DISTRIBUTED SYSTEMS
Principles and Paradigms
Second Edition
ANDREW S. TANENBAUM
MAARTEN VAN STEEN

Chapter 6
Synchronization

Overview

Synchronization of distributed processes is more difficult than synchronization of processes in uni/multi-processor systems.

Several issues that we will examine are:

- notion of global time: absolute time versus relative time
- election algorithms: for a electing a coordinator on-the-fly
- distributed mutual exclusion

Computer Clocks

- A computer *timer* is a quartz crystal that oscillates at a well defined frequency. Each oscillation decrements a *counter* by one. When the counter goes down to zero, an interrupt is generated and the counter is reloaded from a *holding register*. Each interrupt is called a *clock tick* (and can be set to interrupt certain number of times per second)
- The time is stored on a battery-backed CMOS RAM. At every clock tick, the interrupt service procedure adds one to the stored time
- With one computer even if the time is off it is usually not a problem. With n computers, all n crystals will run at slightly different rates, causing the software clocks to gradually get out of sync. This difference in time values is called *clock skew*

Clock Skew

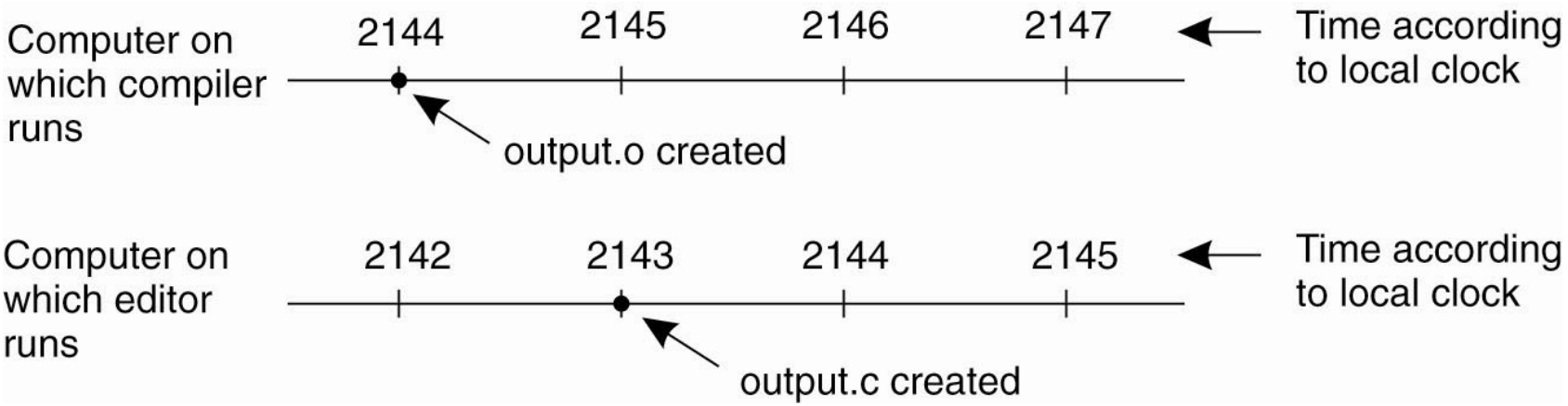

Figure 6-1. When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

Physical Clocks (1)

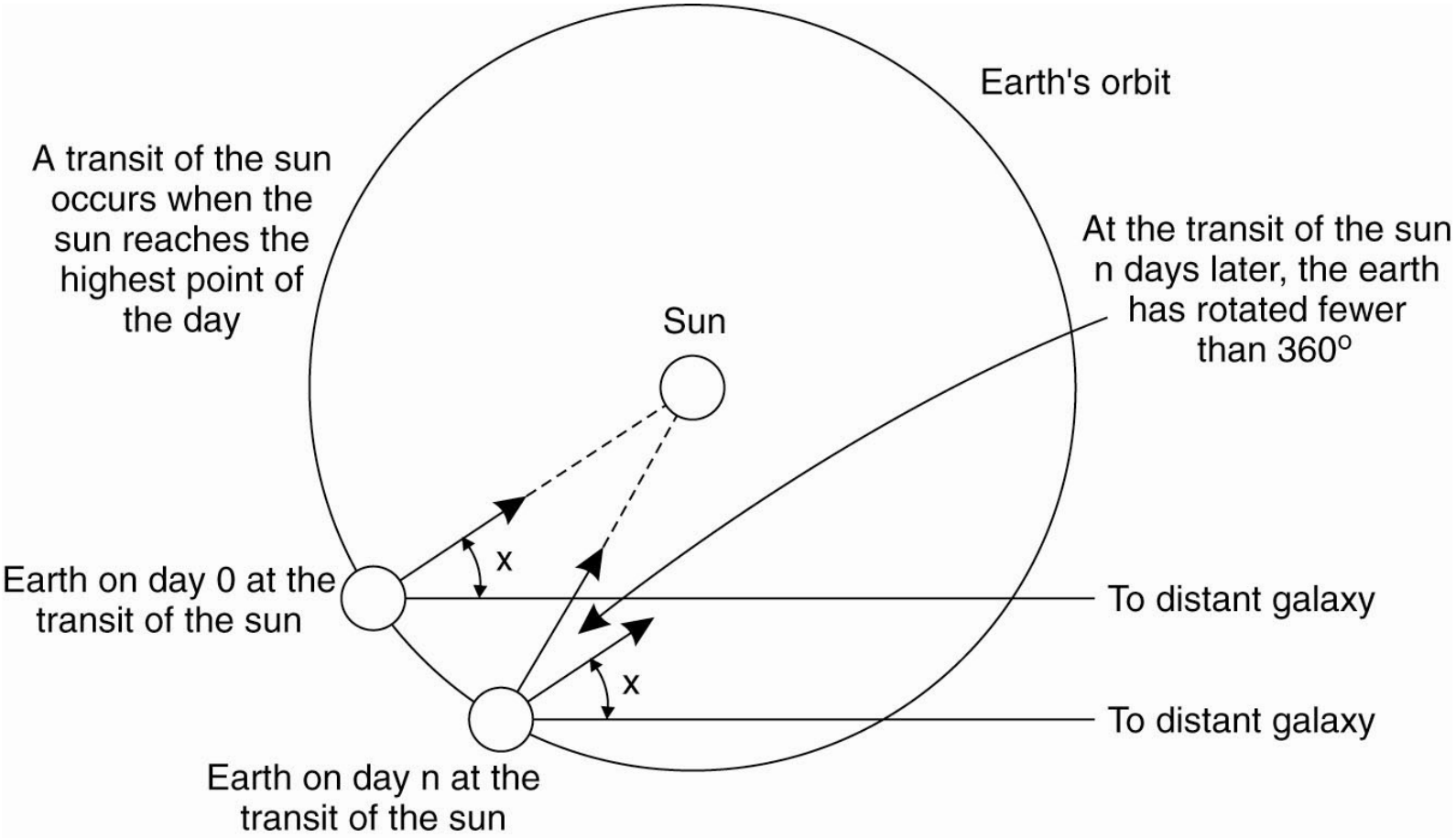

Figure 6-2. Computation of the mean solar day.

Real Time

- Measure a large number of days and average them and then divide by 86400 to obtain **mean solar second**. However, this value is changing
- Atomic time: one second = time taken for Cesium 133 atom to make 9,192,631,770 transitions. The **International Atomic Time (TAI)** is the average of about 50 Cesium clocks around the world. TAI is the mean number of ticks of the Cesium 133 atom since midnight, 1st Jan, 1958 reported by the Bureau International de l'Heure in Paris
- A leap second is introduced whenever the discrepancy between TAI and solar time grows to 800 msec. About 30 leap seconds have been introduced since 1958. This is known as **Universal Coordinated Time (UTC)**
- UTC is broadcast over short-wave by NIST on station WWV (and from satellites). Check http://www.nist.gov for more info

Physical Clocks (2)

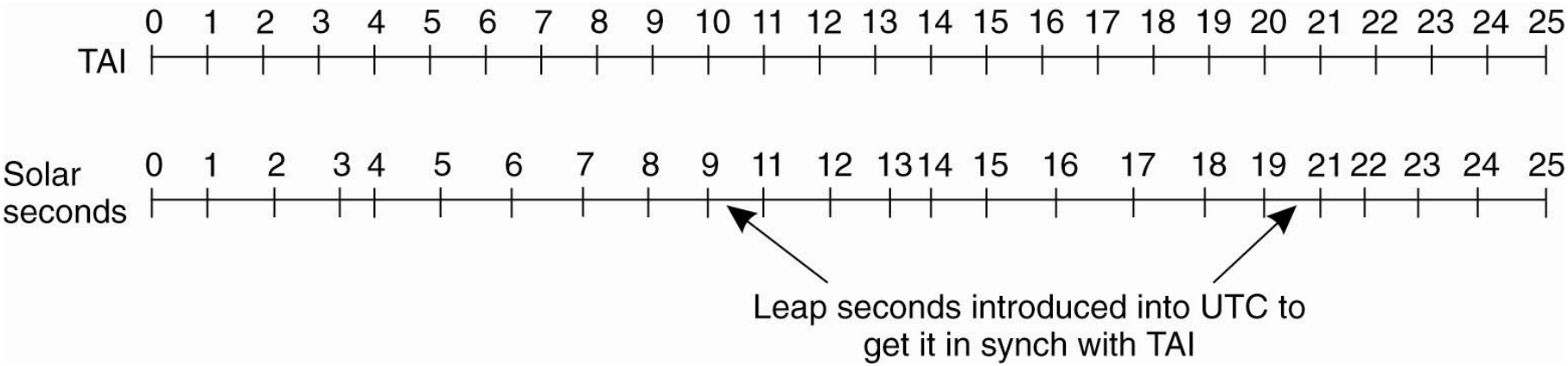

Figure 6-3. TAI seconds are of constant length, unlike solar seconds. Leap seconds are introduced when necessary to keep in phase with the sun.

Global Positioning System (1)

- 29 Satellites at an orbit of 20,000 km above Earth
- Each Satellite has four atomic clocks, which are regularly calibrated from Earth
- Each satellite continuously broadcasts its position and timestamps its message with its local time
- A GPS receiver can compute its own position using three satellites, assuming that the receiver has accurate time. Otherwise it requires four satellites
- Typical accuracy is 1-5m but can be as good as less than one foot

Global Positioning System (2)

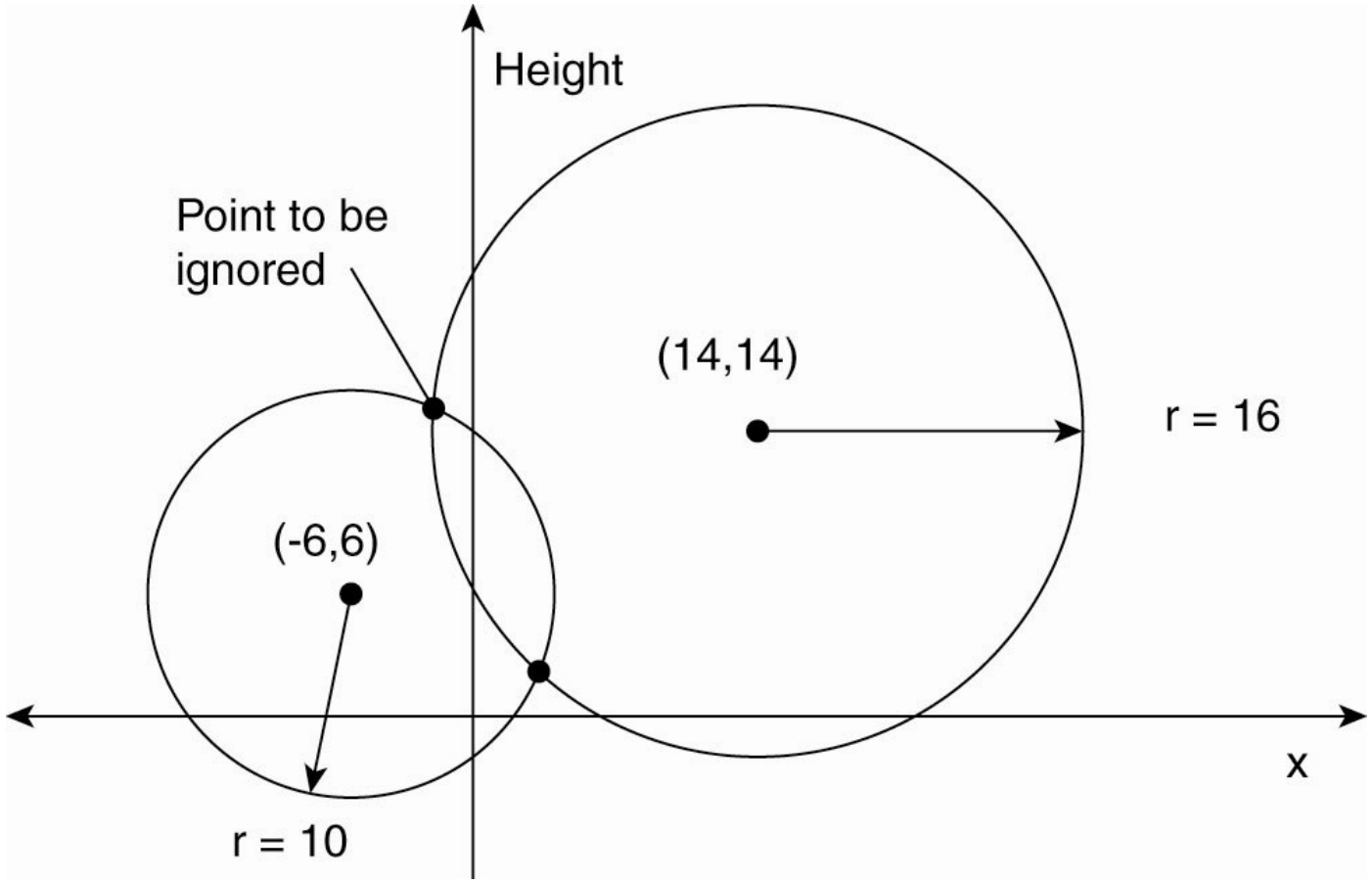

Figure 6-4. Computing a position in a two-dimensional space.

Global Positioning System (3)

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 sync with that of a satellite

Clock Synchronization Algorithms

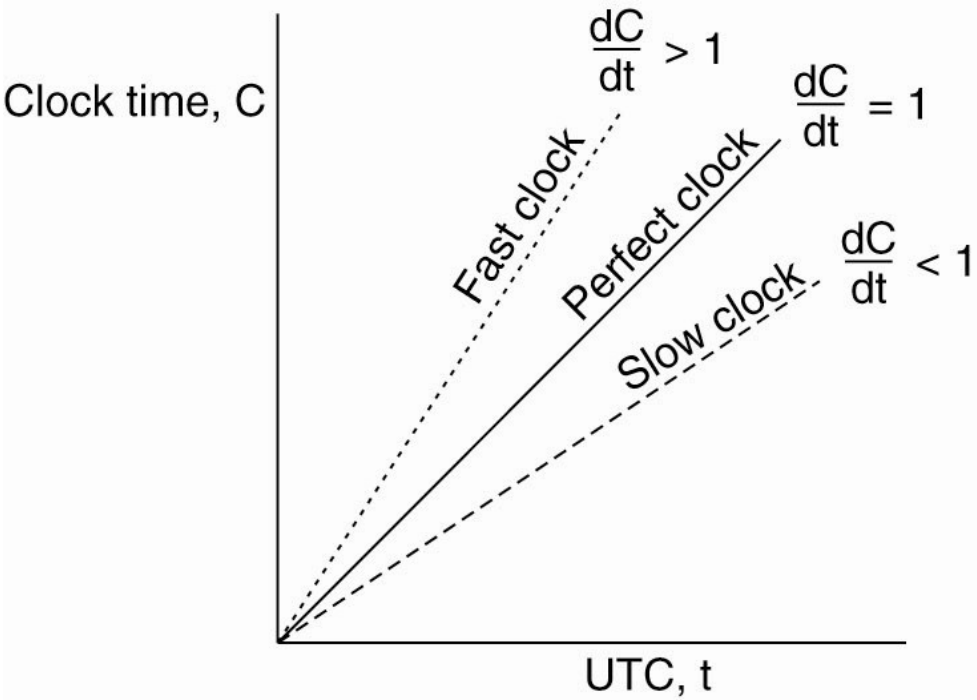

Figure 6-5. The relation between clock time and UTC when clocks tick at different rates. The relation between clock time and UTC when clocks tick at different rates. Let maximum drift rate be ρ. Then $1 - \rho \leq dC/dt \leq 1 + \rho$ where dC/dt is the rate of drift of the clock relative to UTC. Ideally, we want dC/dt to be 1.To ensure two clocks never differ more than δ, the clocks must be synchronized at least every δ/2 ρ seconds.

Network Time Protocol (1)

- Can achieve worldwide accuracy in the range 1-50 msec. Widely used on the Internet
- Uses combination of various advanced clock synchronization algorithms (RFC1305)
- Uses a distributed shortest paths algorithm to determine who gets served by whom. Has mechanisms for dealing gracefully with servers being down
- Clients need to slow down or speed up local clocks to sync up gradually with a server

Network Time Protocol (2)

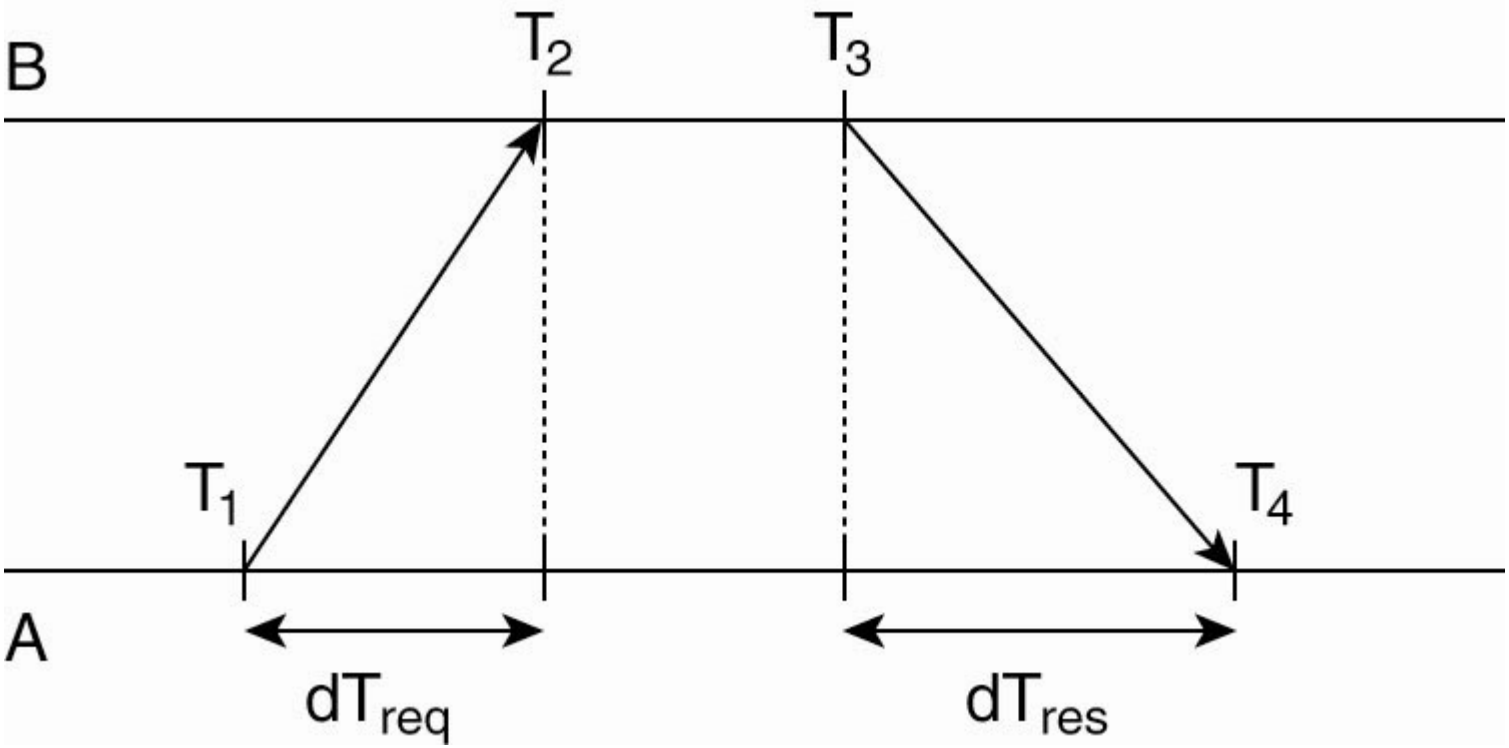

Figure 6-6. Getting the current time from a time server. Relative offset $\theta = T_3 - ((T_2 - T_1) + (T_4 - T_3))/ 2$

Network Time Protocol (3)

- NTP can be setup pair-wise between servers. Both servers ask each other for time and calculate the θ and δ, where $\delta = ((T_2 - T_1) + (T_4 - T_3))/2$
- Eight pairs of θ and δ are buffered and the minimal value is taken as the delay between the servers
- A server with a *reference clock* such as a WWV receiver or an atomic clock is a *stratum-1 server*. When A contacts B it will only adjusts its clock if its stratum number if higher than B. Moreover, after the synchronization, A's stratum level becomes one more than B's level

The Berkeley Algorithm (1)

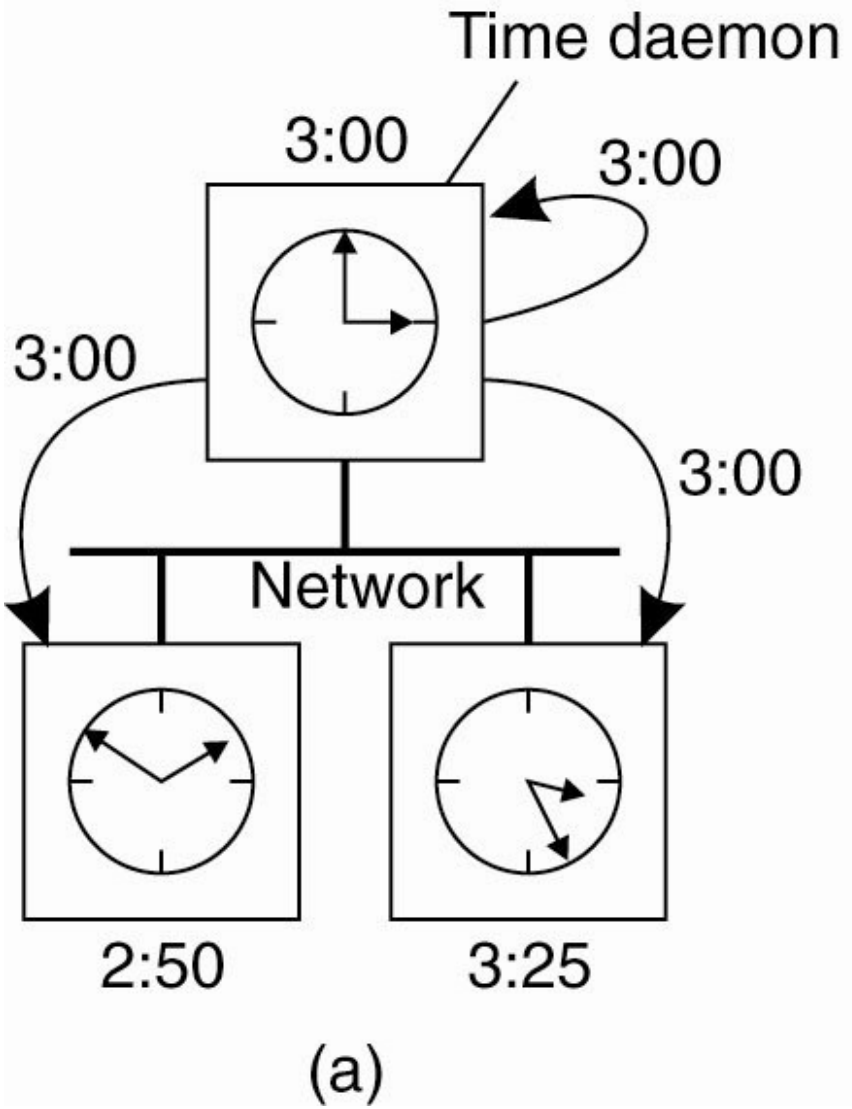

Figure 6-7. (a) The time daemon asks all the other machines for their clock values.

The Berkeley Algorithm (2)

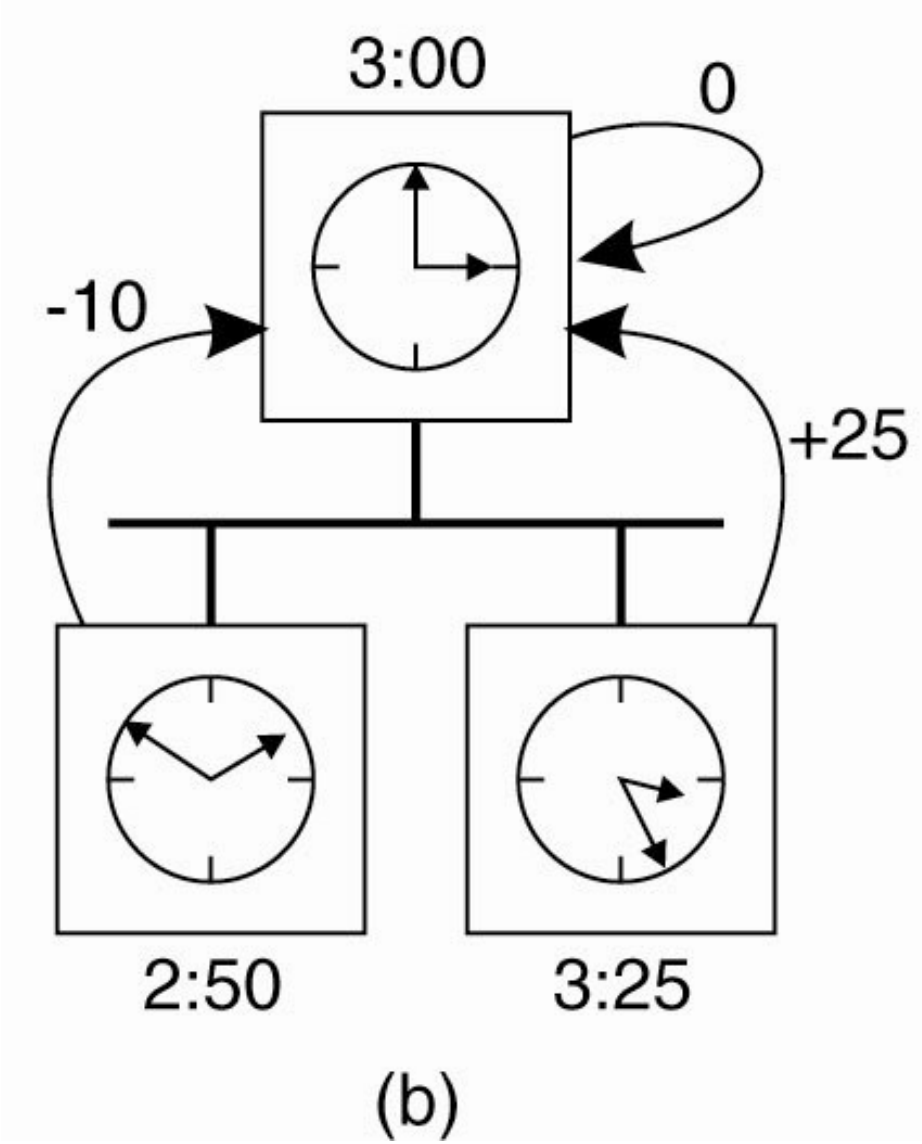

Figure 6-7.

(b) The machines answer.

The Berkeley Algorithm (3)

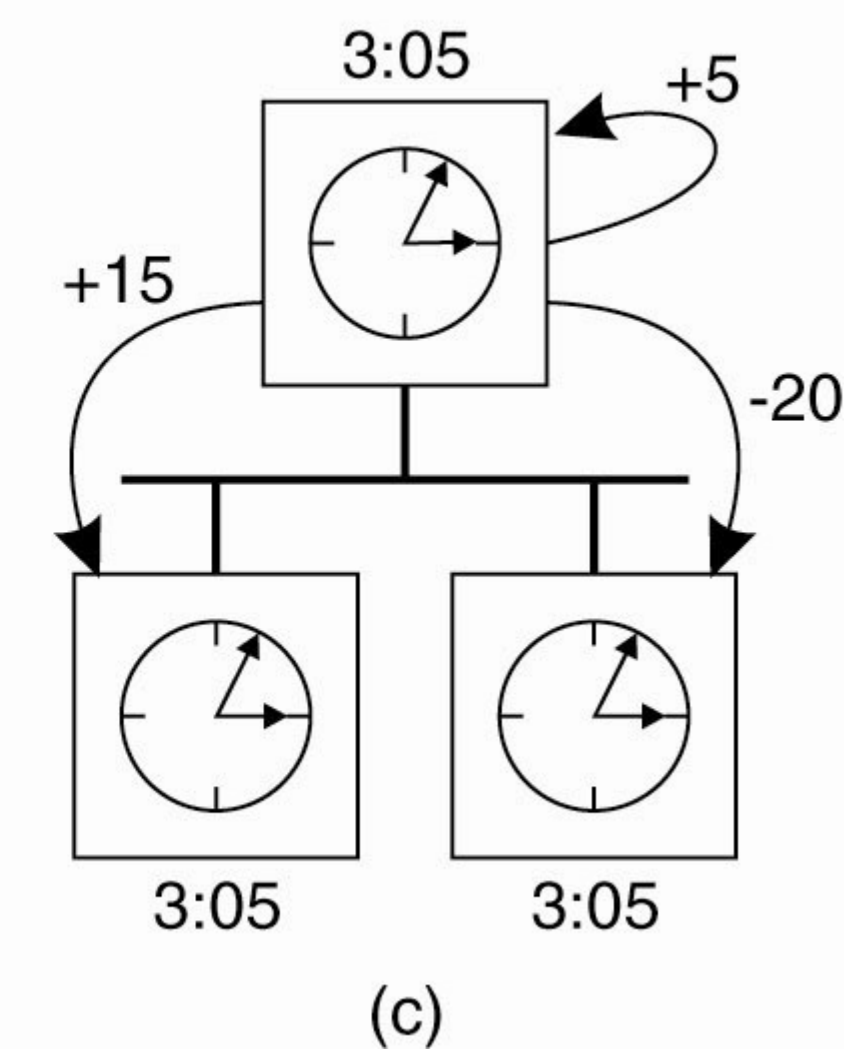

Figure 6-7. (c) The time daemon tells everyone how to adjust their clock.

Lamport's Logical Clocks (1)

- For many purposes, it is sufficient that machine agree on the same time even though that time may not agree with real world.

- If two process do not interact, it is not necessary that their clocks be synchronized. Furthermore, if all processes agree on the order in which events occur, then they need not agree on the time.

Lamport's Logical Clocks (2)

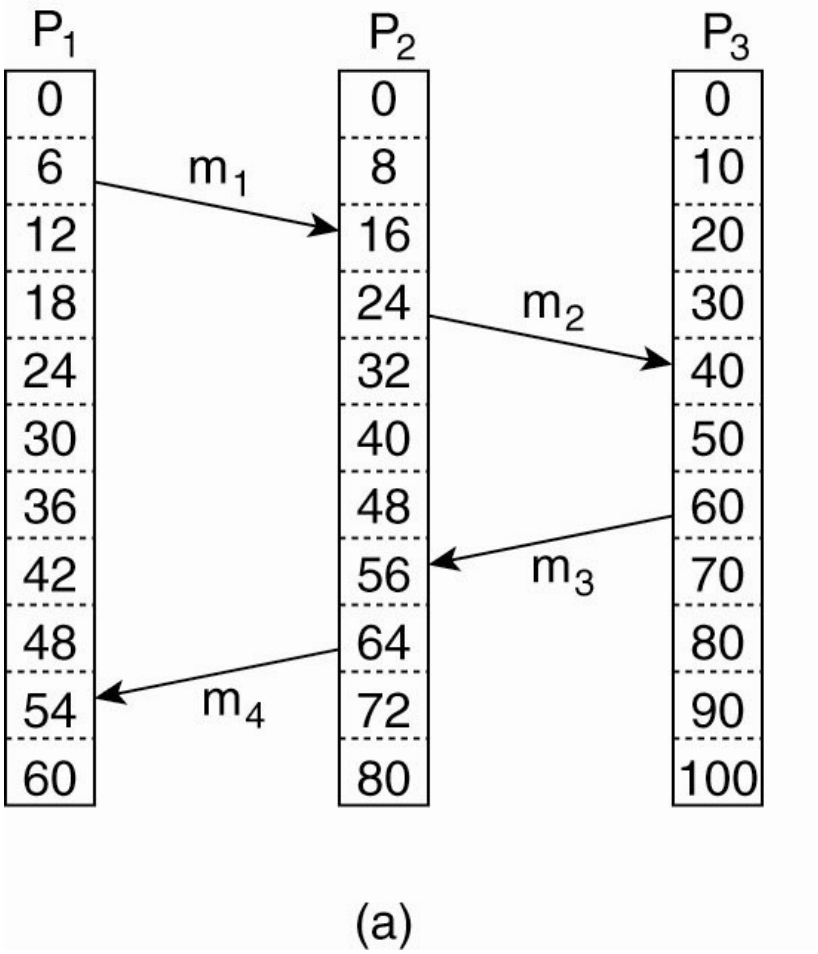

Figure 6-9. (a) Three processes, each with its own clock. The clocks run at different rates.

Lamport's Logical Clocks (3)

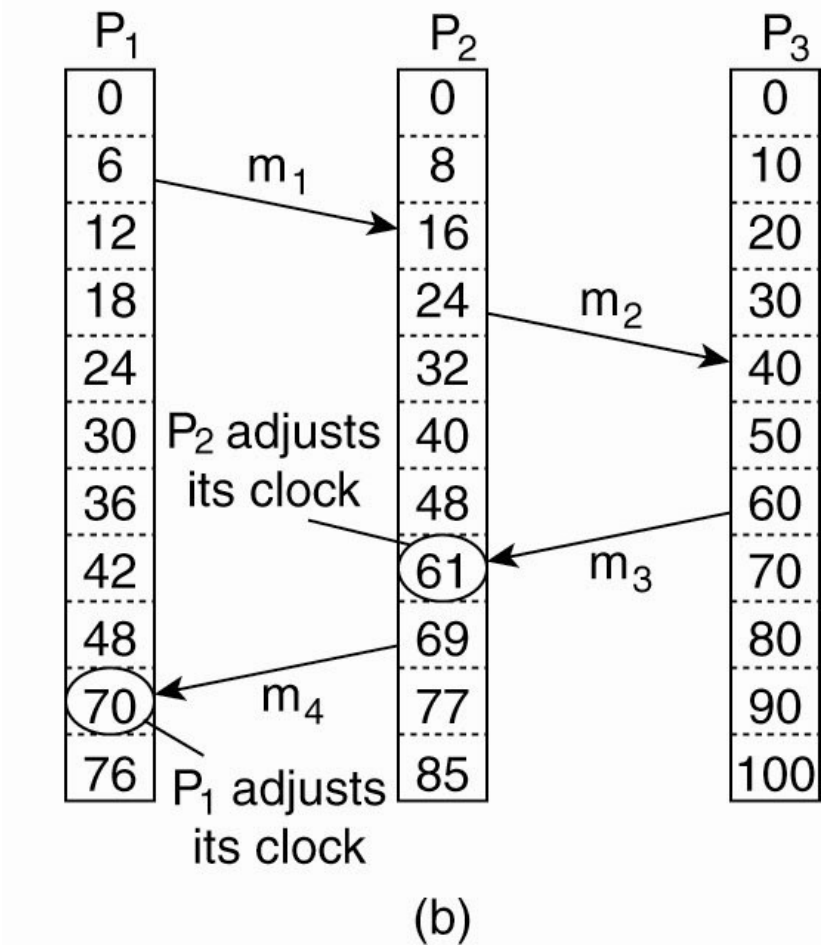

Figure 6-9. (b) Lamport's algorithm corrects the clocks.

Lamport's Logical Clocks (4)

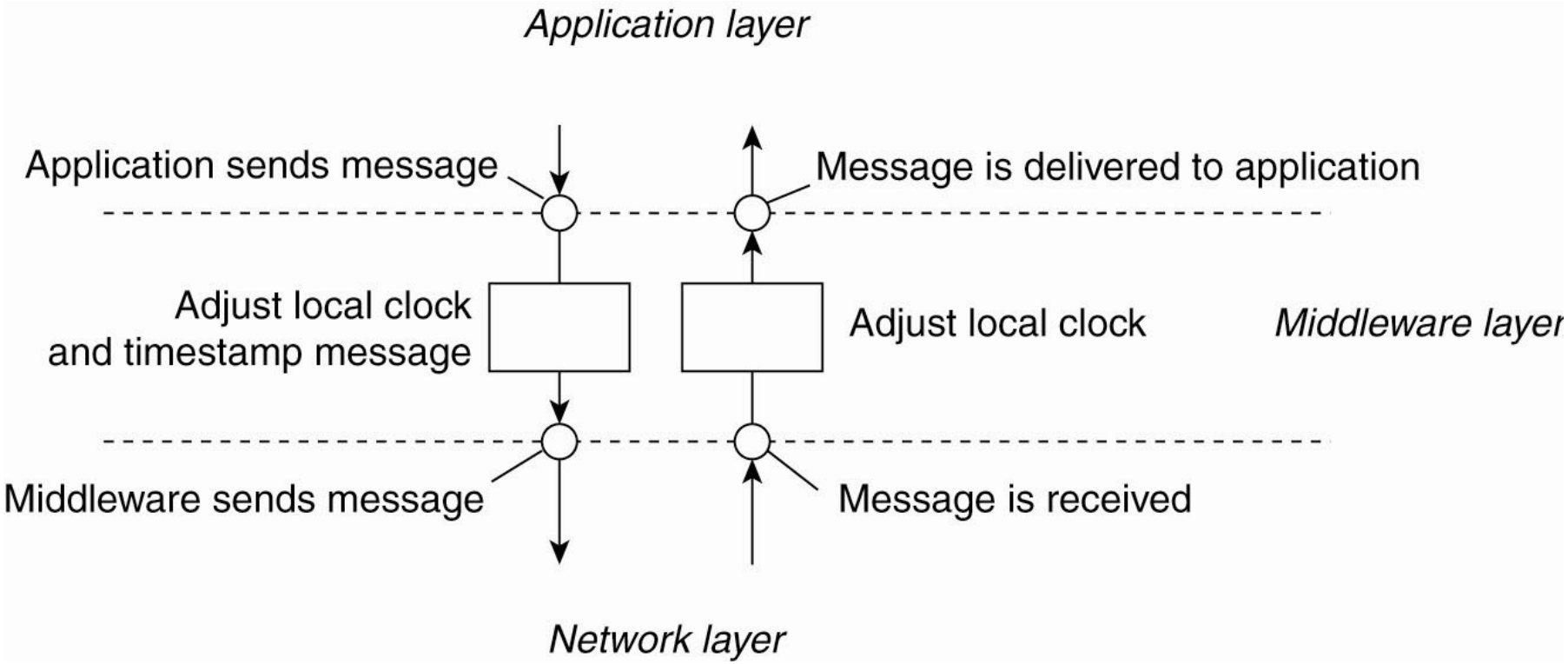

Figure 6-10. The positioning of Lamport's logical clocks in distributed systems.

Lamport's Logical Clocks (5)

C is the timestamp function that is defined as follows:

1) If *a happens before b in the same process, C(a) < C(b)*
2) If *a represents sending and b receiving of a message, then C(a) < C(b)*
3) *For all distinct events a and b,C(a) ≠ C(b)*

- The time is always adjusted forward. Each message carries the sending time according to the sender's clock. If receiver's time is prior to the sending time, the receiver fast forwards its clock to 1 more than the sending time
- In addition, between every two events the clock must tick at least once
- No two events ever occur at exactly the same time. Tag process ids to low bits of time to make time be unique

Lamport's Logical Clocks (6)

Updating counter C_i for process P_i

- Before executing an event P_i executes
  $C_i \leftarrow C_i + 1$
- When process P_i sends a message m to P_j, it sets *m*'s timestamp *ts (m)* equal to C_i after having executed the previous step
- Upon the receipt of a message *m*, process P_j adjusts its own local counter as
  $C_j \leftarrow \max\{C_j , ts(m)\}$,
  after which it then executes the first step and delivers the message to the application

Example: Totally Ordered Multicasting

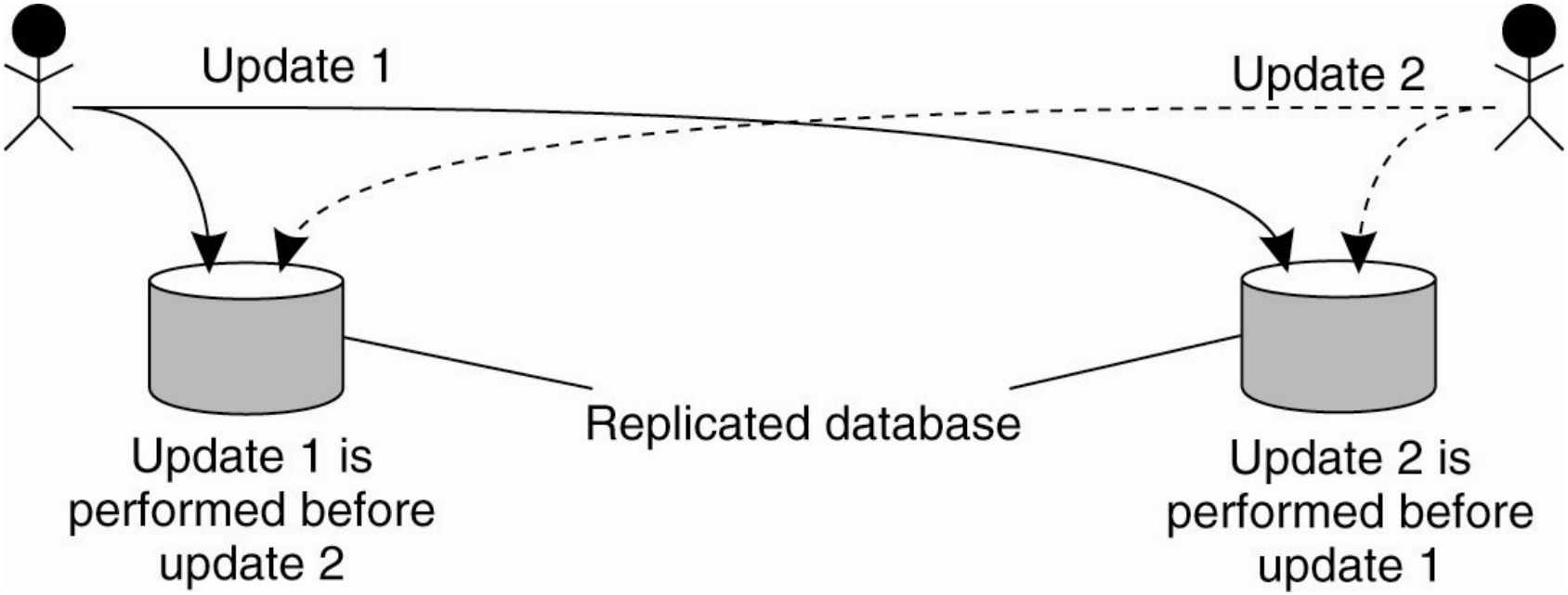

Figure 6-11. Updating a replicated database and leaving it in an inconsistent state.

Implementing Totally Order Multicasts

- *Totally Ordered Multicast*: A multicast operation by which all messages are delivered in the same order to each receiver. Can be implemented using Lamport's logical clock algorithm
- Each message is time-stamped with the current (logical) time of the sender. Messages from one receiver are ordered and messages aren't lost
- A process puts received messages into a queue ordered by timestamps. It acknowledges the messages with a multicast to all other processes. Eventually the local queues are the same at all processes
- A process can deliver a queued message to an application only if it is at the head of queue and has been acknowledged by each other process

Vector Clocks (1)

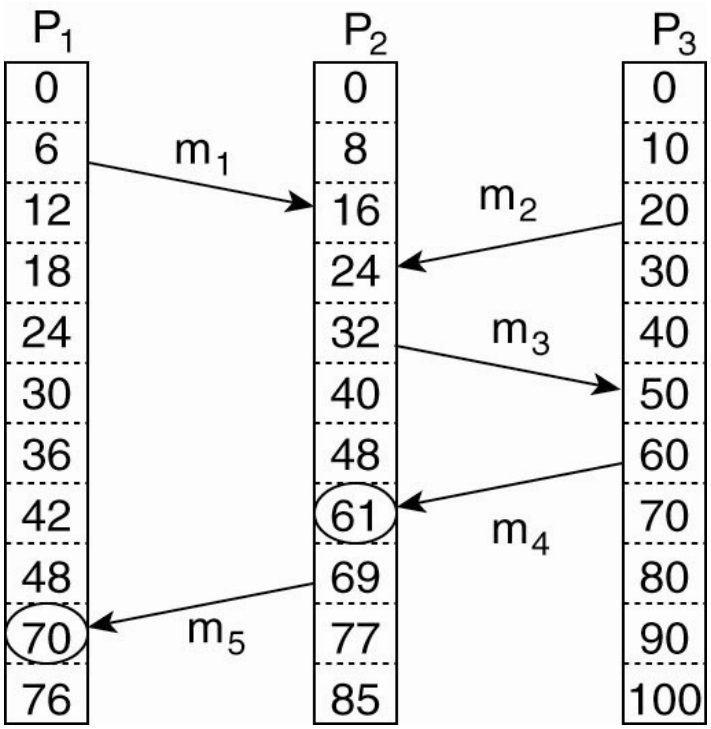

Figure 6-12. Concurrent message transmission using Lamport logical clocks.

Knowing that m_1 was received before m_2 doesn't tell us if they are connected.

Lamport clocks do not capture **causality**. We need **vector clocks** to capture causality.

Vector Clocks (2)

Vector clock $VC(a)$ assigned to an event a has the property that if $VC(a) < VC(b)$ for some event b, then event a is known to causally precede event b.

Vector clocks are constructed by letting each process P_i maintain a vector VC_i with the following two properties:

1. $VC_i[i]$ is the number of events that have occurred so far at P_i. In other words, $VC_i[i]$ is the local logical clock at process P_i
2. If $VC_i[j] = k$ then P_i knows that k events have occurred at P_j. It is thus P_i's knowledge of the local time at P_j

Vector Clocks (3)

Step 2 is carried out by piggybacking vectors along with messages. The details are shown below:

- Before an event (send/receive or internal event), P_i executes $VC_i[i] \leftarrow VC_i[i] + 1$
- When process P_i sends a message m to P_j, it sets m's (vector) timestamp $ts(m)$ equal to VC_i after having executed the previous step
- Upon the receipt of a message m, process P_j adjusts its own vector by setting:

  $VC_j[k] \leftarrow \max\{VC_j[k], ts(m)[k]\}$ for each k,

  after which it executes the first step and delivers the message to the application

Enforcing Causal Communication

- We want to ensure that a message is delivered only if all messages that casually precede it have been delivered. We assume that the message are multicast within the group. This is known as *causally-ordered multicasting*
- Clocks are adjusted only when sending or receiving a message
- Then if P_j receives a message m from P_i with vector timestamp $ts(m)$, the delivery is delayed until the following conditions are met:
  - $ts(m)[i] = VC_j[i] + 1$
    (m is the next process P_j was expecting from P_i)
  - $ts(m)[k] \leq VC_j[k]$ for all $k \neq i$
    (P_j has seen all the messages that have been seen by P_i when it sent message m)

Enforcing Causal Communication

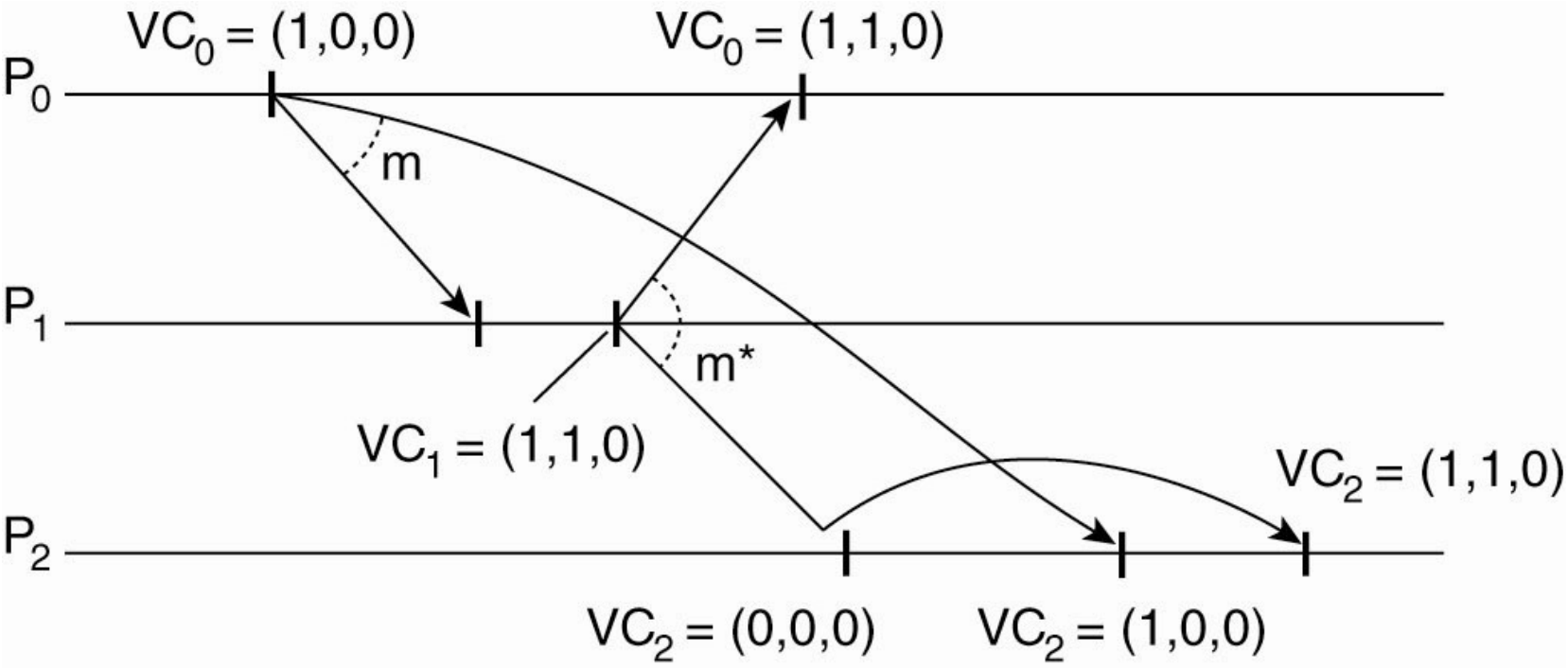

Figure 6-13. Enforcing causal ordered multicasting

Whose problem is it?

- Middleware deals with message ordering:
  - The middleware cannot tell what the message contains so only potential causality is captured
  - Two messages sent by the same process are always marked as causally related
- Middleware cannot be aware of external communication. Ordering issues can be adequately solved by looking at the application for which the communication is taking place. This is known as the *end-to-end argument*

Distributed Mutual Exclusion

- *Token-based solutions:* Based on passing a special message between the processes know as a *token*. There is only one token available and the process that has it can enter the critical section. It avoids starvation and deadlocks. But what if the token gets lost?
- *Permission-based approach:* A process wanting to enter the critical section first requires permission of other processes. Some ways of getting the permission:
  - Centralized algorithm
  - Decentralized algorithm
  - Distributed algorithm

Mutual Exclusion
A Centralized Algorithm (1)

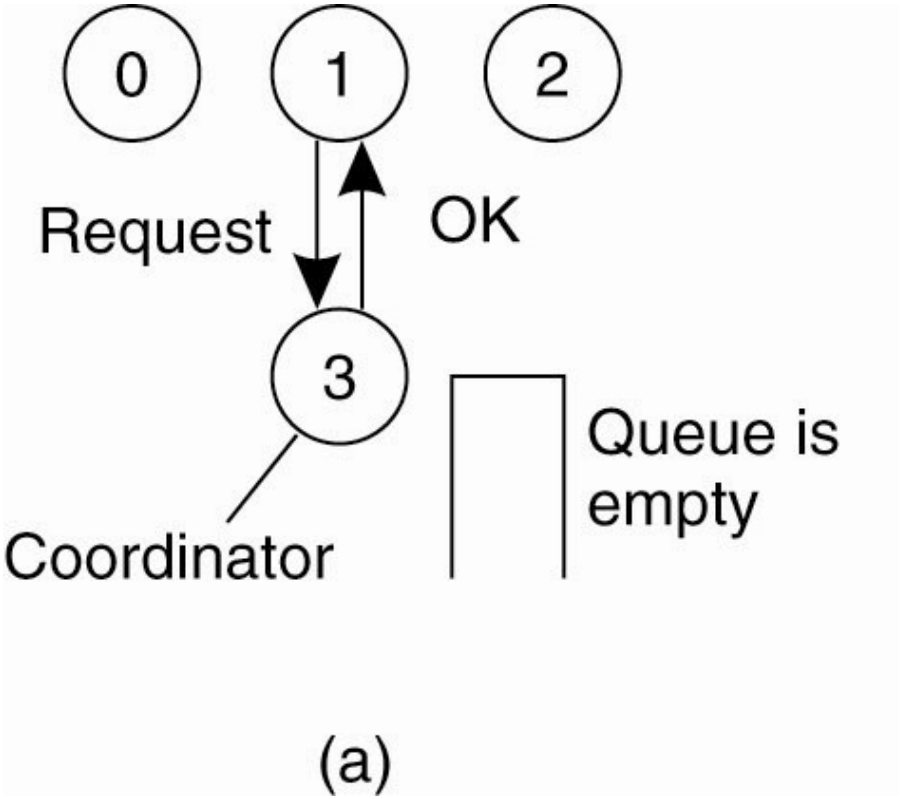

Figure 6-14. (a) Process 1 asks the coordinator for permission to access a shared resource. Permission is granted.

Mutual Exclusion
A Centralized Algorithm (2)

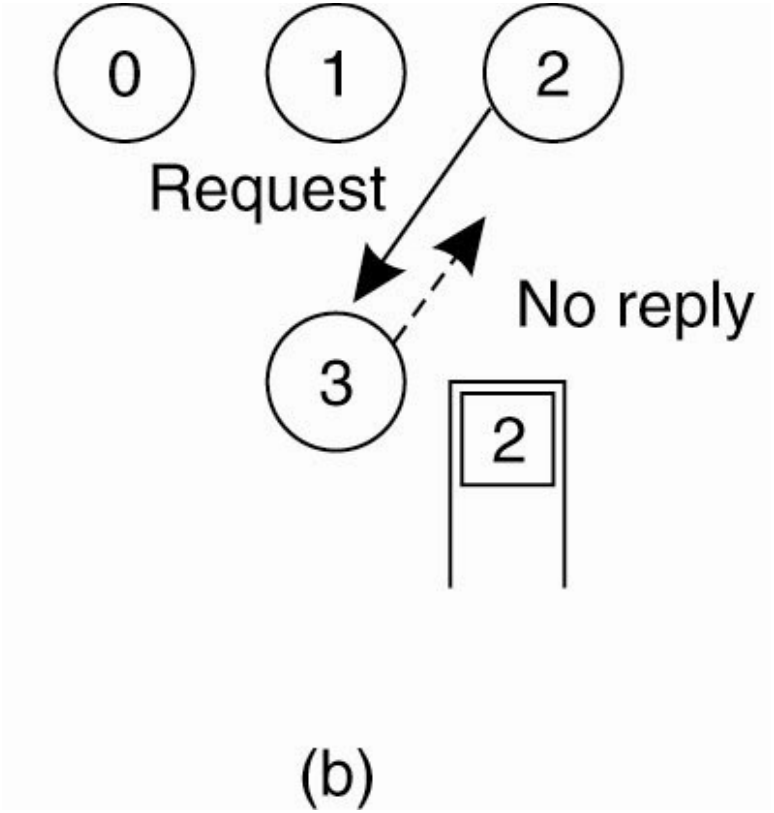

Figure 6-14. (b) Process 2 then asks permission to access the same resource. The coordinator does not reply.

Mutual Exclusion
A Centralized Algorithm (3)

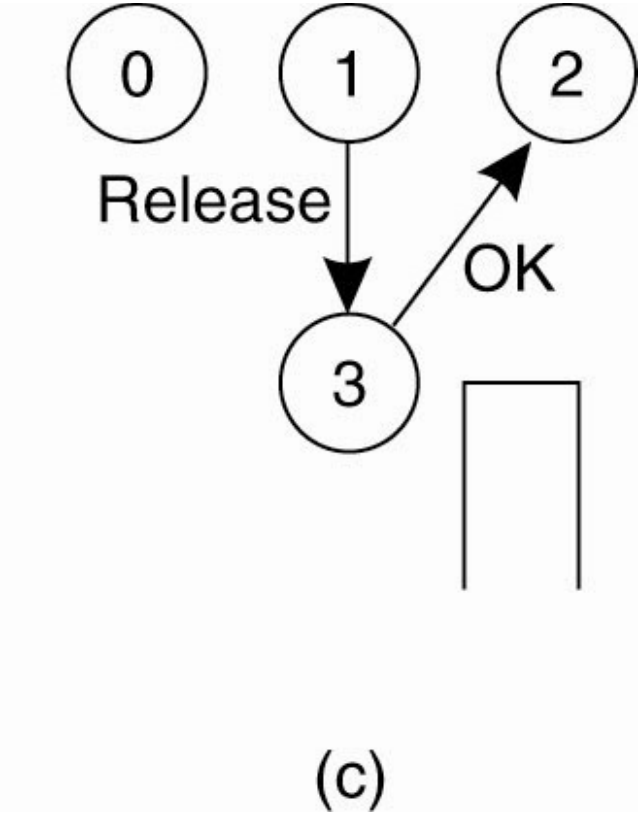

Figure 6-14. (c) When process 1 releases the resource, it tells the coordinator, which then replies to 2.

Centralized Mutual Exclusion

- Coordinator is a single point of failure
- Processes cannot distinguish between “Permission denied” and dead coordinator as in both cases no messages come back
- Performance bottleneck
- *Distributed version*: Each resource is replicated *n* times with one coordinator per replica. To access a resource we need to get a vote from a majority of the coordinators. If access is denied, the process is informed. Possible to incorrectly give access because of coordinator crashing and missing information. In practical cases, this probability is very small.

Decentralized Algorithm

- Each resource is replicated *n* times with one coordinator per replica. To access a resource we need to get a vote from a majority of the coordinators. If access is denied, the process is informed.
- Possible to incorrectly give access because of coordinator crashing and missing information. In practical cases, this probability is very small
- Implemented using DHT (Distributed Hash Table) based system. Has the problem if many nodes are competing to get the critical section, no one is able to get enough votes. However there are solutions to this problem in the literature

A Distributed Algorithm (1)

Ricart and Agarwala's Algorithm: Requires total ordering of events using Lamport logical clocks.

To access a shared resource, the process builds a message containing the name of the resource, its process number and its current logical time.

Then it sends the message to all processes (reliably).

A Distributed Algorithm (2)

Three different cases for the replies:

- If the receiver already has access to the resource, it simply does not reply. Instead, it queues the request
- If the receiver is not accessing the resource and does not want to access it, it sends back an OK message to the sender
- If the receiver wants to access the resource as well but has not yet done so, it compares the timestamp of the incoming message with the one contained in the message that it has sent everyone. The lowest one wins. Sends an OK if incoming timestamp is lower

A Distributed Algorithm (3)

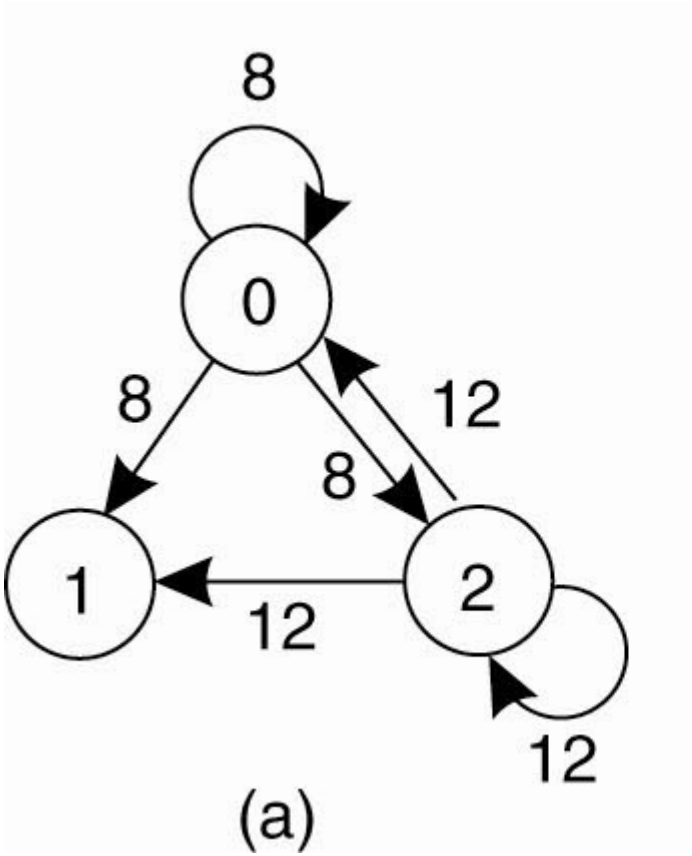

Figure 6-15. (a) Two processes want to access a shared resource at the same moment.

A Distributed Algorithm (4)

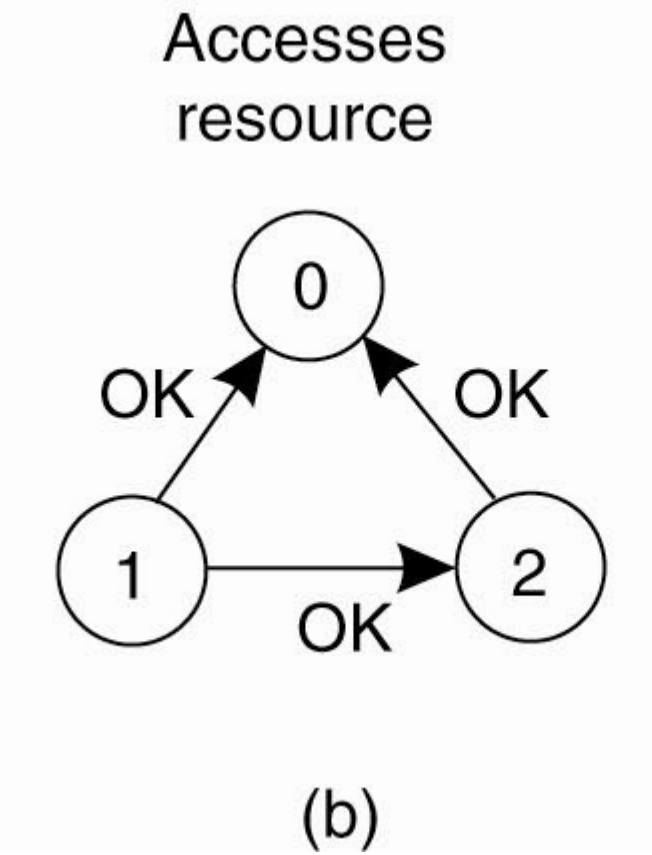

Figure 6-15. (b) Process 0 has the lowest timestamp, so it wins.

A Distributed Algorithm (5)

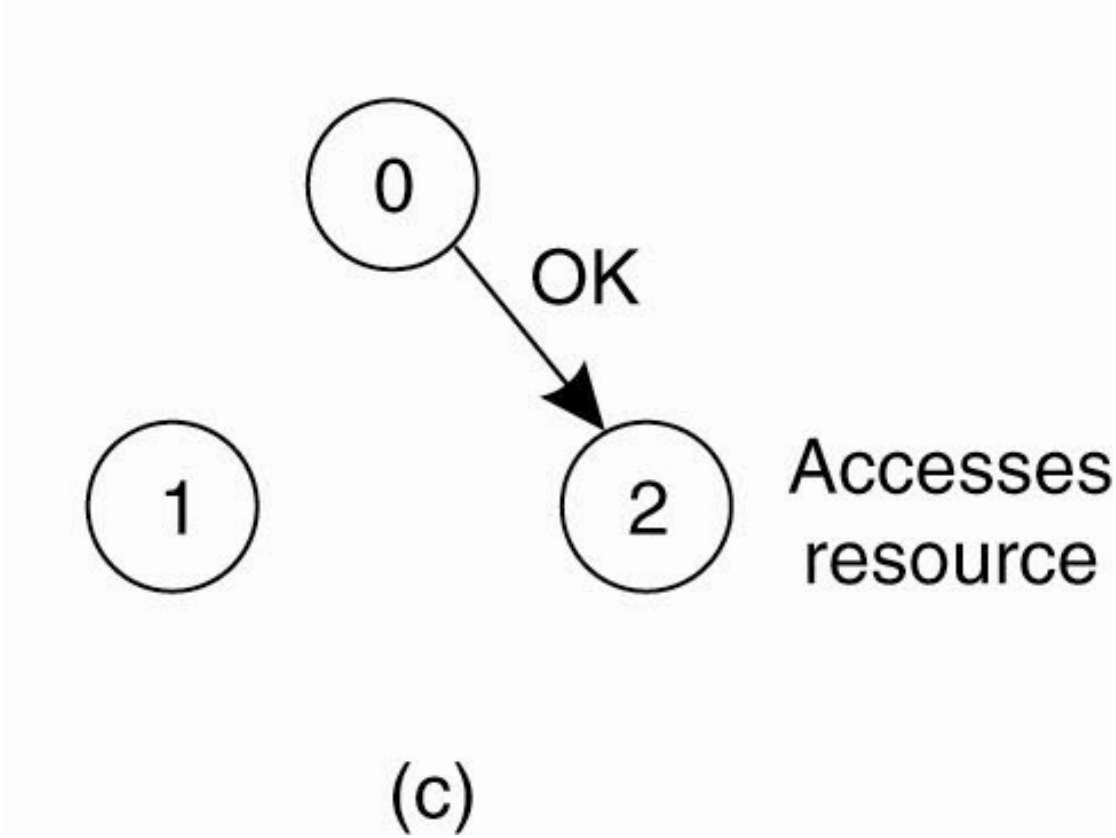

Figure 6-15. (c) When process 0 is done, it sends an OK also, so 2 can now go ahead.

A Distributed Algorithm (6)

- If a process crashes, it's silence will be incorrectly interpreted as a denial of permission
  - *Solution:* The receiver always sends a reply: either granting or denying permission
- Requires a multicast primitive or having to simulate one by maintaining group membership
- All processes are involved in all decisions, increasing the load on all processes! We could change it to a majority of processes making a decision rather than all

A Token Ring Algorithm

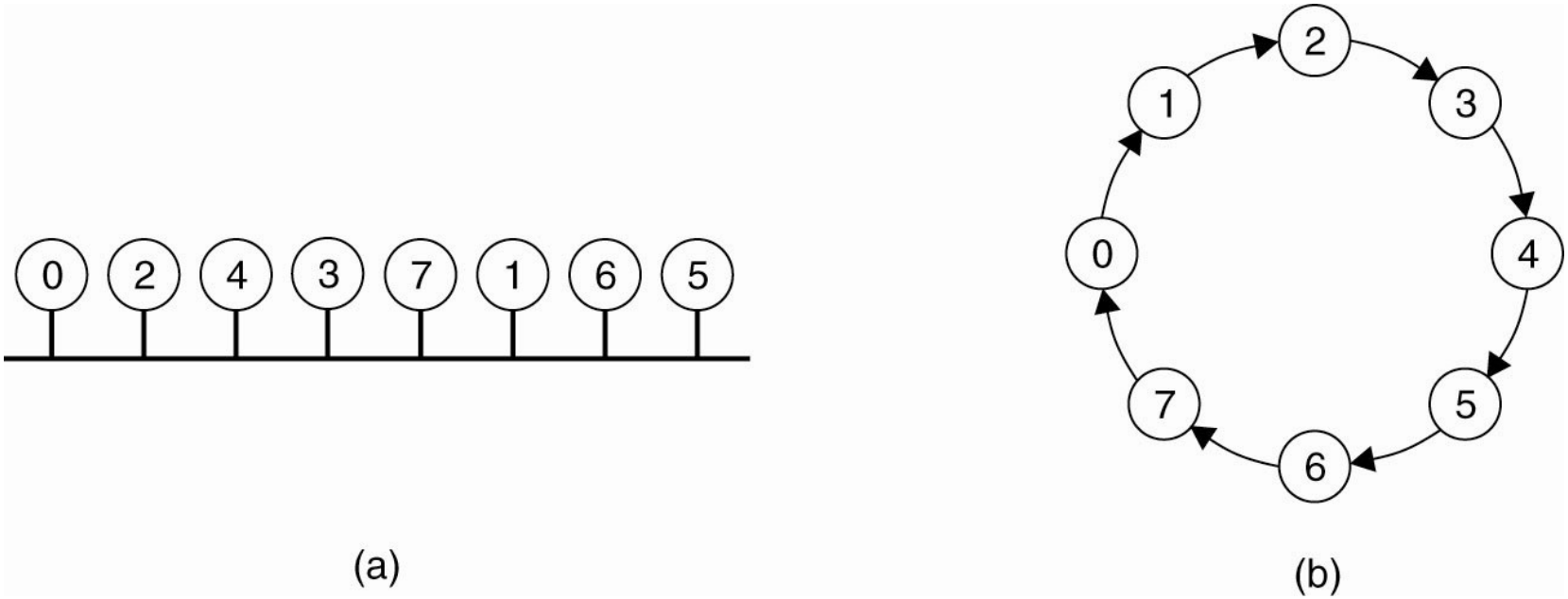

Figure 6-16. (a) An unordered group of processes on a network. (b) A logical ring constructed in software.

Token Ring Algorithm (contd)

- A token is passed from process k to k+1 (modulo ring size) in point-to-point messages. Initially process 0 has the token
- When a process has the token, then it can enter the critical section **once** if it so desires. After exiting the critical section, it passes the token onwards
- When a process receives a token and it is not interested in entering the critical section, then it just passes the token. When no one wants to enter the critical section, the token just circulates in the ring at high speed
- If a token is lost, then it must be regenerated. How can we lose the token?
  - A process that had the token crashes ... if we require an ack for the receipt of the token, then we can detect this and throw the token over the dead process to the next one down (possible if everyone has the ring topology)
  - Detecting a lost token can be difficult

A Comparison of the Four Algorithms

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Decentralized	3mk, k = 1,2,...	2 m	Starvation, low efficiency
Distributed	2 (n – 1)	2 (n – 1)	Crash of any process
Token ring	1 to ∞	0 to n – 1	Lost token, process crash

Figure 6-17. A comparison of three mutual exclusion algorithms.

Election Algorithms

The *Bully Algorithm*

- *P* sends an *ELECTION* message to all processes with higher numbers
- If no one responds, *P* wins the election and becomes coordinator
- If one of the higher-ups answers, it takes over. *P*'s job is done

The Bully Algorithm (1)

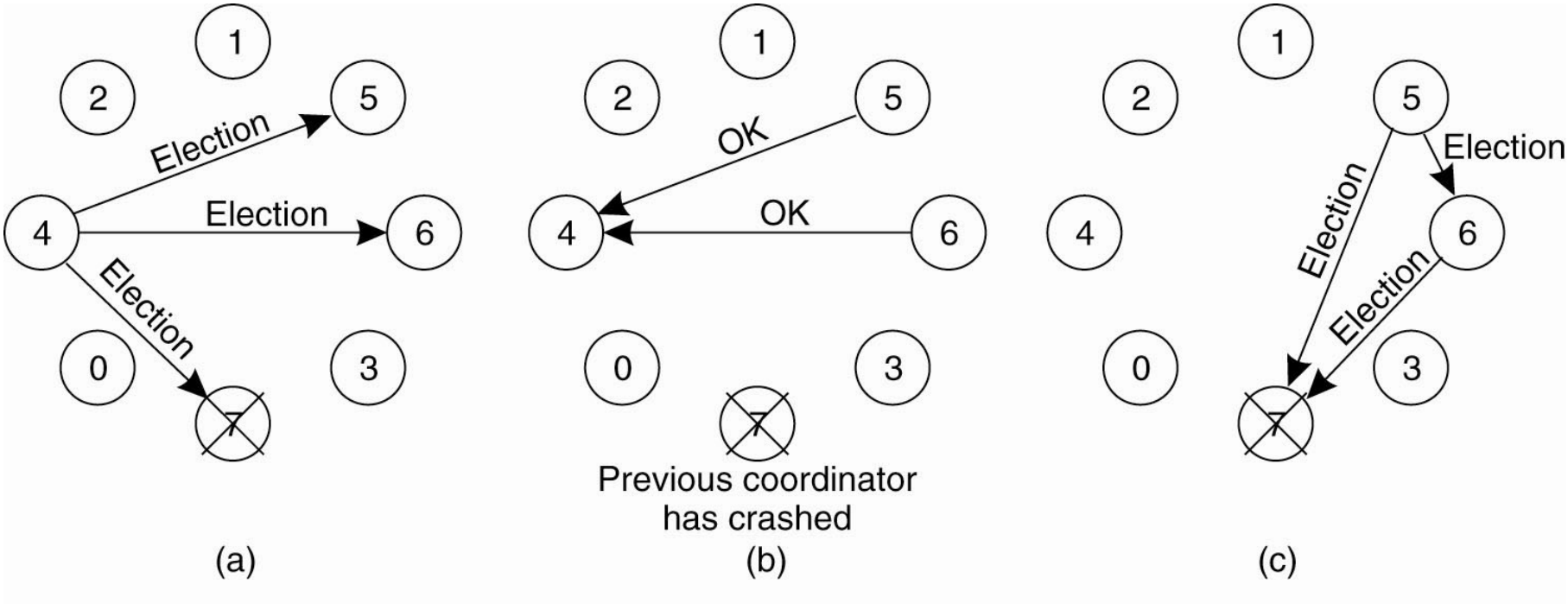

Figure 6-20. The bully election algorithm. (a) Process 4 holds an election. (b) Processes 5 and 6 respond, telling 4 to stop. (c) Now 5 and 6 each hold an election.

The Bully Algorithm (2)

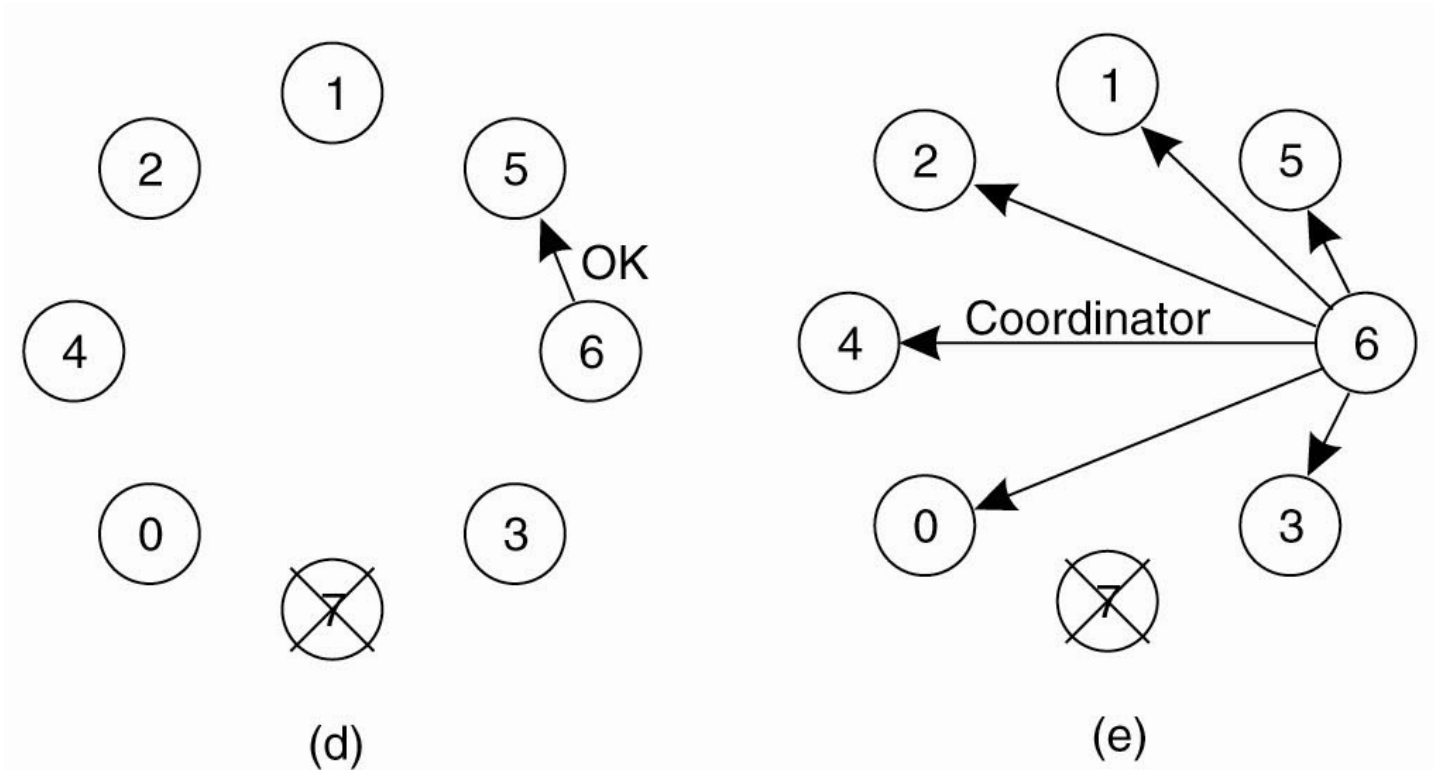

Figure 6-20. The bully election algorithm. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone.

The Bully Algorithm (3)

- The new coordinator typically has to pick up the state information left off by the old coordinator before it announces the results of the election
- If Process 7 comes back up, it just sends a new election message and bully them into submission
- We can use *Are You Alive* messages periodically to speed up detection of absconding coordinators

A Ring Algorithm

- The processes are logically arranged in a ring. Each process knows its neighbor in the ring as well who all is in the ring. When any process notices that the coordinator is nor responding, it builds an *ELECTION* message containing its own process number and sends it to its successor

- At each step, the sender adds its own number to the list in the message thus making itself be a candidate. Eventually, the message reaches the process that started it all. Then it looks through the message and decides which process has the highest number and that becomes the coordinator

- Then the message type is changed to *COORDINATOR* and the message circulates once again so everyone knows the new coordinator and the new ring configuration.

A Ring Algorithm

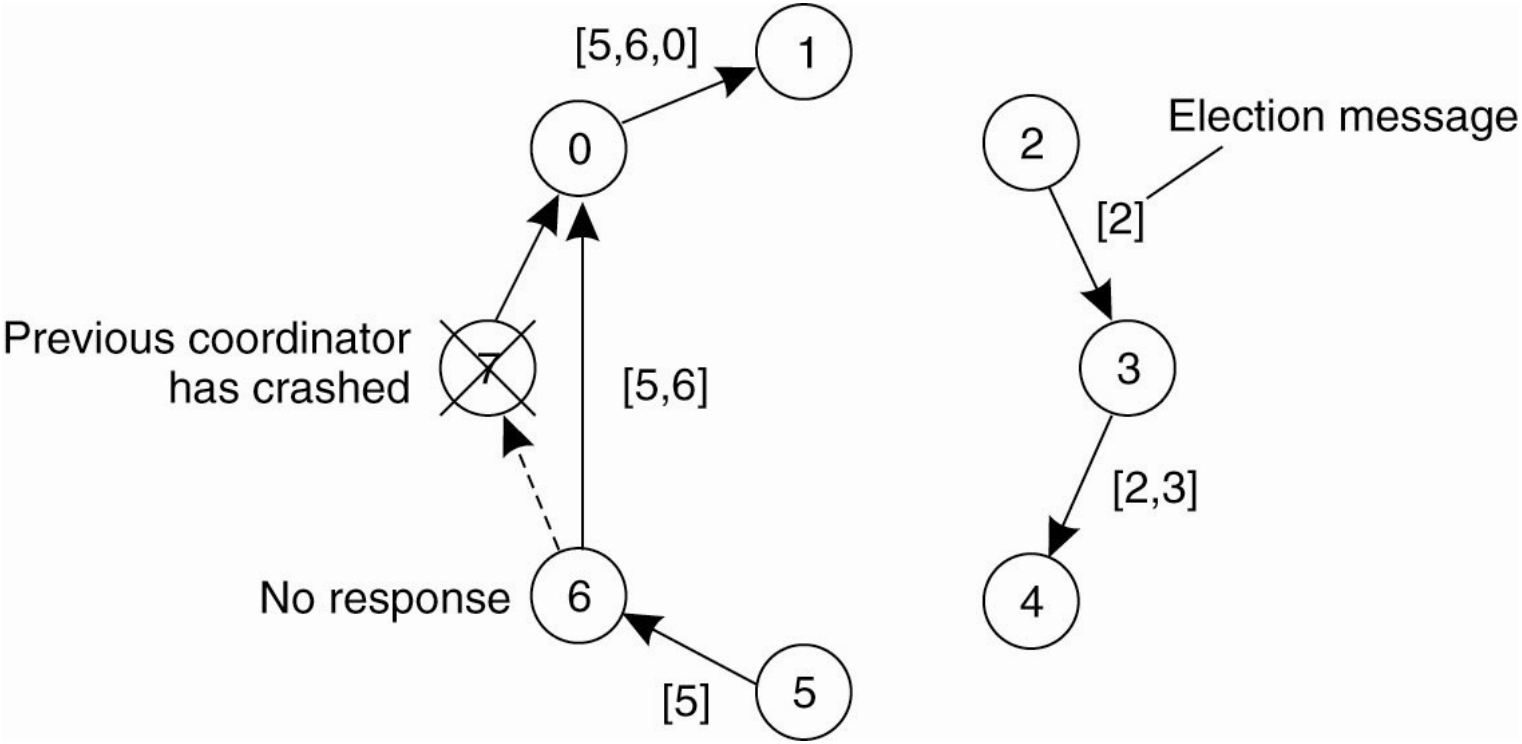

Figure 6-21. Election algorithm using a ring.