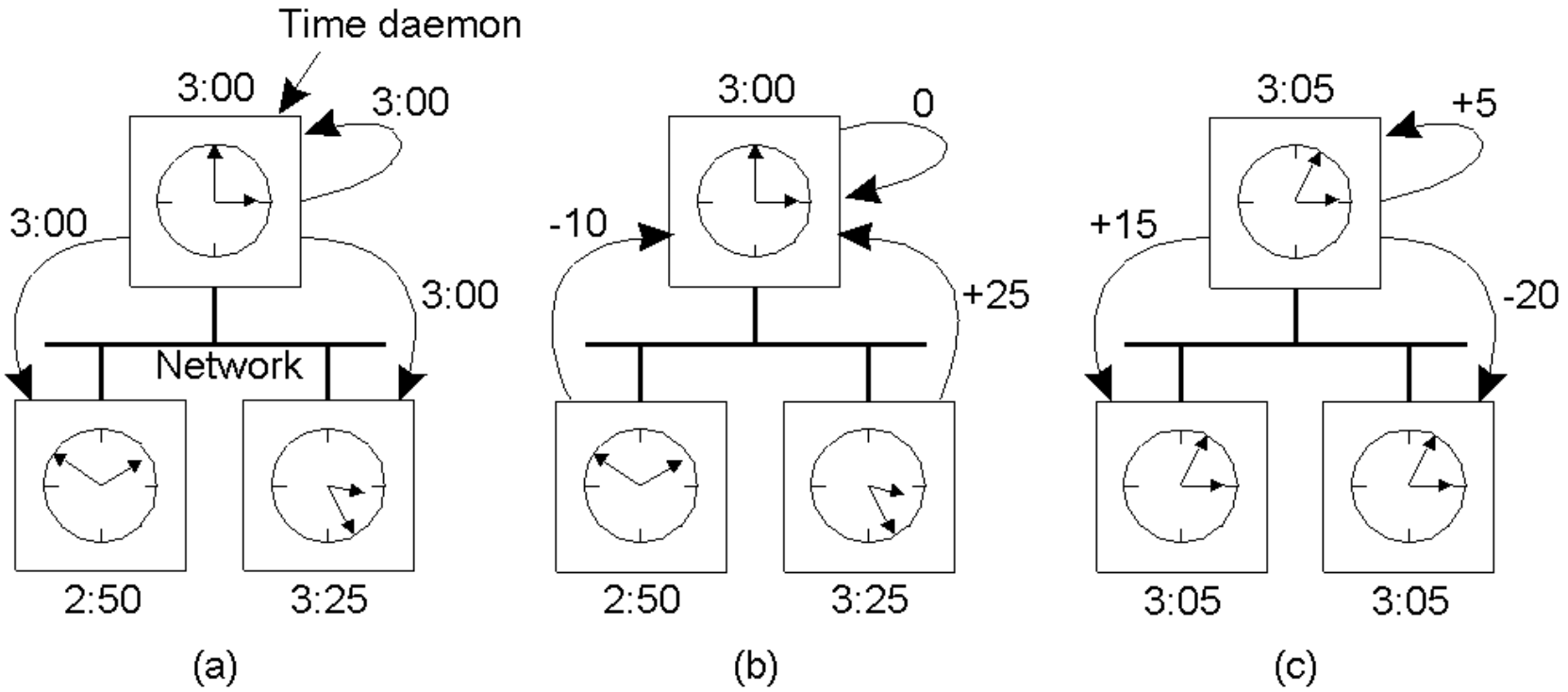


Network Time Protocol

- A calculates a **delay δ** (average one-way propagation delay) as follows:
  $\delta = ((T4–T1)–(T3–T2))/2 = ((T4–T3)+(T2–T1))/2$
- Eight pairs (θ, δ) of offsets, delays are calculated
  - **The minimum value of the delay δ corresponds to the best estimate of the offset θ**
- However, if one of the two clocks is known to be more accurate than the other, some other method should be used



The Berkeley Algorithm



(a) The time daemon asks all the other machines for their clock values
(b) The machines respond with their deviations
(c) The time daemon computes an average time and tells every machine how to adjust its clock

The Happened-Before Relation

- **Problem:** First, we need a notion of ordering before we can order anything
- The **happened-before** relation $\rightarrow$ on the set of events in a distributed system is the smallest relation satisfying:
  - **If *a* and *b* are two events in the same process, and *a occurs* before *b*, then $a \rightarrow b$**
  - **If *a* represents the sending of a message *m* and *b* represents the receiving of the message *m*, then $a \rightarrow b$**
  - **If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$ (referred to as transitivity)**
- **Note:** The happened-before relation $\rightarrow$ introduces a partial order of events in a distributed system with concurrently operating processes

Logical Clocks

- **Problem:** How do we maintain a **global view** of the system's behavior that is consistent with the happened-before relation?
- **Solution:** Attach a **logical clock (timestamp)** *C(e)* to each event *e*, that satisfies the following properties:

  L1: If *a* and *b* are two events in the same process and $a \rightarrow b$, then *C(a)* < *C(b)*

  L2: If *a* represents the sending of a message *m* and *b* represents the receiving of the message *m*, then *C(a)* < *C(b)*

Logical Clocks

- **Problem:** Two events in different processes in a distributed system can happen at the same (logical or physical) time
- **Solution:** To distinguish these events, process *Pi* attaches its process number *i* to event *e,* that is:

  Pi timestamps event *e* with *Ci(e)* and *i*

- Then ***a* happened before *b*** if and only if

  (1) *Ci(a)* < *Cj(b)*, or

  (2) *Ci(a)* = *Cj(b)* and *i* < *j*

- **Question:** How do we maintain a **consistent** set of logical clocks, one per process?

Lamport's Algorithm

- Each process *Pi* maintains a **local counter *Ci***, and updates *Ci* according to the following rules:

  (1) On occurrence of an event (other than sending or receiving a message), process *Pi* updates its local counter *Ci* ← *Ci* + *1*

  (2) When process *Pi* sends a message *m*, process *Pi* updates its local counter *Ci* ← *Ci* + *1* and sets message *m*'s timestamp *Tm* to C_i

  (3) When process *Pj* receives a message *m* with timestamp *Tm*, process *Pj* updates its local counter *Cj* ← max(*Cj*, *Tm*) + 1

  (4) Process Pj then delivers the message to the application (not necessarily in causal or total order)

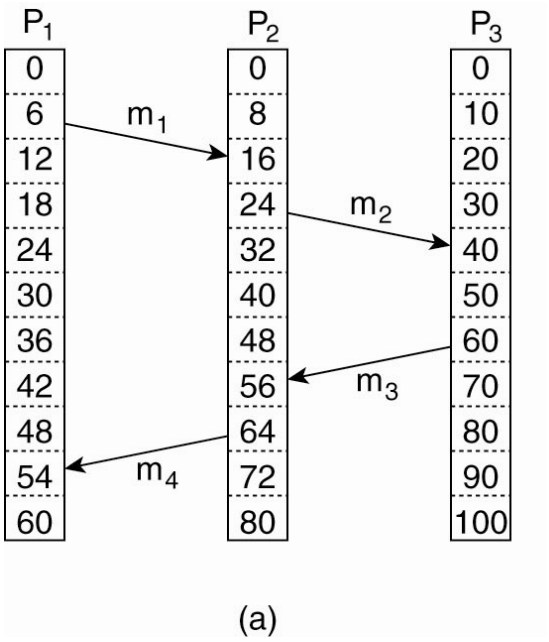
- Property **L1** is satisfied by (1)
- Property **L2** is satisfied by (2) and (3)

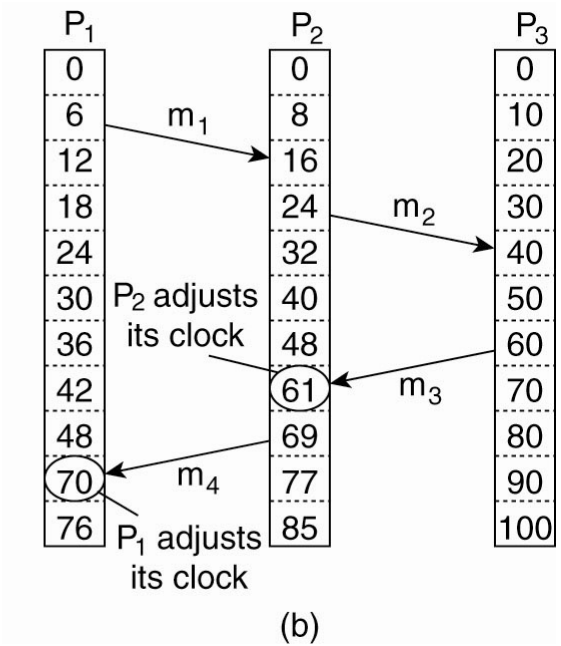
Lamport's Algorithm

(a) Three processes with physical clocks that run at different rates

- **6 is the time at which m1 is sent by P1, 16 is the time at which m1 is received by P2, etc**

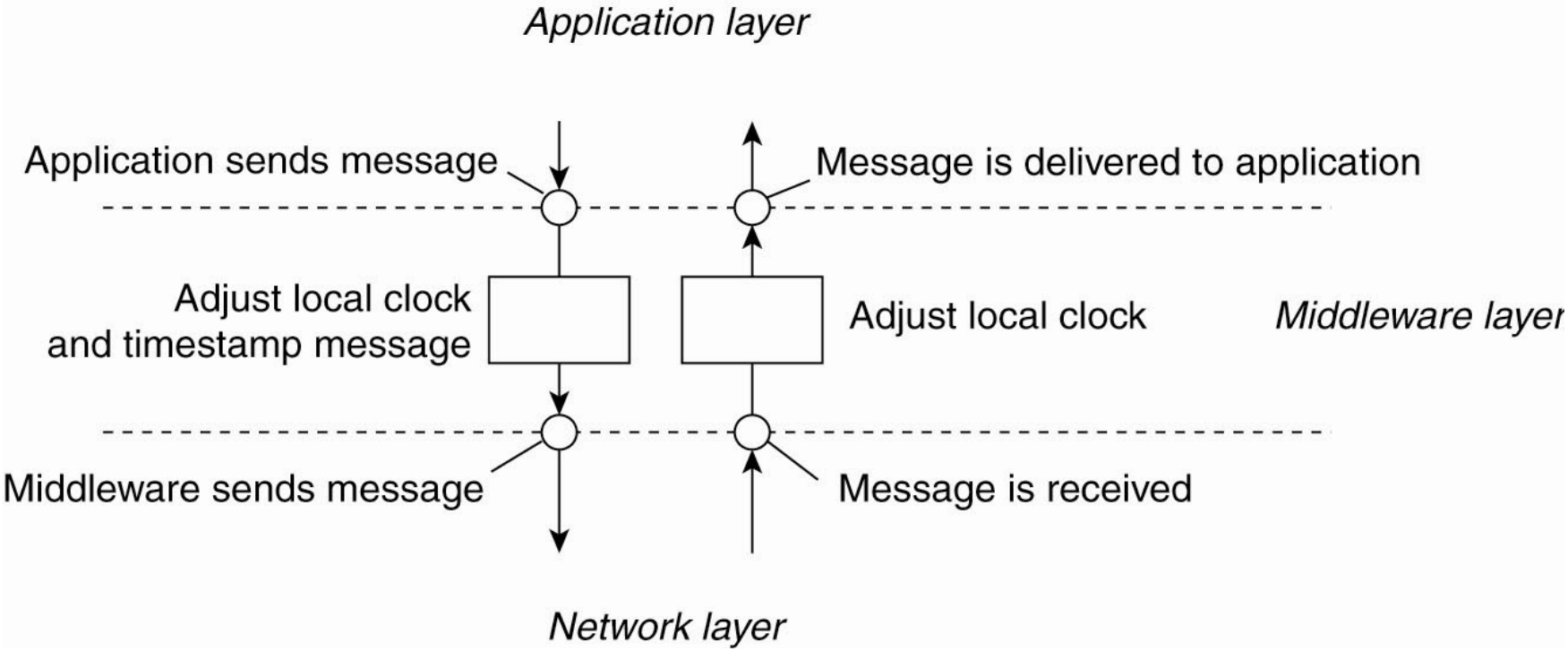
(b) Logical clocks of the three processes, using Lamport's algorithm applying Rules (1) - (3)





Lamport's Logical Clocks

- The position of Lamport's logical clocks in a distributed system



Vector Timestamps

- Lamport's logical clocks (timestamps) do **not** guarantee that if $C(a) < C(b)$, then indeed a happened before b
- For that, we need **vector timestamps:** Each process Pi has a vector $VCi[1..n]$, where $VCi[j]$ denotes the number of events that process Pi knows have taken place at process Pj

(1) Before processing an event, process Pi updates ***VCi[i] ← VCi[i] + 1***

(2) When process Pi sends a message m, it

Updates *VCi[i] ← VCi[i] + 1*, and

Sends *VCi* as a vector timestamp *vt(m)* with the message *m*

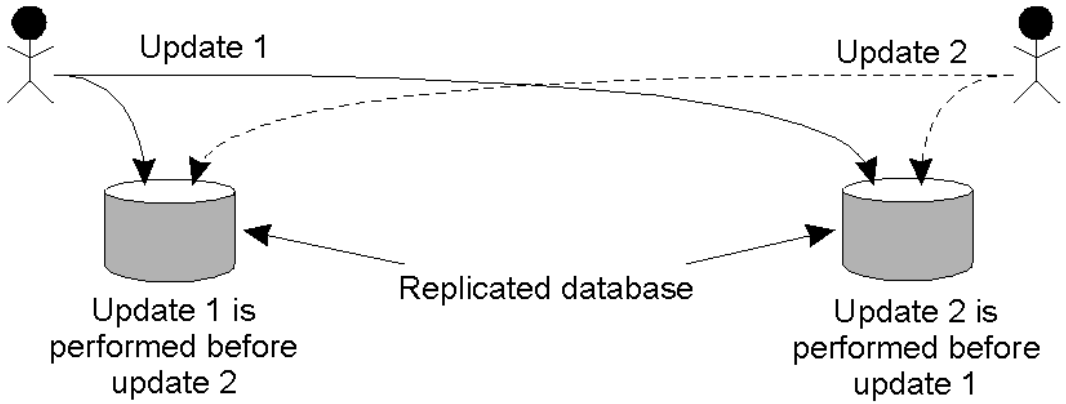
(3) When a process Pj receives a message m from process Pi with vector timestamp $vt(m)$, it

Updates *VCj[k] ← max(VCj[k], vt(m)[k])* for all *k*, and

Updates *VCj[j] ← VCj[j] + 1*

Total Ordering of Events

- **Problem:** Sometimes we need to guarantee that concurrent updates of a replicated database are seen in the same order everywhere
  - **(1) Process *P*1 adds $100 to an account (initial value $1000)**
  - **(2) Process *P*2 increments the account by 1% interest**
- There are two database replicas R1 and R2



- **Outcome:** In the absence of proper synchronization
  - **Replica R1 ends up with $1111 = (1000+100)x1.01**
  - **Replica R2 ends up with $1110 = (1000x1.01)+100**

Totally Ordered Multicast

- Process *Pi* increments its **logical clock** and sends a message *mi* with that **timestamp** to all the other processes and puts the outgoing message *mi* into its local queue *qi*
- Process *Pj* puts the incoming message *mi* into its local queue q*j* ordered according to the timestamp of *mi*
- Process *Pj* updates its logical clock and delivers a message *ml* to the application iff:

  (1) Message *ml* is at the head of its queue q*j,* and

  (2) For each process *Pk, k≠ l*, there is a message *mk* in *Pj*' s queue *qj* with a timestamp greater than that of *ml*

  (Thus, no message from *Pk* with a timestamp less than that of *ml* can arrive later.)
- **Note:** We are assuming that communication is reliable, source ordered, and timely

Causally Ordered Multicast

- Process *Pi* increments its **vector clock** and sends a message *mi* with a **vector timestamp *vt(mi)*** to all the other processes and puts the outgoing message *mi* into its local queue *qi*
- Process *Pj* puts the incoming message *mi* from process *Pi* into its local queue *qi* for process *Pi* according to the timestamp *vt(mi)[i]*
- Process *Pj* updates its vector clock and delivers a message *ml* to the application iff:

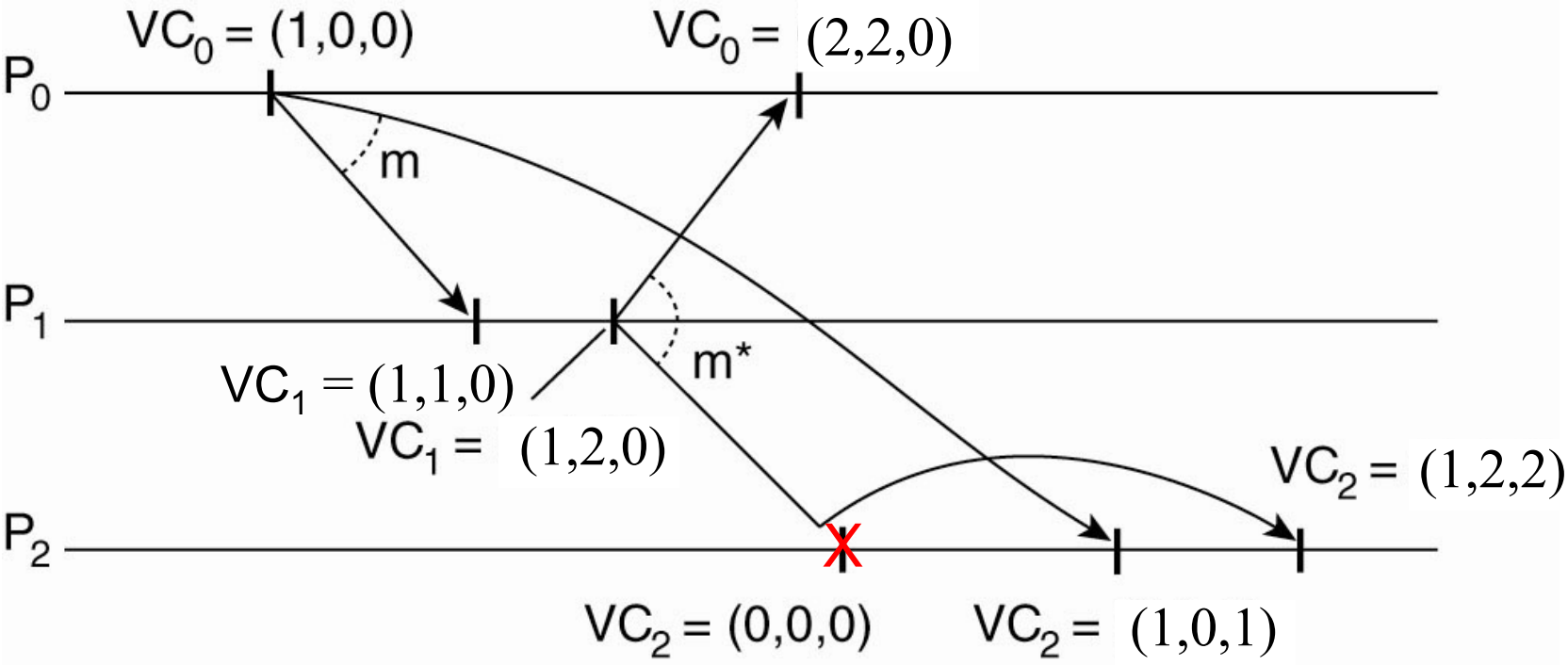
  (1) Message *ml* is at the head of its queue *ql* for process *Pl,* and

  (2) For each process *Pk, k≠ l,* queue *qk* is not empty and message *mk* at the head of queue *qk* satisfies *vt(mk)[k] > vt(ml)[k]*

- **Note:** Again, we are assuming that communication is reliable, source ordered, and timely

Causally Ordered Multicast

- Enforcing causal ordering using vector timestamps
- Assume all processes start with the vector clock (0,0,0)
- X indicates that m* is not delivered at that point; instead, m* is delivered later after m is delivered

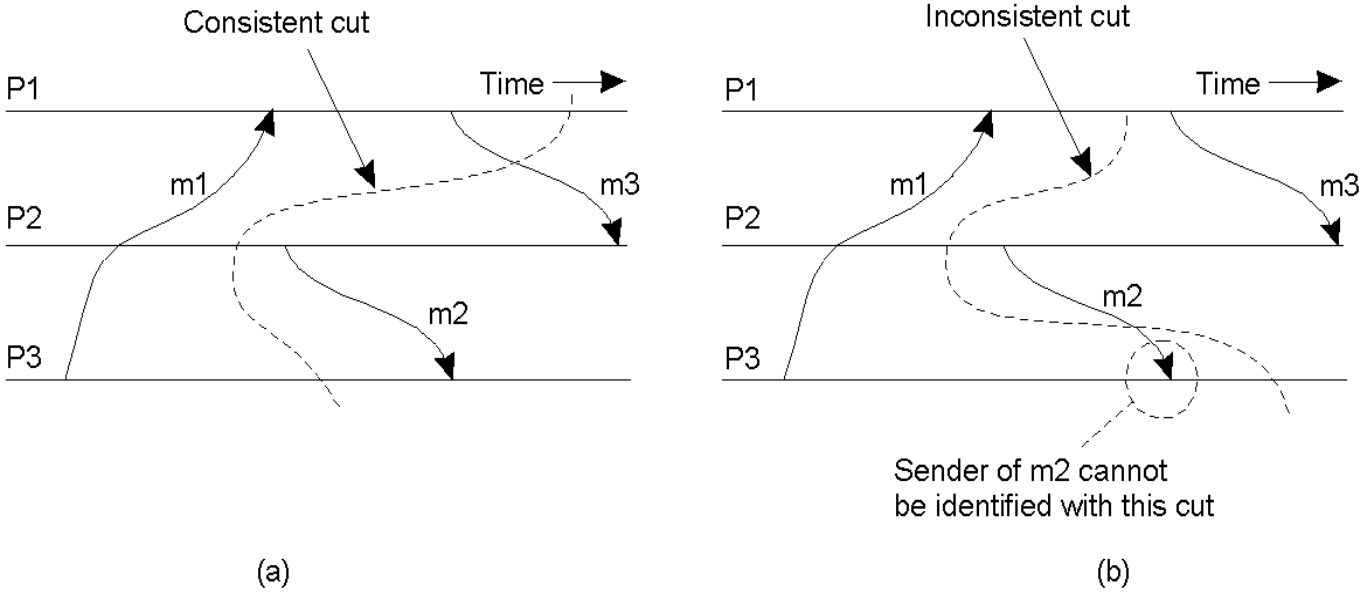


Distributed Snapshots

- Sometimes you want the current **global** state of a distributed computation, i.e., a distributed snapshot
- A **distributed snapshot** consists of all local states and messages in transit; it should reflect a **consistent** state

(a) Consistent cut: m3 is sent but not yet received

(b) Inconsistent cut: m2 is received but not yet sent

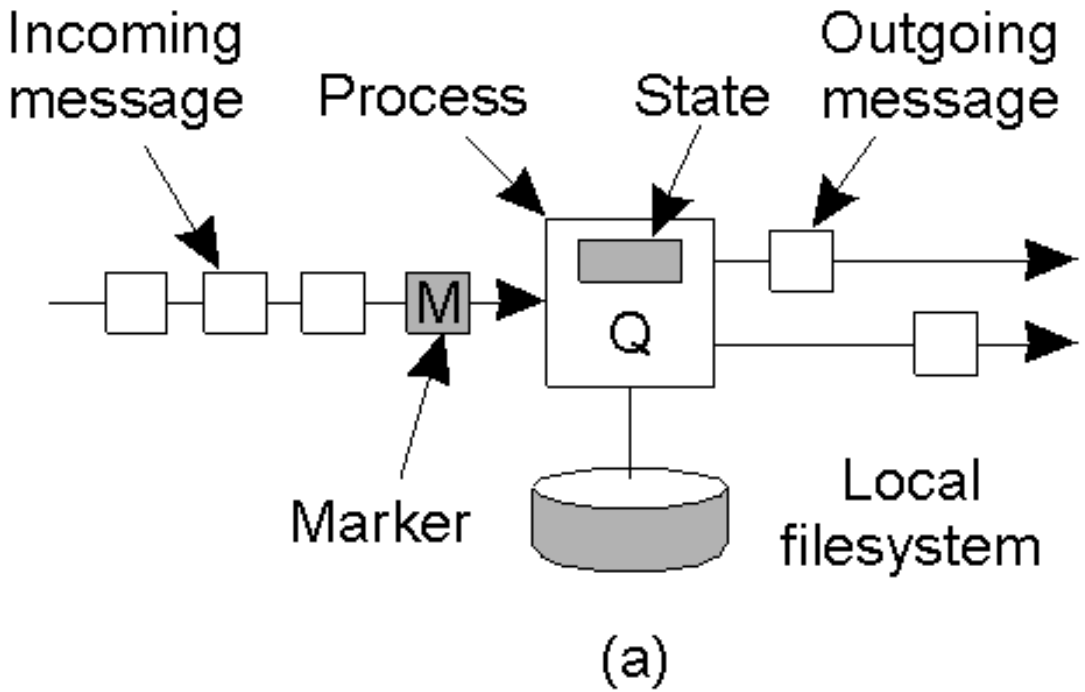


Distributed Snapshots

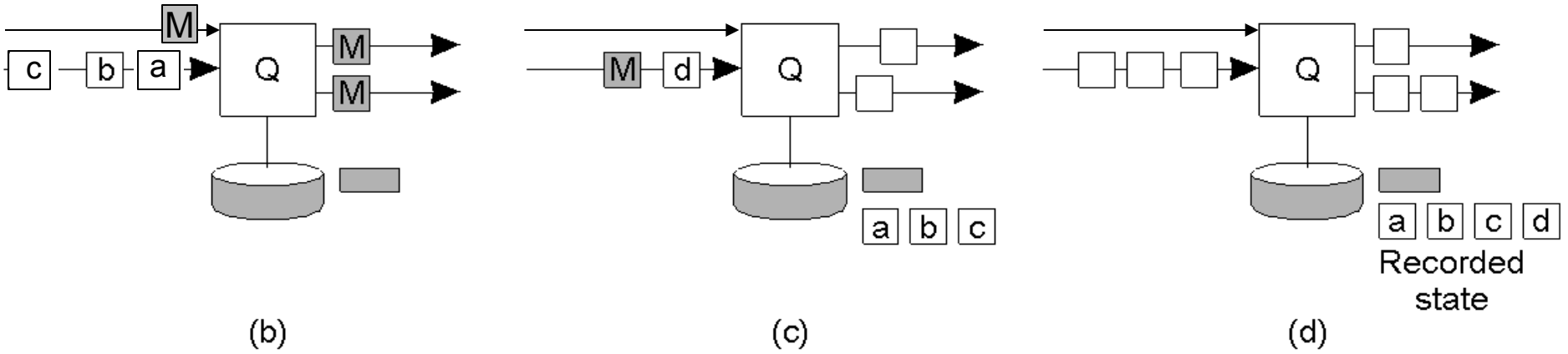
- Any process *P* can initiate taking a distributed snapshot
  - ***P* starts by recording its own local state**
  - ***P* sends a marker *M* on all of its outgoing channels**
- When process *Q* receives a marker *M* on channel *C*, its action depends on whether it already recorded its local state
  - **Not yet recorded: Process *Q* records its local state, and sends the marker *M* on each of its outgoing channels**
  - **Already recorded: Process *Q* records *C*'s channel state by recording all messages that *Q* received on channel *C* after it recorded its local state and before it received the marker *M* on channel *C***
- Process *Q* is finished when it has received a marker on each of its incoming channels

Distributed Snapshots

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



Distributed Snapshots



(b) Process *Q* receives a marker *M* on its first incoming channel, records its local state and sends *M* on all of its outgoing channels

(c) Process *Q* receives a marker *M* on its second incoming channel, and records messages *a*, *b, c* which it received on its second channel

(d) Process *Q* receives incoming message *d* on its second channel and finishes by recording message *d*