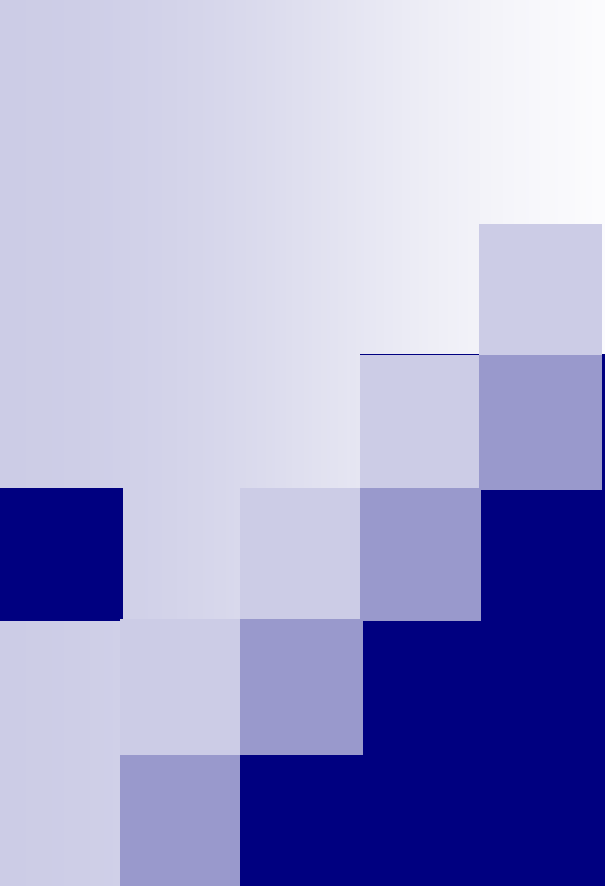




Distributed Systems

Lecture 6

Professor Louise E. Moser

Spring 2013

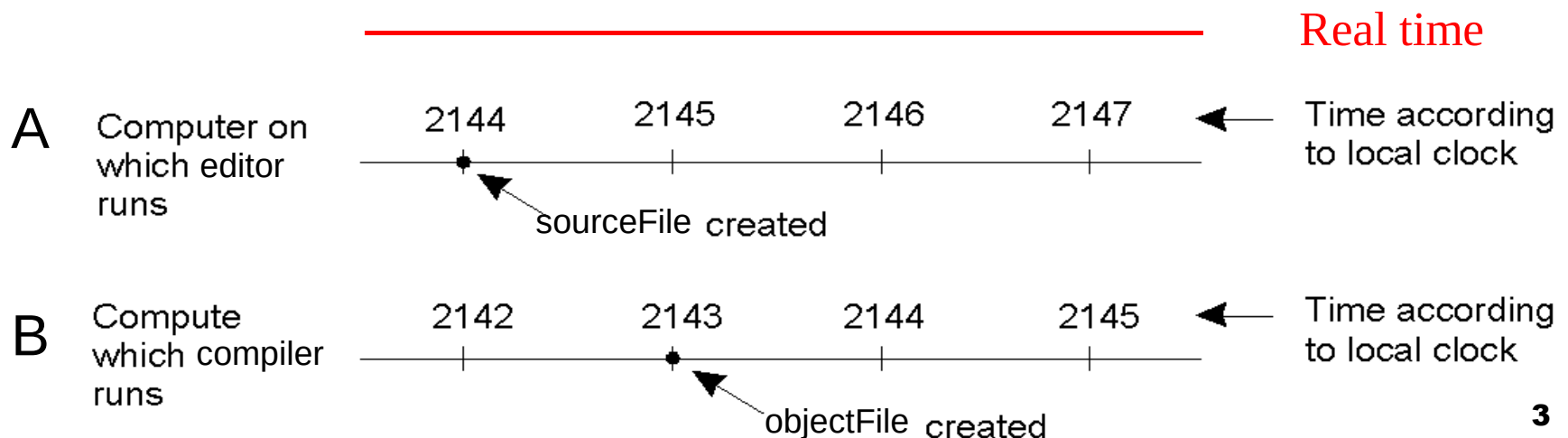


Clocks

- 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 compiling occurred before editing



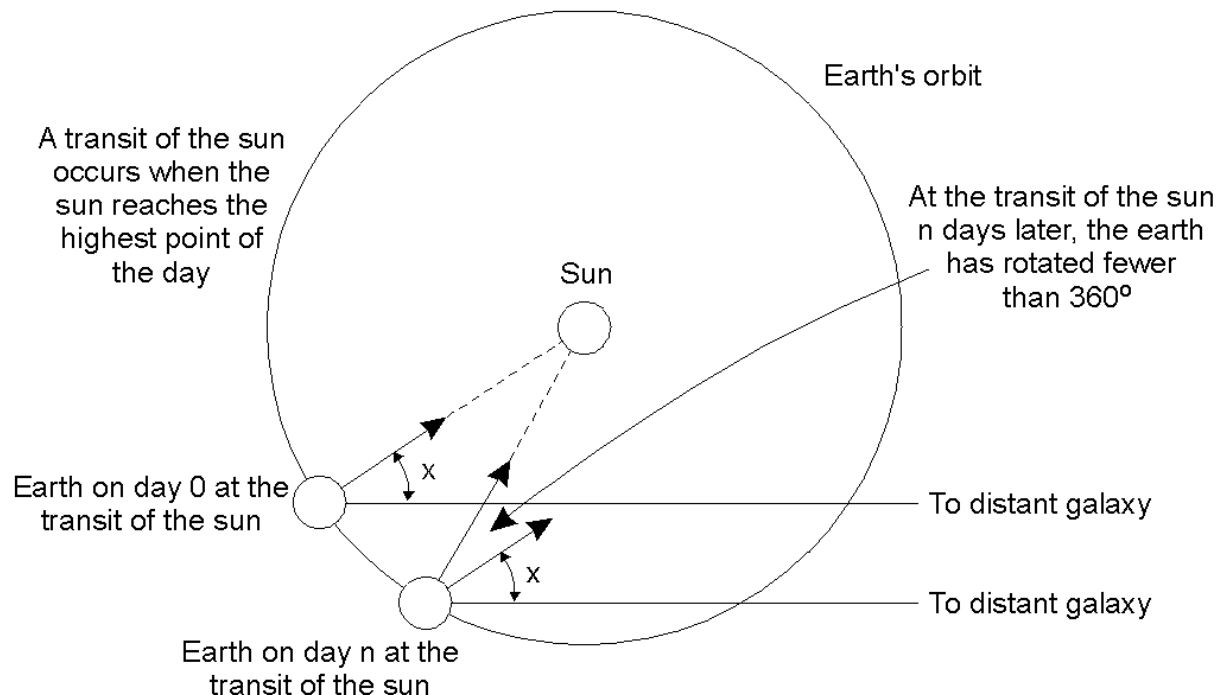
Physical Clocks

- **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 (quite accurate)
 - The real time is taken as the average of 50 cesium clocks around the world
 - Introduces 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 0.5 ms
- **Questions:** Does UTC address all of our problems? Don't we now have a global timing mechanism?

Physical Clocks

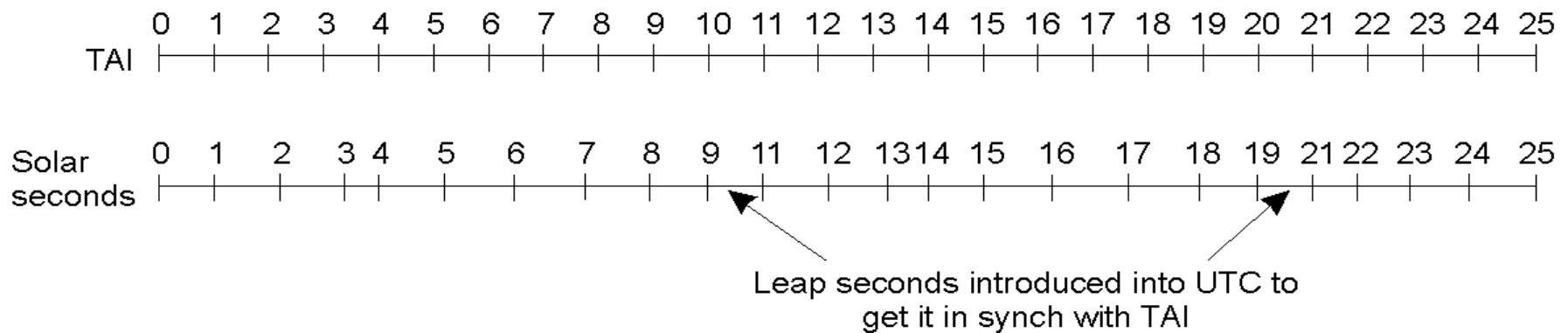
■ Computation of the mean solar day

- The solar day is slightly shorter (about 4 minutes) than the distant galaxy day (Andromeda day)
- One solar day corresponds to about 359° for rotation of the Earth about its axis and about 1° for 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)**
- **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 $C_p(t)$, where t is UTC time
 - **Ideally, we have that, for each processor p , $C_p(t) = t$ and, thus, $dC_p/dt = 1$**

Clock Synchronization Algorithms

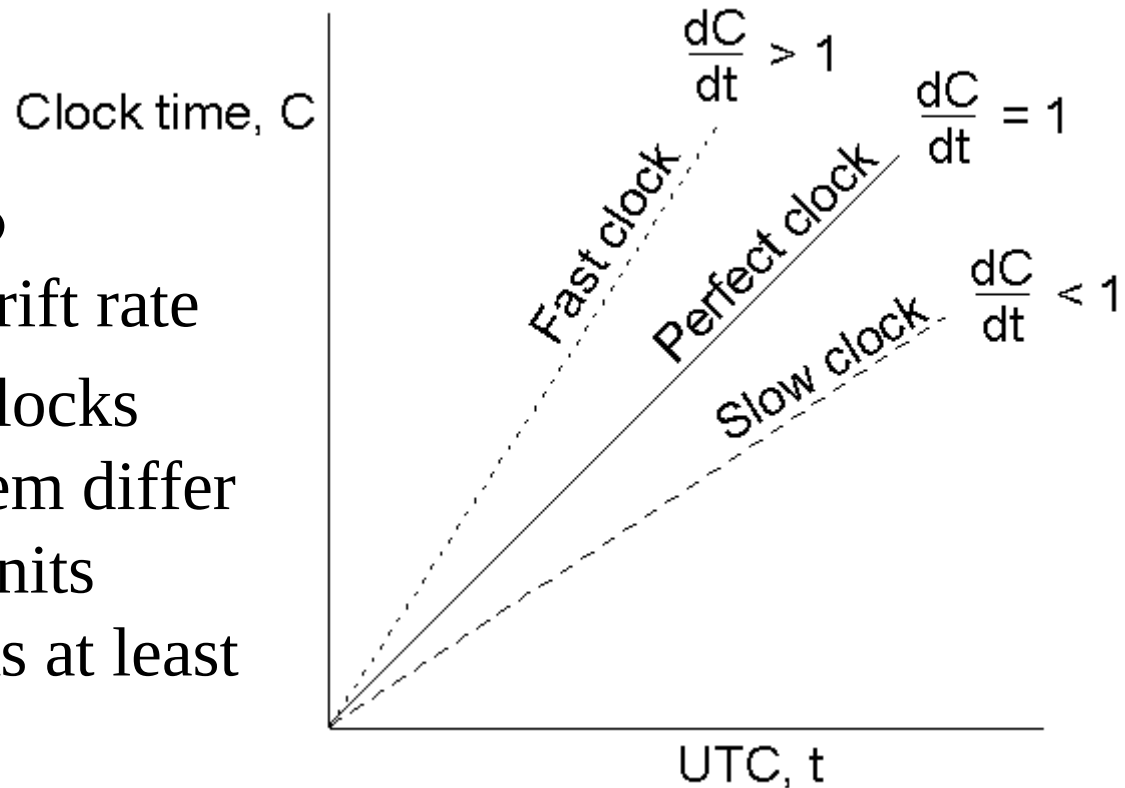
- The relation between clock time and UTC time when clocks tick at different rates

- In practice:

$$1 - \rho < dC/dt < 1 + \rho$$

where ρ is the clock drift rate

- **Goal:** Never let two clocks in the distributed system differ by more than δ time units
Synchronize the clocks at least every $\delta/(2\rho)$ seconds

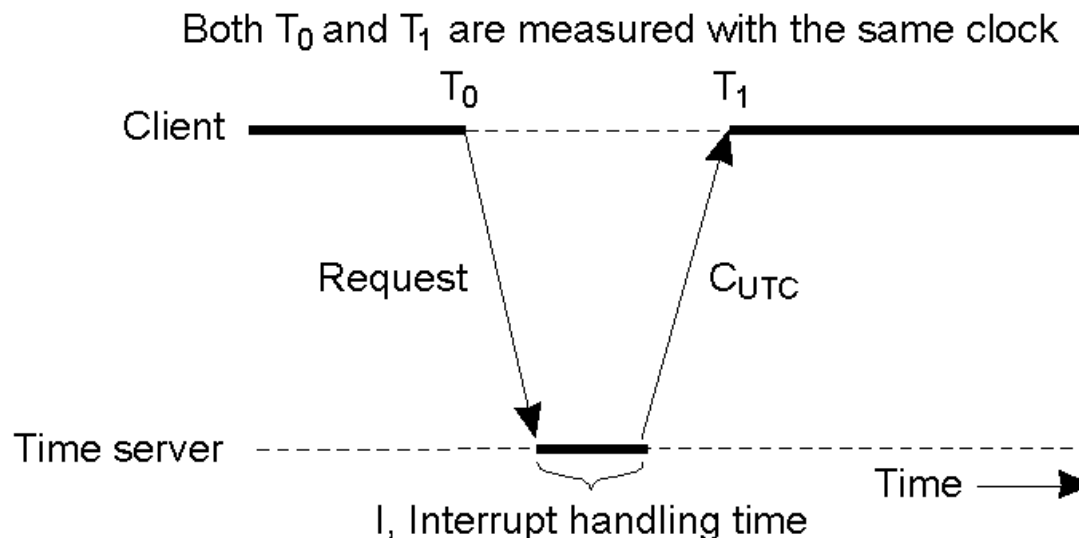


Clock Synchronization Principles

- **Principle I:** Each machine asks 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
- **Principle 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

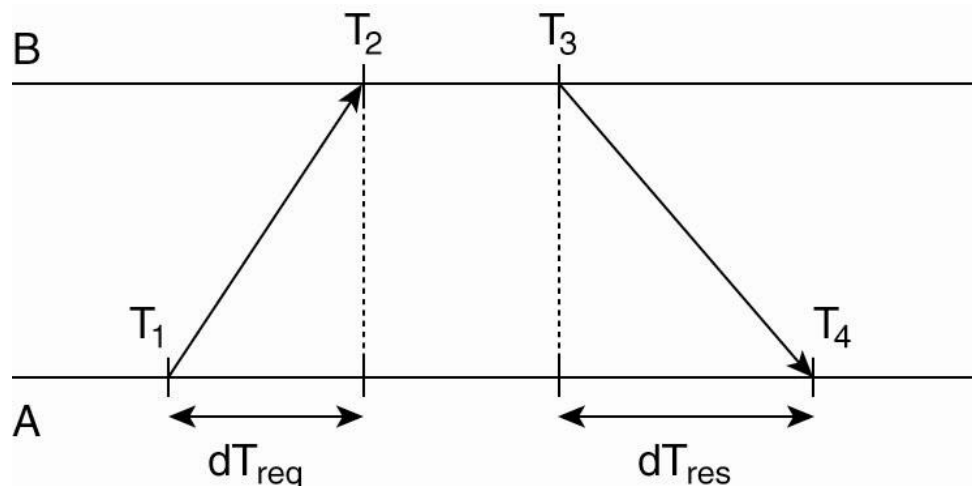
Cristian's Algorithm

- 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 with the smallest round-trip delay



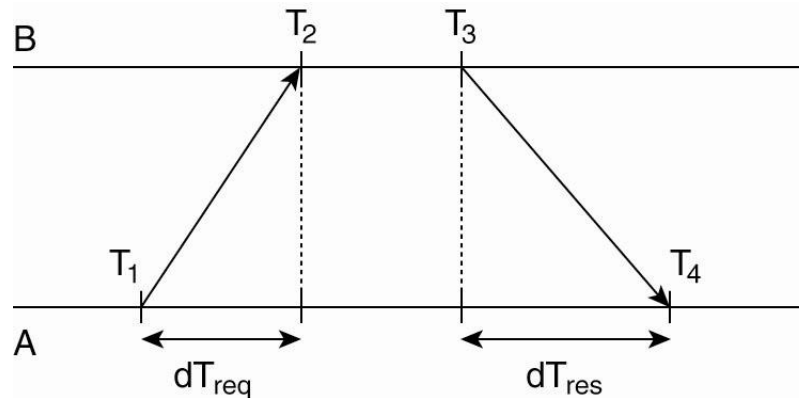
Network Time Protocol

- The Network Time Protocol (NTP) operates as follows:
- A requests the current time from the time server B
- B responds with T_2 and T_3
- A calculates an offset $\theta = ((T_2 - T_1) - (T_4 - T_3)) / 2$
- If $\theta < 0$, A would set its clock backward which is not allowed
- Thus, if $\theta < 0$, A must reduce its clock gradually by reducing its clock rate

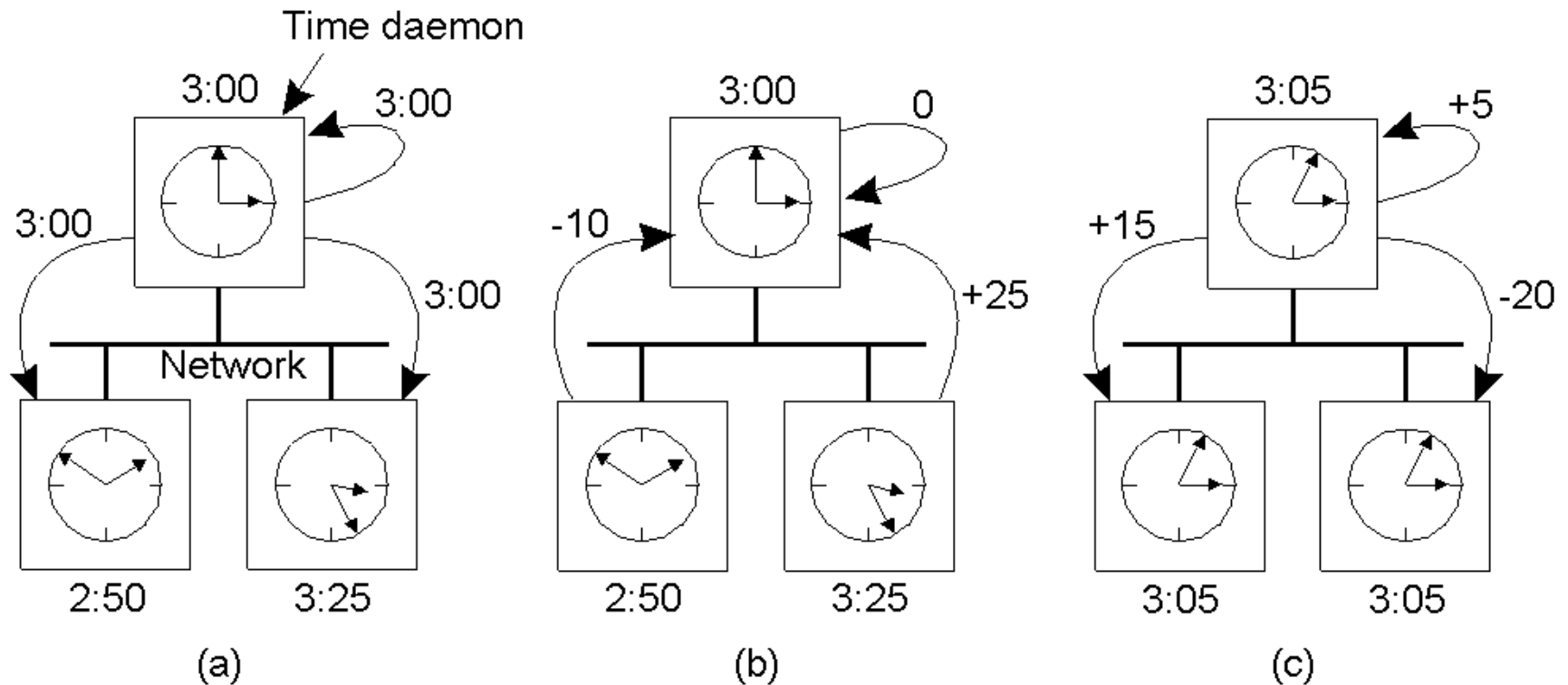


Network Time Protocol

- A calculates a delay $\delta = ((T4 - T1) - (T3 - T2)) / 2$
- Eight pairs (θ, δ) of offsets are calculated
 - **The minimum value of 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 answer
- (c) The time daemon tells every machine how to adjust its clock

The Happened-Before Relation

- **Problem:** First, we need to introduce a notion of ordering before we can order anything
- The **happened-before** relation 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 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 with the following properties:
 - P1:** If a and b are two events in the same process and $a \rightarrow b$, then $C(a) < C(b)$
 - P2:** If a represents the sending of a message m and b represents the receiving of the message m , then $C(a) < C(b)$
- **Questions:** How do we attach a timestamp to an event when there is no global clock? How do we maintain a **consistent** set of logical clocks, one per process?

Lamport's Algorithm

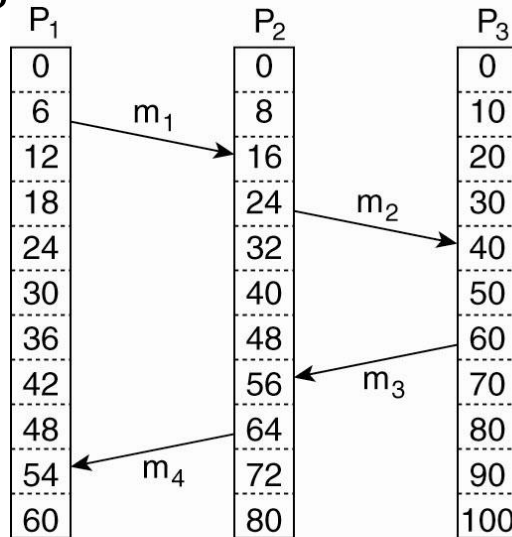
- Each process P_i maintains a local counter C_i and updates C_i according to the rules:
 - (1) On executing an event (including sending or receiving a message), P_i updates its local counter $C_i \leftarrow C_i + 1$
 - (2) When process P_i sends a message m , process P_i updates its local counter $C_i \leftarrow C_i + 1$ and sets message m 's timestamp T_m to C_i
 - (3) When process P_j receives a message m with timestamp T_m , process P_j updates its local counter $C_j \leftarrow \max(C_j, T_m) + 1$
 - (4) Process P_j then delivers the message to the application (not necessarily in causal or total order)
- Property **P1** is satisfied by (1)
- Property **P2** is satisfied by (2) and (3)

Lamport's Algorithm

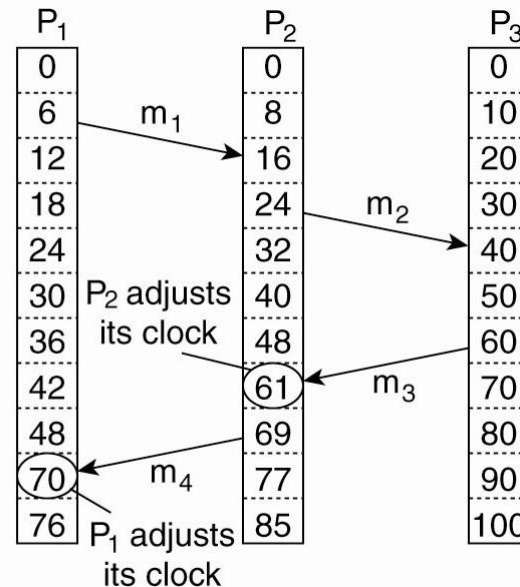
(a) Logical clocks of three processes, with events occurring at different rates in the different processes

- 6 is after m₁ is sent, 16 is after m₁ is received and adding 1, etc
- Rule (3) of Lamport's algorithm is NOT applied

(b) Logical clocks of three processes, using Lamport's algorithm including Rule (3)



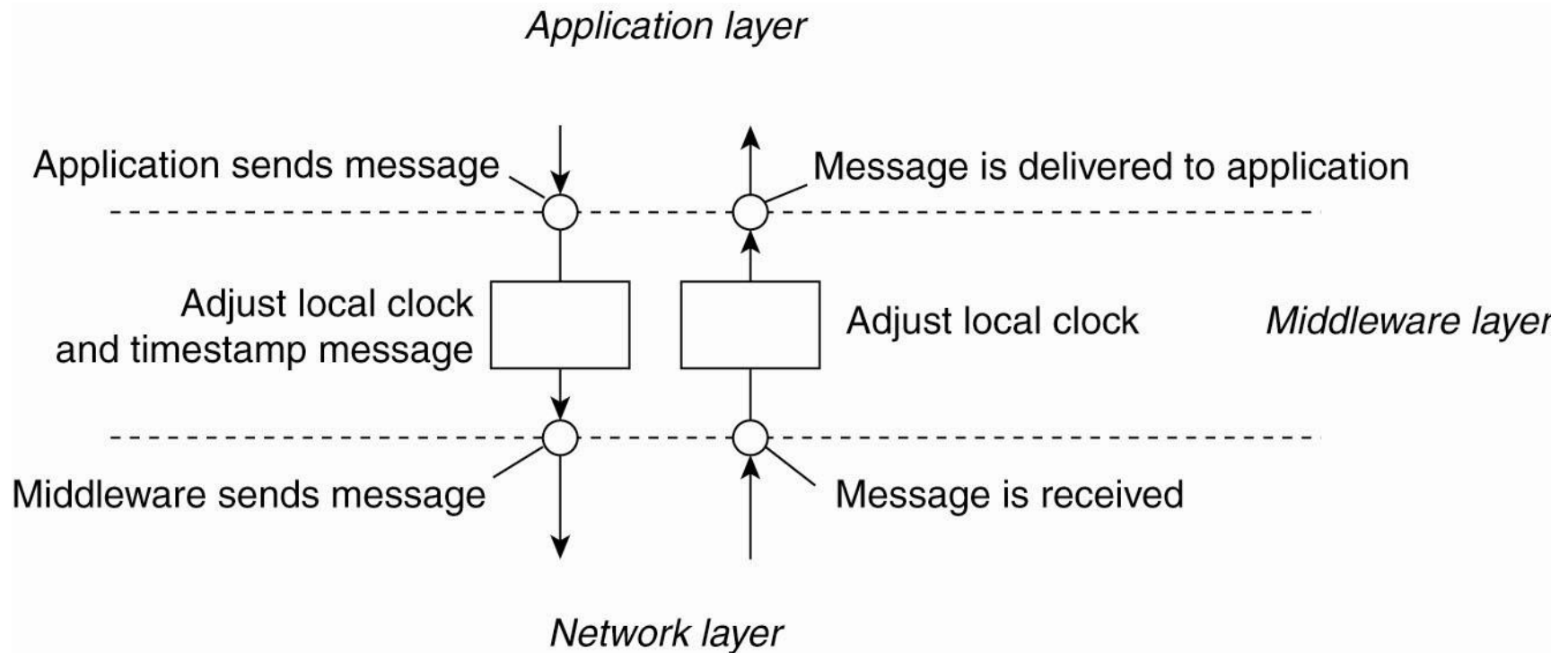
(a)



(b)

Lamport's Logical Clocks

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

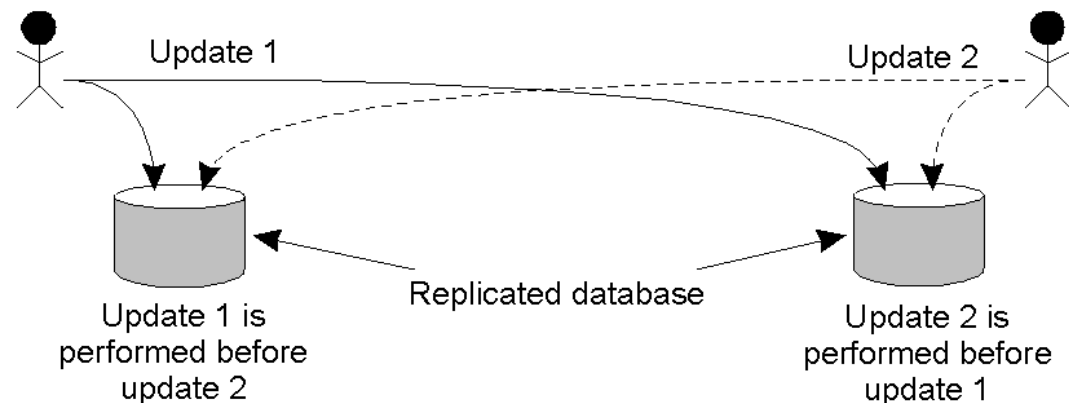


Vector Timestamps

- Lamport 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 P_i has an vector $VC_i[1..n]$, where $VC_i[j]$ denotes the number of events that process P_i knows have taken place at process P_j
- (1) Before executing an event, process P_i updates $VC_i[i] \leftarrow VC_i[i] + 1$
 - (2) When process P_i sends a message m , it
 - Updates $VC_i[i] \leftarrow VC_i[i] + 1$, and
 - Sends VC_i as a vector timestamp $vt(m)$ with message m
 - (3) When a process P_j delivers a message m from process P_i with vector timestamp $vt(m)$, it
 - Updates $VC_j[k] \leftarrow \max(VC_j[k], vt(m)[k])$ for all k , and
 - Updates $VC_j[j] \leftarrow VC_j[j] + 1$

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 *P1* adds \$100 to an account (initial value \$1000)
 - (2) Process *P2* 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 $\$1111 = (1000+100) \times 1.01$
 - Replica R2 ends up with $\$1110 = (1000 \times 1.01) + 100$

Total Ordering with Logical Clocks

- **Problem:** It can still occur that two events in different processes happen at the same time
- **Solution:** To avoid this problem, process P_i attaches its process number i to event e , that is:

P_i timestamps event e with $C_i(e)$ and i

- Then a before b if and only if
 - (1) $C_i(a) < C_j(b)$, or
 - (2) $C_i(a) = C_j(b)$ and $i < j$

Totally Ordered Multicast

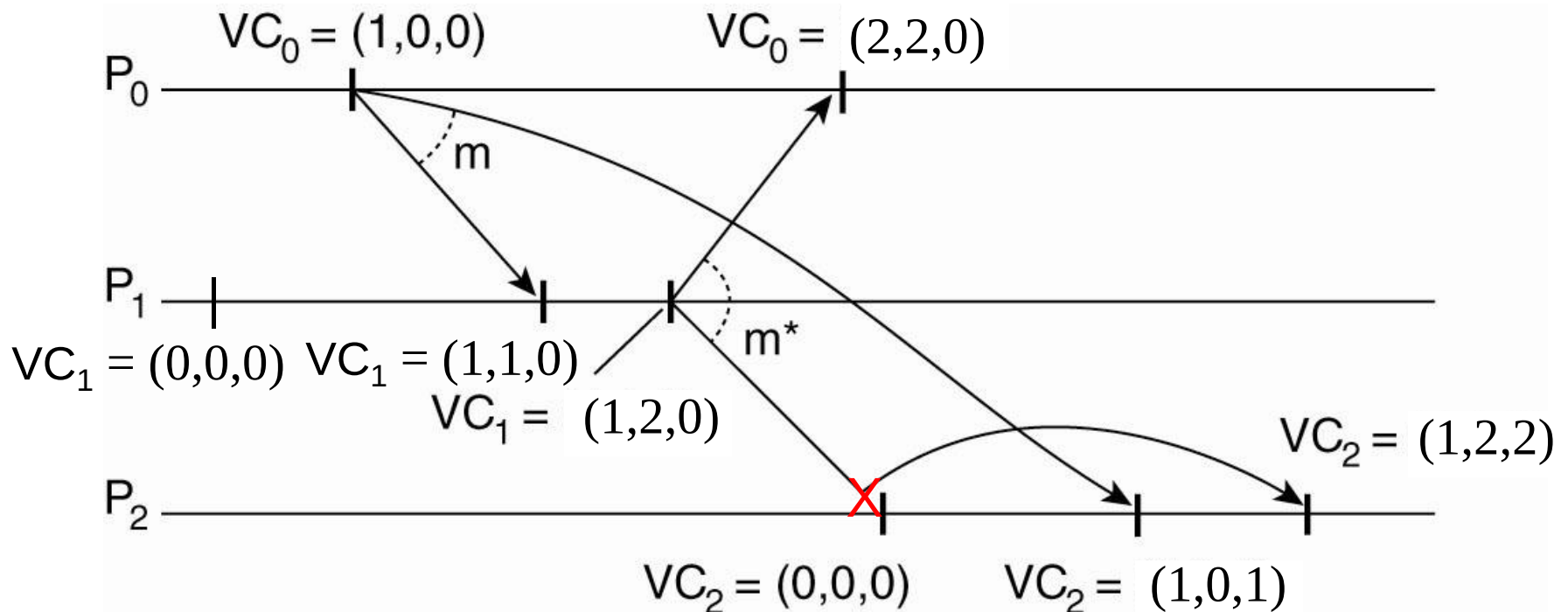
- Process P_i increments its **logical clock** and sends a message m_i with the **timestamp** to all the other processes and puts the message m_i into its local queue q_i
- Process P_j puts the incoming message m_i into its local queue q_j ordered according to the **timestamp** of m_i
- Process P_j updates its **logical clock** and delivers the message m_i to the application iff:
 - (1) Message m_i is at the head of its queue q_j , and
 - (2) For each process P_k , $k \neq i$, there is a message m_k in its queue q_j with a larger **timestamp**
- **Note:** We are assuming that communication is reliable and FIFO ordered

Causally Ordered Multicast

- Process P_i increments its vector clock and sends a message m_i with a vector timestamp to all the other processes and puts the message m_i into its local queue q_i
- Process P_j puts the incoming message m_i from process P_i into its local queue q_i for process P_i according to the timestamp $vt(m_i)[i]$
- Process P_j updates its vector clock and delivers the message m_i to the application iff:
 - (1) Message m_i is at the head of the queue q_i , and
 - (2) For each process P_k , $k \neq i$ **and** $k \neq j$, (there is a **delivered** message m_k in queue q_k with a larger timestamp, i.e., $vt(m_k)[k] \geq vt(m_i)[k]$) **or** $vt(m_i)[k]=0$
- **Note:** Again, we are assuming that communication is reliable and FIFO ordered

Causal Ordering

- Enforcing causal order using vector timestamps

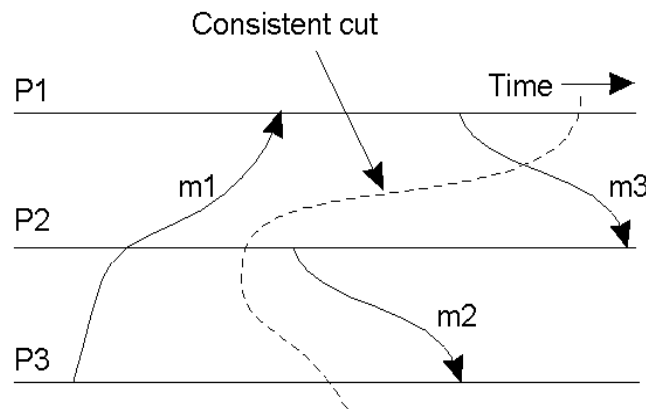


Distributed Snapshots

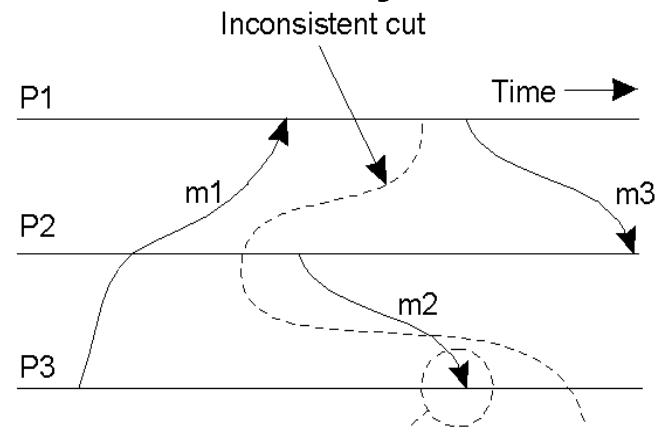
- Sometimes you want the current **global** state of a distributed computation, i.e., a distributed snapshot
- A **distributed snapshot** consists of all local states and messages in transit; it should reflect a **consistent** state

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

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



(a)



Sender of m2 cannot
be identified with this cut

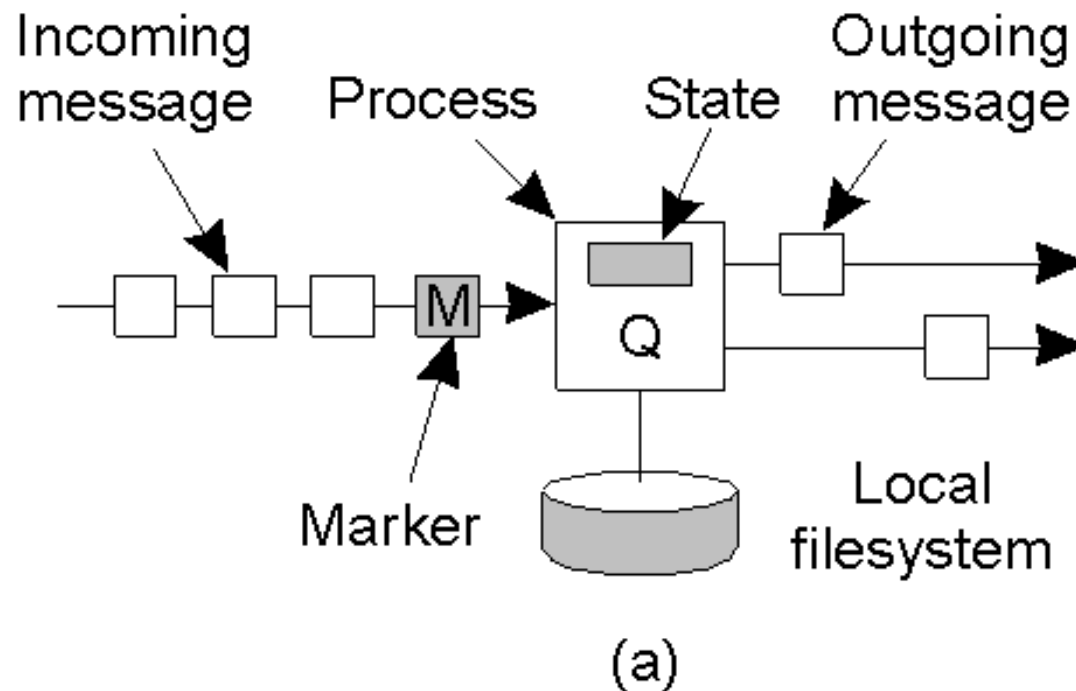
(b)

Distributed Snapshots

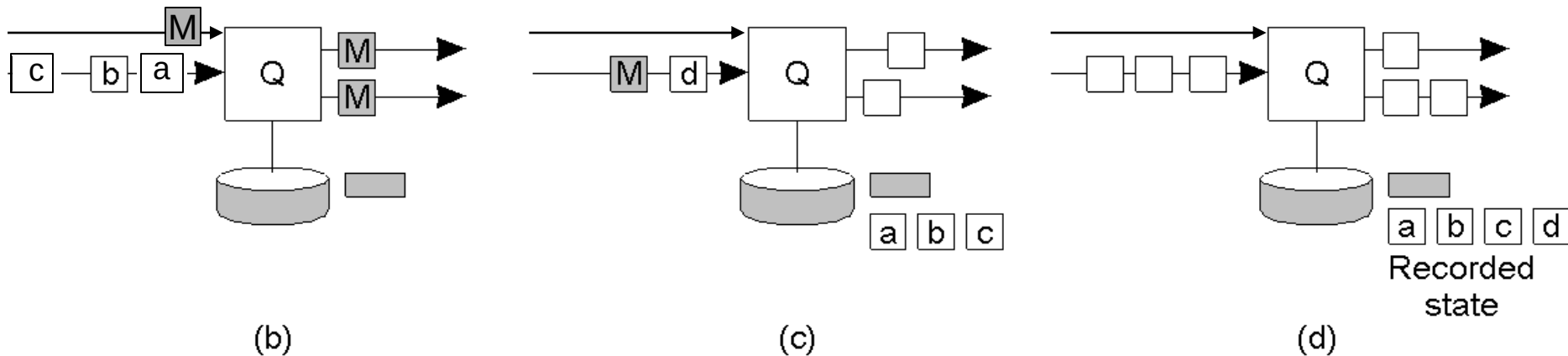
- Any process P can initiate taking a distributed snapshot
 - P starts by recording its own local state
 - P sends a marker M along each of its outgoing channels
- When process Q receives a marker M through channel C , its action depends on whether it already recorded its local state:
 - **Not yet recorded:**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

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

(d) Process *Q* receives incoming message *d* and marker *M* on its second channel and finishes recording the state of the second channel