DISTRIBUTED SYSTEMS
Principles and Paradigms

Chapter 6
Synchronization

Introduction

- ➢In this, we mainly concentrate on ***how processes can synchronize***.

- ➢For example, it is important that ***multiple processes do not simultaneously access a shared resource***, such as **printer**, but instead *cooperate in granting each other temporary exclusive access*.

- ➢Another example is that *multiple processes may sometimes need to agree on the ordering of events*, such as whether *message m1 from process P* was sent **before or after** *message m2 from process Q*.

- ➢**Synchronization in distributed systems** is often **much more difficult** compared to *synchronization in uniprocessor or multiprocessor systems*.

Contents

Clock Synchronization

Background

- ➔Synchronization: coordination of actions between processes.
- ➔Processes are usually asynchronous, (operate independent of events in other processes)
- ➔Sometimes need to cooperate/synchronize
  - For mutual exclusion
  - For event ordering (was message x from process P sent before or after message y from process Q?)

Clock Synchronization(contd..)

Introduction

Synchronization in centralized systems is primarily accomplished through shared memory

- Event ordering is clear because all events are timed by the same clock

Synchronization in distributed systems is harder

-No shared memory

-No common clock

Clock Synchronization(contd..)

Introduction (contd..)

➢Some applications rely on event ordering to be successful

- Ex: **UNIX make program**
- Normally, *in UNIX, large programs are split up into multiple source files, so that a change to one source file only requires one file to be recompiled, not all the files*.
- When the *programmer has finished changing all the source files, he runs make, which examines the times at which all the source and object files were last modified*.
- If the source file ***input.c has time 2151*** and the corresponding object file ***input.o has time 2150***, make knows that ***input.c*** has been changed since ***input.o*** was created, and thus ***input.c*** must be recompiled.
- On the other hand, if ***output.c has time 2144*** and ***output.o has time 2145***, ***no compilation*** is needed.
- Thus make goes through all the source files to find out which ones need to be recompiled and calls the compiler to recompile

Clock Synchronization(contd..)

Introduction (contd..)

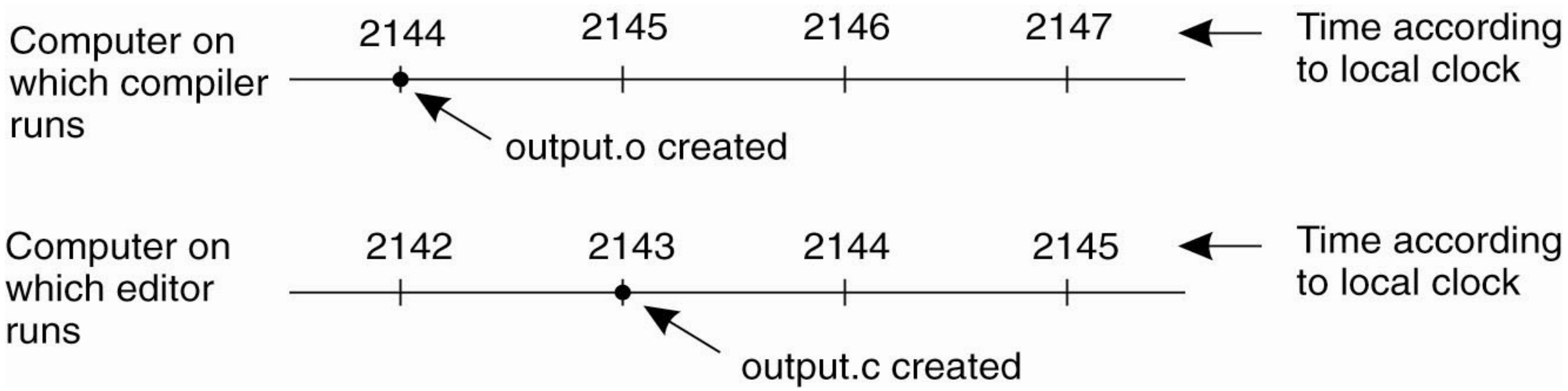

Computer on which compiler runs
2144
2145
2146
2147
Time according to local clock
output.o created
Computer on which editor runs
2142
2143
2144
2145
Time according to local clock
output.c created

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

- ✘Now imagine what could happen in a distributed system in which there were no global agreement on time.
- ✘Suppose that ***output.o has time 2144*** as above, and shortly thereafter ***output.c is modified*** but is ***assigned time 2143*** because the clock on its machine is slightly behind, as shown in Fig. 6-1.
- ✘Make will not call the compiler. The resulting executable binary program will then contain a mixture of object files from the old sources and the new sources.

Clock Synchronization(contd..)

Introduction (contd..)

➔Event ordering is easier if you can accurately time-stamp events, but in a distributed system the clocks may not always be synchronized.

Is it possible to synchronize clocks in a distributed system?

Clock Synchronization(contd..)

- ***Physical Clocks***
- ***Global Positioning System***
- ***Clock Synchronization Algorithms***

Physical Clocks

◆The word "***clock***" to refer to *an electronic device, which counts oscillations in crystal at a particular frequency*.

◆Timer: A computer timer is usually a precisely machined quartz crystal.

◆Physical clock example: counter + holding register + oscillating quartz crystal

-The counter is decremented at each oscillation

-Counter interrupts when it reaches zero

-Reloads from the holding register

-Interrupt = clock tick (often 60 times/second)

◆Software clock: counts interrupts

-This value represents number of seconds since some predetermined time (Jan 1, 1970 for UNIX systems; beginning of the Gregorian calendar for Microsoft)

-Can be converted to normal clock times

Physical Clocks (contd..)

Clock Skew

- ✓In a distributed system each computer has its own clock
- ✓Each crystal will oscillate at slightly different rate.
- ✓Over time, the software clock values on the different computers are no longer the same.
- ✓*Clock skew(offset)*: the difference between the times on two different clocks
- ✓*Clock drift* : the difference between a clock and actual time
- ✓Ordinary quartz clocks drift by ~ 1sec in 11-12 days. (10^{-6} secs/ sec)
- ✓High precision quartz clocks drift rate is somewhat better

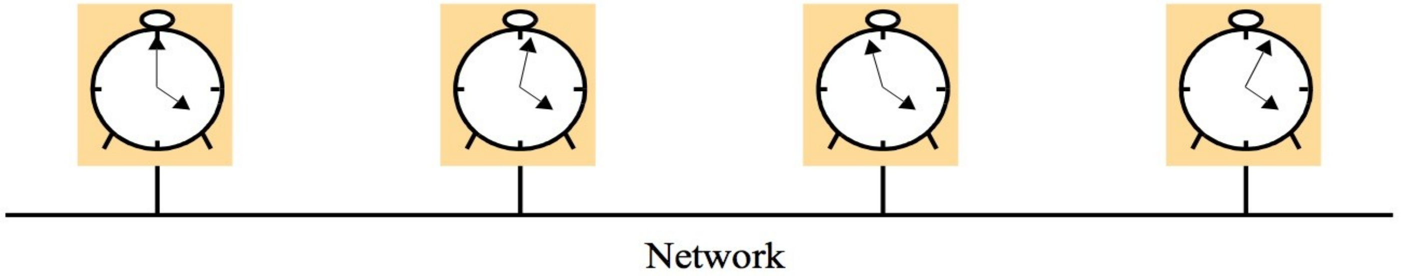


Figure 6- Skew Between Computer Clocks in a Distributed System

Physical Clocks (contd..)

Various Ways of Measuring Time*

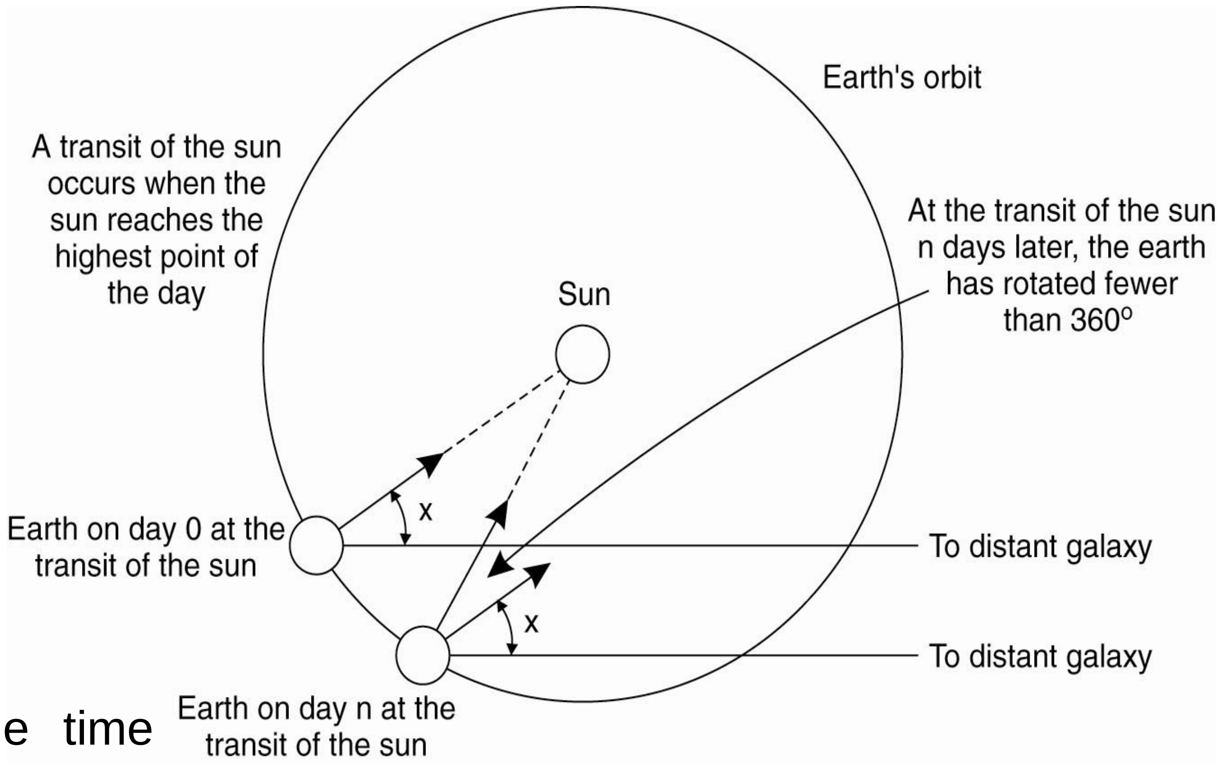
- ***The SUN***

- The mechanical clocks (17th century)
- Time has been measured ***astronomically***.
- Every day, the sun appears to rise on the eastern horizon, then climbs to a maximum height in the sky, and finally sinks in the west.
- The event of the sun's reaching its highest apparent point in the sky is called the ***transit of the sun***.
- This event occurs at about noon each day.
- The interval between two consecutive transits of the sun is called the solar day.
- Since there are 24 hours in a day, each containing 3600 seconds, the solar second is defined as exactly 1/86400th of a solar day.
- The geometry of the mean solar day calculation is shown in Fig. 6-2.

Physical Clocks (contd..)

Various Ways of Measuring Time*

The SUN



Earth's orbit
A transit of the sun occurs when the sun reaches the highest point of the day
Sun
At the transit of the sun n days later, the earth has rotated fewer than 360°
x
Earth on day 0 at the transit of the sun
To distant galaxy
x
To distant galaxy
Earth on day n at the transit of the sun

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

The mean solar day is the time between to sun transits.

1 second = 1 solar day / (24*60*60).

Sun transit: reaching highest apparent point. 300 millions years ago, the year was 400 days.

Mean solar second – gradually getting longer as earth’s rotation slows.

Physical Clocks (contd..)

Various Ways of Measuring Time*

- International Atomic Time (TAI) -1948

 -Atomic clocks are based on transitions of the cesium 133 atom
 -Atomic second = value of solar second at some fixed time (no longer accurate)

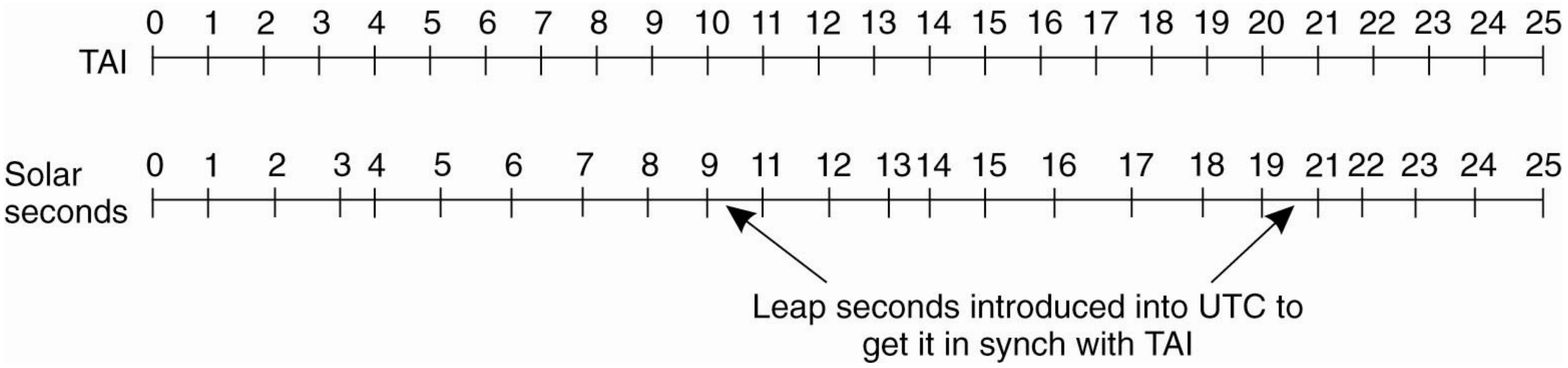


Figure 6-3. TAI seconds are of constant length, unlike solar seconds. Leap seconds are introduced when necessary to keep in phase with the sun.

Physical Clocks (contd..)

Various Ways of Measuring Time*

➢ Universal Coordinated Time (UTC)

-Based on TAI seconds, but more accurately reflects sun time (inserts leap seconds to synchronize atomic second with solar second)

✓ Getting the Correct (UTC) Time*

- WWV radio station or similar stations in other countries (accurate to +/- 10 msec)
- UTC services provided by earth satellites (accurate to 0.5 msec)
- GPS (Global Positioning System) (accurate to 20-35 nanoseconds)

Physical Clocks (contd..)

Various Ways of Measuring Time*

Problem: Sometimes we simply need the exact time, not just an ordering.

Solution: Universal Coordinated Time (UTC):

- Based on the number of transitions per second of the cesium 133 atom (pretty accurate), started in 1958.
- At present, the real time is taken as the average of some 50 cesium-clocks around the world.
- Introduces a leap second from time to time to compensate that days are getting longer.

Note: UTC is broadcast through short wave radio (WWV) and geostationary environment operational satellite (GEOS). Satellites can give an accuracy of about ± 0.5 ms.

Global Positioning System (GPS)

- GPS is a **satellite-based distributed system** that was launched in 1978.
- Although it has been used mainly for **military applications,** in recent years it has found its way to many **civilian applications**, notably for **traffic navigation**.
- GPS uses **29 satellites** each circulating in an orbit at a height of approximately **20,000 km.**

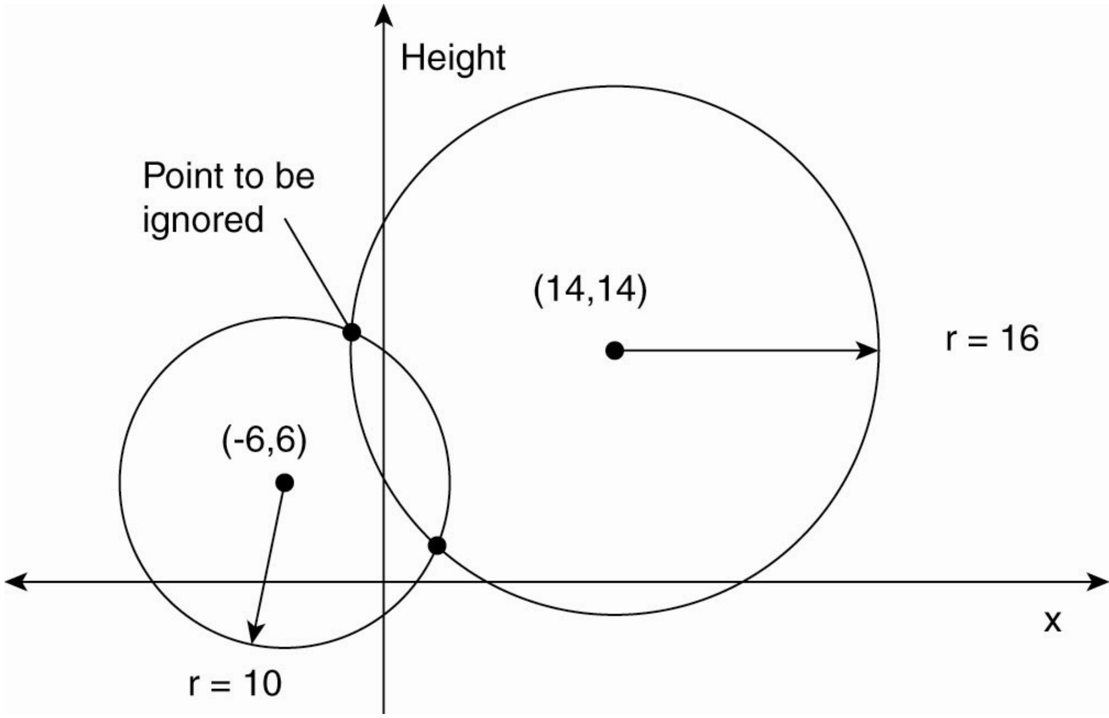


Figure 6-4. Computing a position in a two-dimensional space.

- In order to compute a position, consider first the two-dimensional case, as shown in Fig. 6-4, in which two satellites are drawn, along with the circles representing points at the same distance from each respective satellite.
- The y-axis represents the height, while the x-axis represents a straight line along the Earth's surface at sea level.
- Ignoring the highest point, we see that the intersection of the two circles is a unique point.

Global Positioning System (GPS) [contd..]

Real world facts that complicate GPS

1. It takes a while before data on a satellite's position reaches the receiver.
2. The receiver's clock is generally not in synch with that of a satellite.

Therefore, need 4 satellites → 4 equations in 4 unknowns.

Clock Synchronization Algorithms

- In a distributed system one machine may have a WWV receiver and some technique is used to keep all the other machines in synch with this value.
- Or, no machine has access to an external time source and some technique is used to keep all machines synchronized with each other, if not with “real” time.

Clock Synchronization Algorithms (contd..)

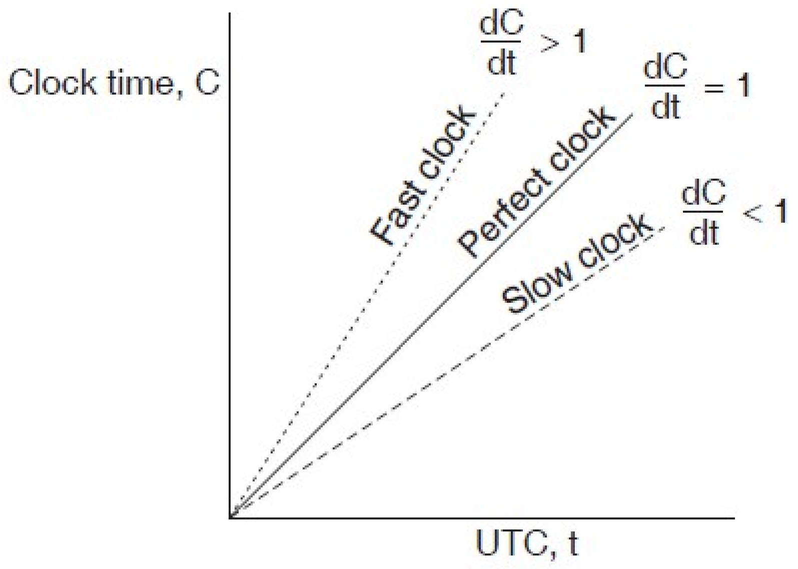
UTC

Problem: Suppose we have a distributed system with a UTC-receiver somewhere in it → we still have to distribute its time to each machine.

Basic principle

- Every machine has a timer that generates an interrupt H times per second.
- There is a clock in machine p that 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 machine p, $C_p(t) = t$, or, in other words, $dC/dt = 1$.

Clock Synchronization Algorithms (contd..)



In practice: $1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho$.

Goal: Never let two clocks in any system differ by more than δ time units $\rightarrow$ synchronize at least every $\delta / (2\rho)$ seconds.

Clock Synchronization Algorithms (contd..)

Network Time Protocol

Originally proposed by **Cristian** (1989) is to let ***clients contact a time server***.

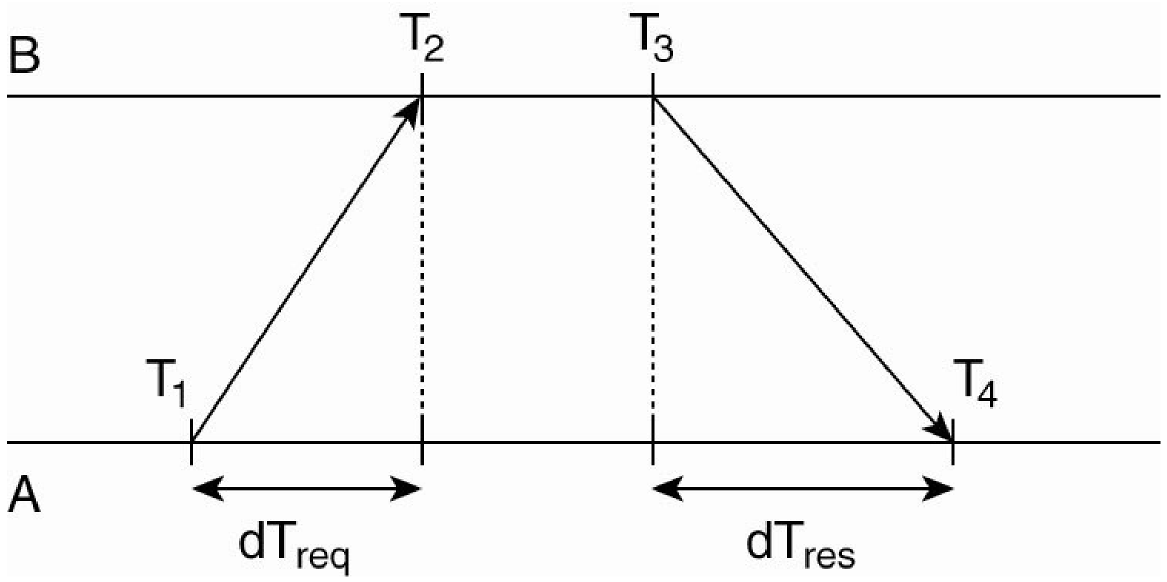


Figure 6-6. Getting the current time from a time server.

Consider the situation sketched in Fig. 6-6.

- In this case, A will send a request to B, timestamped with value T1. B, in turn, will record the time of receipt T2 (taken from its own local clock), and returns a response timestamped with value T3, and ***piggybacking*** the previously recorded value T2.
- Finally, A records the time of the response's arrival, T4.

Clock Synchronization Algorithms (contd..)

Network Time Protocol [contd..]

Let us assume that the propagation delays from A to B is roughly the same as B to A, meaning that T2 - T1 $\approx$ T4 -T3.

In that case, A can estimate its offset relative to Bas

$$\theta = T_3 + ((T_2-T_1)+(T_4-T_3))/2 - T_4$$

$\theta = ((T2-T1)+(T3-T4))/2$; -ve slowdown clock, +ve run faster, no jumbs.

Getting the current time from a time server.

The NTP does 8 rounds and adopt the minimum time round.

Clock Synchronization Algorithms (contd..)

Network Time Protocol [contd..]

- Network Time Protocol (NTP) is a protocol for synchronizing the clocks of computer systems over packet-switched, variable-latency data networks with primary servers synchronizing to primary sources:

  GPS satellite (worldwide), GOES satellite, WWV radio (US), MSF radio (UK), Modem time service (NIST,PTB Germany, etc.)

- NTP uses a hierarchical, layered system of levels of clock sources. Each level is called a “stratum” with a level number i.e. 0, 1, 2, 3. The level indicates its distance from the reference clock.

  Stratum 0: devices such as atomic clocks (caesium), GPS clocks or radio clocks
  Stratum 1, 2, 3 are time servers with high numbered servers sending NTP requests.

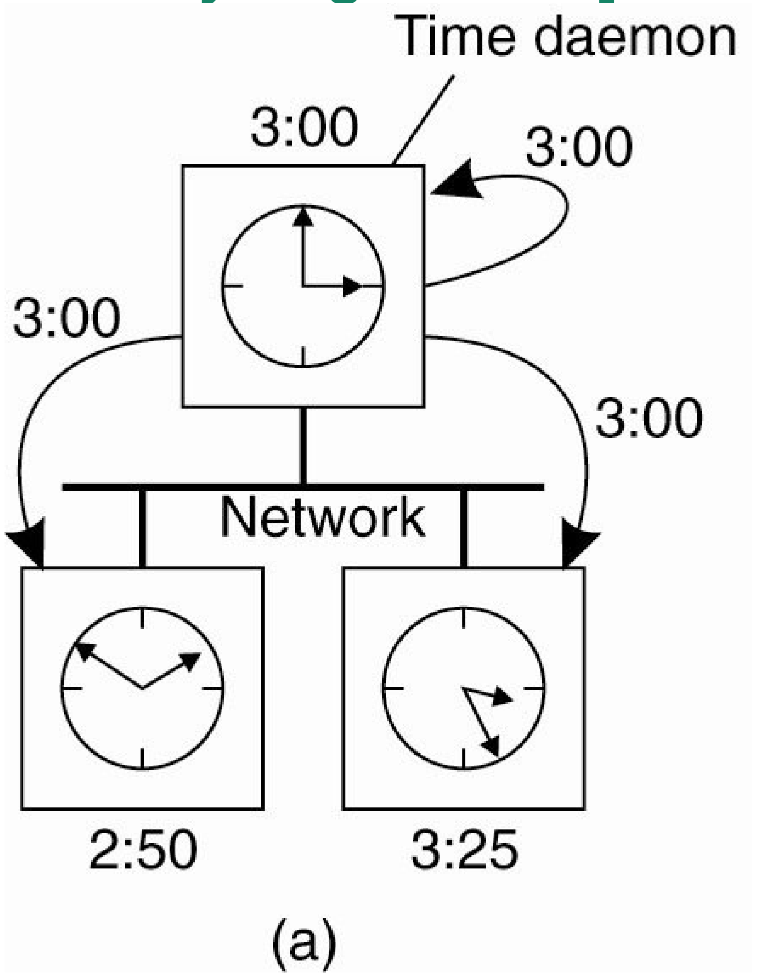
- NTP supports upto 256 strata

Clock Synchronization Algorithms (contd..)

The Berkeley Algorithm

- In NTP, the time server is **passive**. Other machines **periodically ask it for the time.** All it does is respond to their queries.
- Where as in Berkeley UNIX, here the **time server (actually, a time daemon)** is *active*, *polling every machine from time to time to ask what time* it is there.
- Based on the answers, *it computes an average time* and ***tells all the other machines to advance their clocks to the new time*** or ***slow their clocks down until some specified reduction*** has been achieved.

Clock Synchronization Algorithms (contd..)
The Berkeley Algorithm [contd..]



- In Fig. 6-7(a), at 3:00, the time daemon tells the other machines its time and asks for theirs.

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

Clock Synchronization Algorithms (contd..)

The Berkeley Algorithm [contd..]

- In Fig. 6-7(b), they respond with how far ahead or behind the time daemon they are.

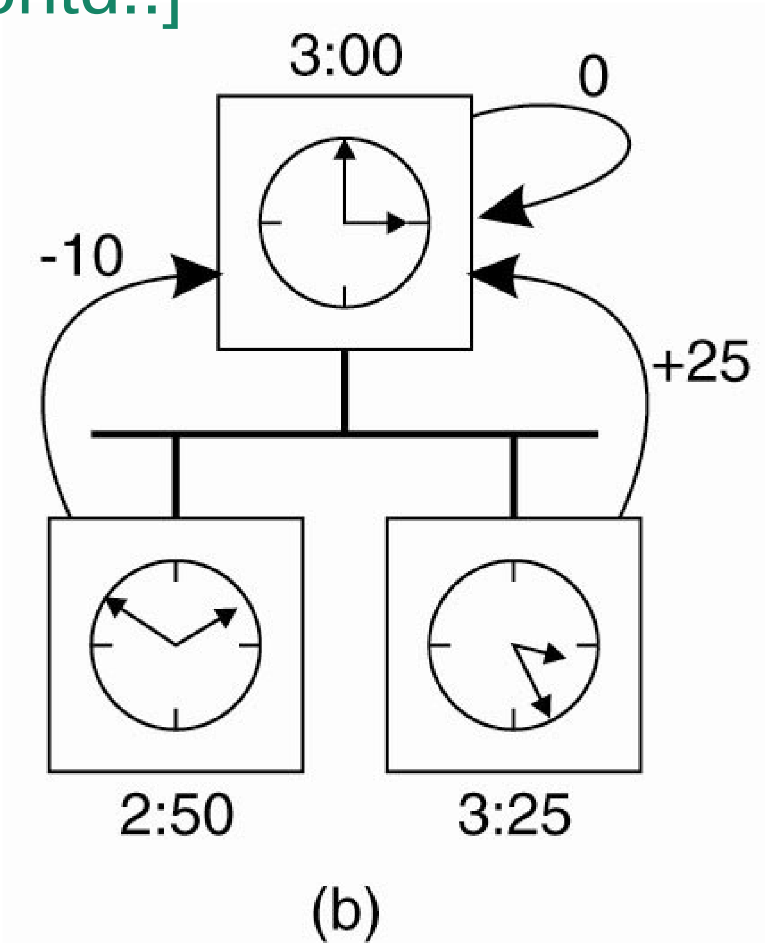


Figure 6-7. (b) The machines answer.

Clock Synchronization Algorithms (contd..)

The Berkeley Algorithm [contd..]

- Armed with these numbers, the time daemon computes the average and tells each machine how to adjust its clock.

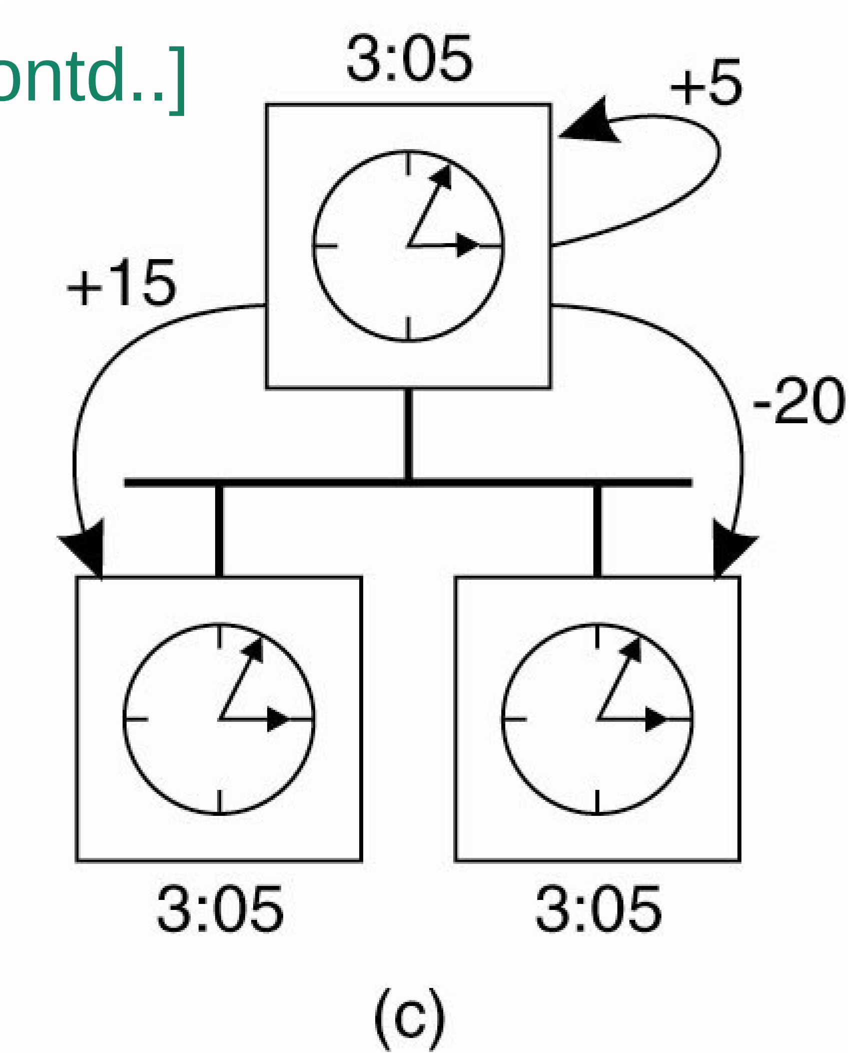


Figure 6-7. (c) The time daemon tells everyone how to adjust their clock.

Clock Synchronization in Wireless Networks (1)

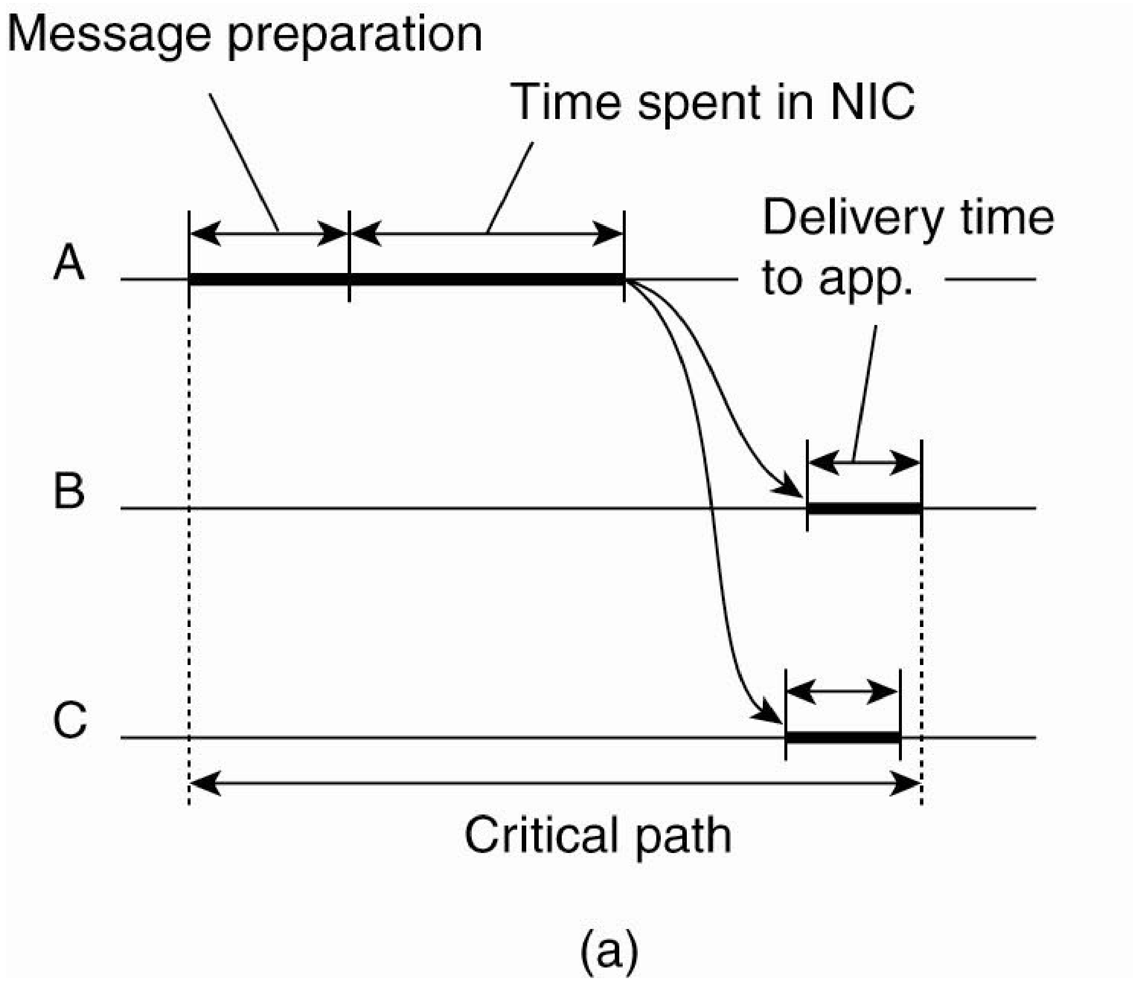


Figure 6-8. (a) The usual critical path in determining network delays.

Clock Synchronization in Wireless Networks (2)

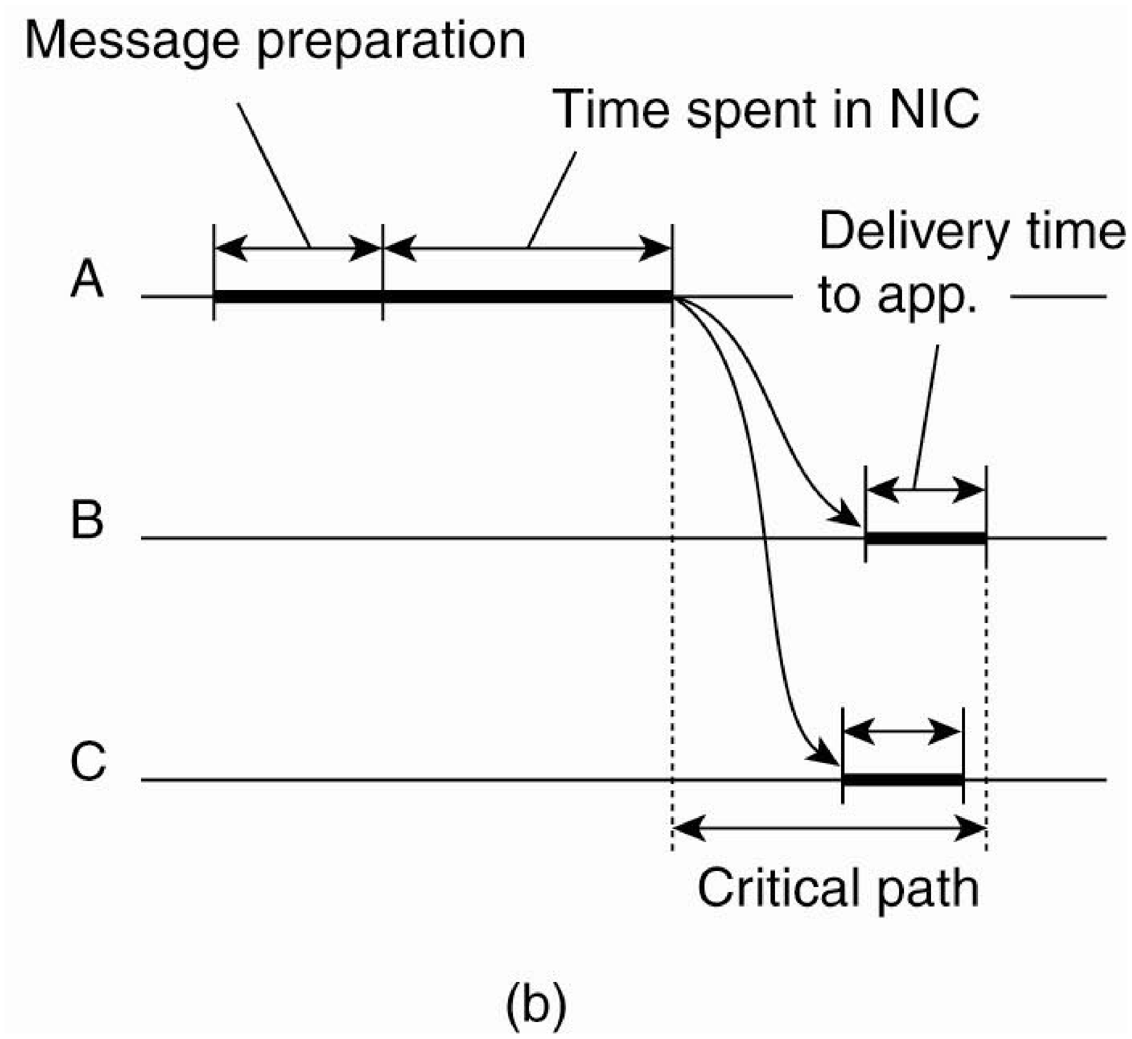


Figure 6-8. (b) The critical path in the case of RBS.

Logical clocks

- **Lamport's Logical Clocks**
  - **Example: Totally-ordered multicast**
- **Vector Clocks**
  - **Causally-ordered multicast**

Lamport’s Logical Clocks (1)

The "happens-before" relation → can be observed directly in two situations:

- If *a* and *b* are events in the same process, and *a* occurs before *b*, then *a* → *b* is true.
- If a is the event of a message being sent by one process, and *b* is the event of the message being received by another process, then *a* → *b*

Lamport's Logical Clocks (2)

Each process P_i maintains a local counter C_i and adjusts this counter according to the following rules:

1. For any two successive events that take place within P_i , C_i is incremented by 1.
2. Each time a message m is sent by process P_i , the message receives a timestamp $ts(m) = C_i$.
3. Whenever a message m is received by a process P_j, P_j adjusts its local counter C_j to max{C_j , $ts(m)$}; then executes step 1 before passing m to the application.

Lamport’s Logical Clocks (3)

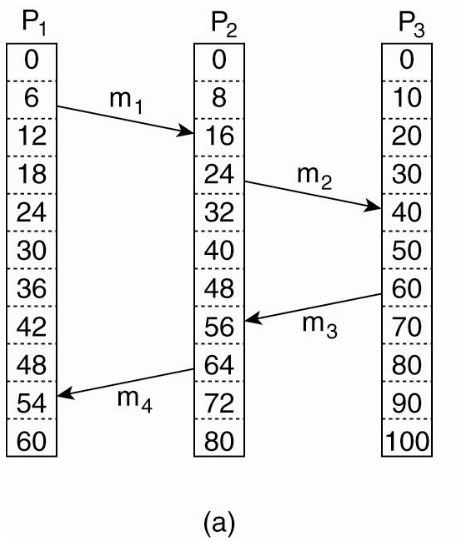


Figure 6-9. (a) Three processes, each with its own clock. The clocks run at different rates.

- The algorithm Lamport proposed for assigning times to events.
- Consider the three processes depicted in Fig. 6-9(a). The processes run on different machines, each with its own clock, running at its own speed. As can be seen from the figure, when the clock has ticked 6 times in process P1, it has ticked 8 times in process P2 and 10 times in process P3.
- Each clock runs at a constant rate, but the rates are different due to differences in the crystals.
- At time 6, process P1 sends message m1 to process P2. How long this message takes to arrive depends on whose clock you believe.

- In any event, the clock in process P2 reads 16 when it arrives. If the message carries the starting time, 6, in it, process P2 will conclude that it took 10 ticks to make the journey.
- This value is certainly possible. According to this reasoning, message m2 from P2 to P3 takes 16 ticks, again a plausible value.
- Now consider message m3. It leaves process P3 at 60 and arrives at 2 at 56. Similarly, message m4 from P2 to P1 leaves at 64 and arrives at 54.
- These values are clearly impossible. It is this situation that must be prevented.

Lamport's Logical Clocks (4)

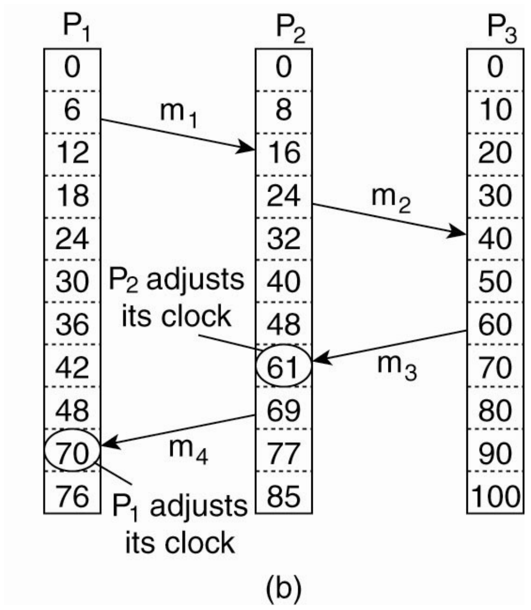


Figure 6-9. (b) Lamport's algorithm corrects the clocks.

- Lamport's solution follows directly from the happens-before relation.
- Since m3 left at 60, it must arrive at 61 or later. Therefore, each message carries the sending time according to the sender's clock. When a message arrives and the receiver's clock shows a value prior to the time the message was sent, the receiver fast forwards its clock to be one more than the sending time.
- In Fig. 6-9(b) we see that m3 now arrives at 61. Similarly, m4 arrives at 70.

Lamport’s Logical Clocks (5)

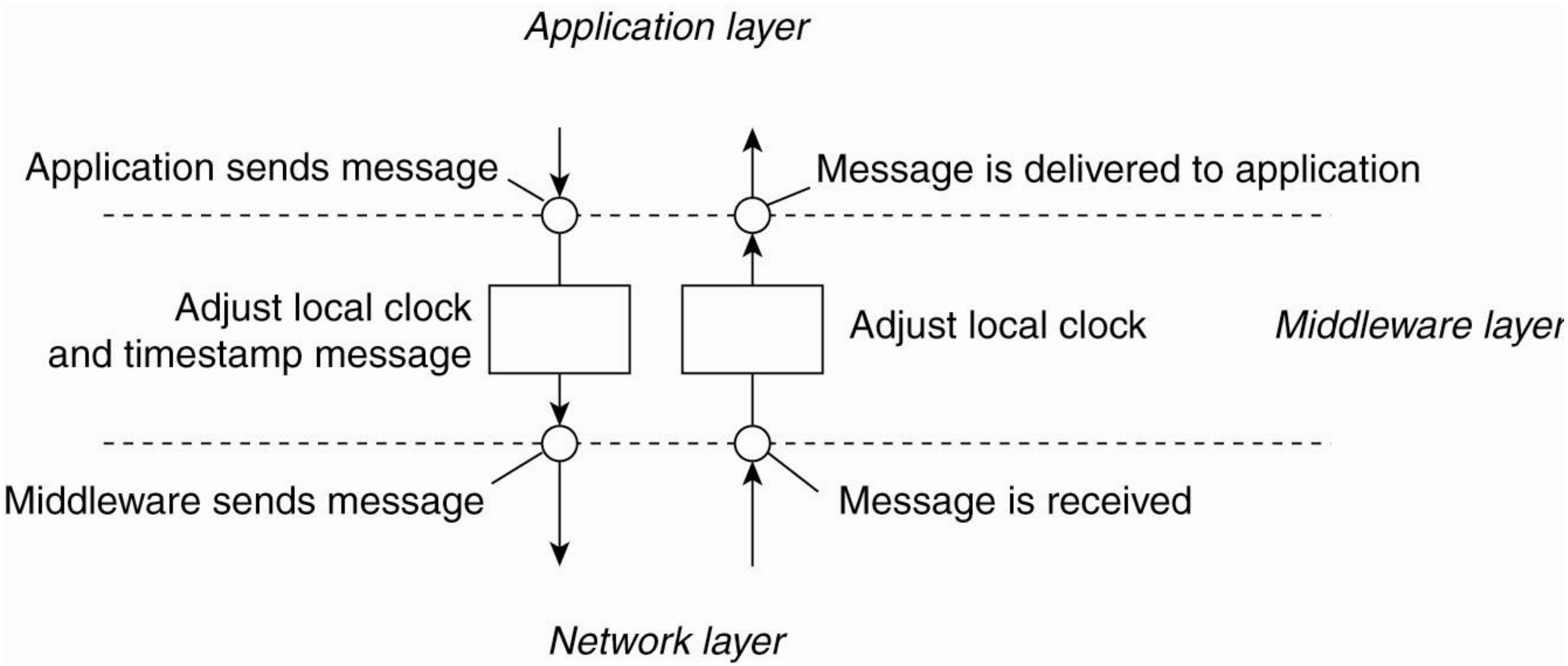


Figure 6-10. The positioning of Lamport’s logical clocks in distributed systems.

Lamport's Logical Clocks (6)

Updating counter C_i for process P_i

1. Before executing an event P_i executes $C_i \leftarrow C_i + 1$.
2. 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.
3. 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

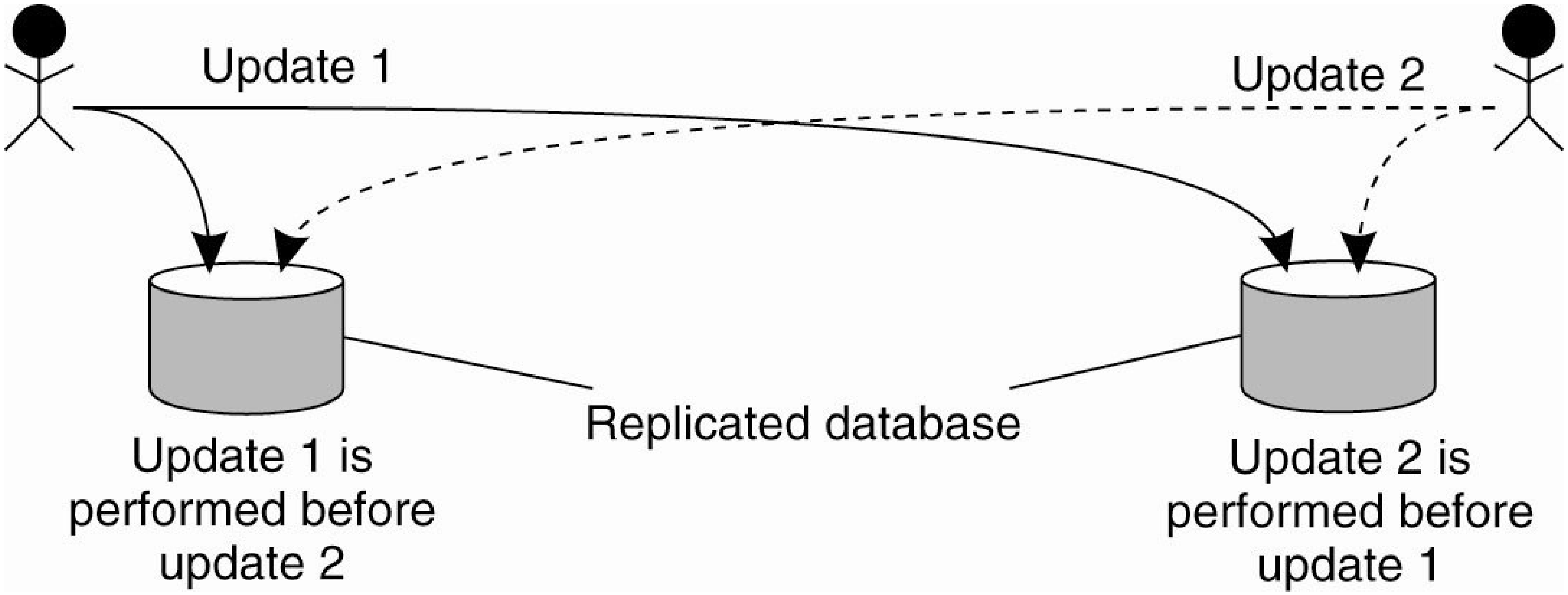

Update 1
Update 2
Replicated database
Update 1 is performed before update 2
Update 2 is performed before update 1

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

Updating a replicated database and leaving it in an inconsistent state.

- P1 adds $100 to an account (initial value: $1000)
- P2 increments account by 1%
- There are two replicas

Example: Totally Ordered Multicasting

Solution

- Process P_i sends timestamped message msg_i to all others. The message itself is put in a local queue $queue_i$.
- Any incoming message at P_j is queued in $queue_j$, according to its timestamp, and acknowledged to every other process.

P_j passes a message msg_i to its application if:

1. msg_i is at the head of $queue_j$
2. for each process P_k, there is a message msg_k in $queue_j$ with a larger timestamp.

Note: We are assuming that communication is reliable and FIFO ordered.

Vector Clocks (1)

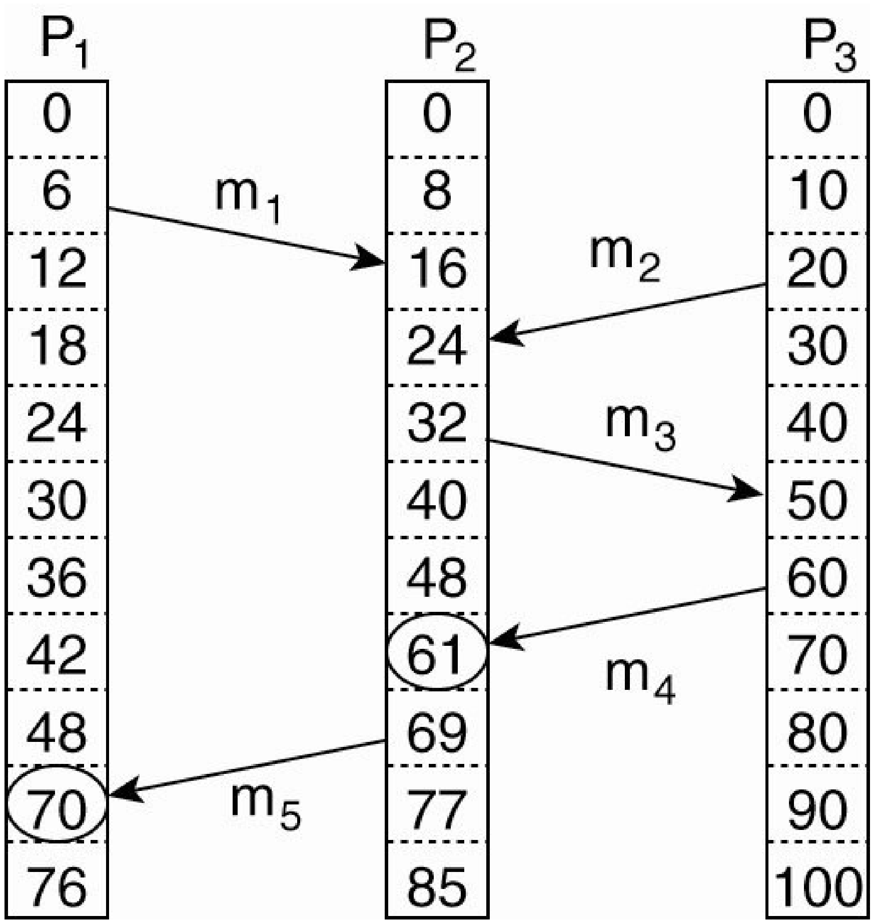


Figure 6-12. Concurrent message transmission using logical clocks.

Lamport's clocks do not guarantee that if *C(a)* < *C(b)* that *a* causally preceded *b*

Event *a: m_1* is received at *T* = 16;

Event *b: m_2* is sent at *T* = 20.

We cannot conclude that *a* causally precedes *b*.

Vector Clocks (2)

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)

Steps carried out to accomplish property 2 of previous slide:

1. Before executing an event P_i executes $VC_i[\,i\,] \leftarrow VC_i\,[i\,] + 1$.
2. 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.
3. 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.

Vector Clocks (4)

Observation

We can now ensure that a message is delivered only if all causally preceding messages have already been delivered.

Adjustment

P_i increments $VC_i[i]$ only when sending a message, and P_j "adjusts" VC_j when receiving a message (i.e., effectively does not change $VC_j[j]$).

Pj postpones delivery of *m* until:

1. $ts(m)[i] = VC_j[i]+1$ (next expected msg from P_i).
2. $ts(m)[k] \leq VC_j[k]$ for all $k \neq i$ (P_j has seen all msgs seen by P_i).

Enforcing Causal Communication

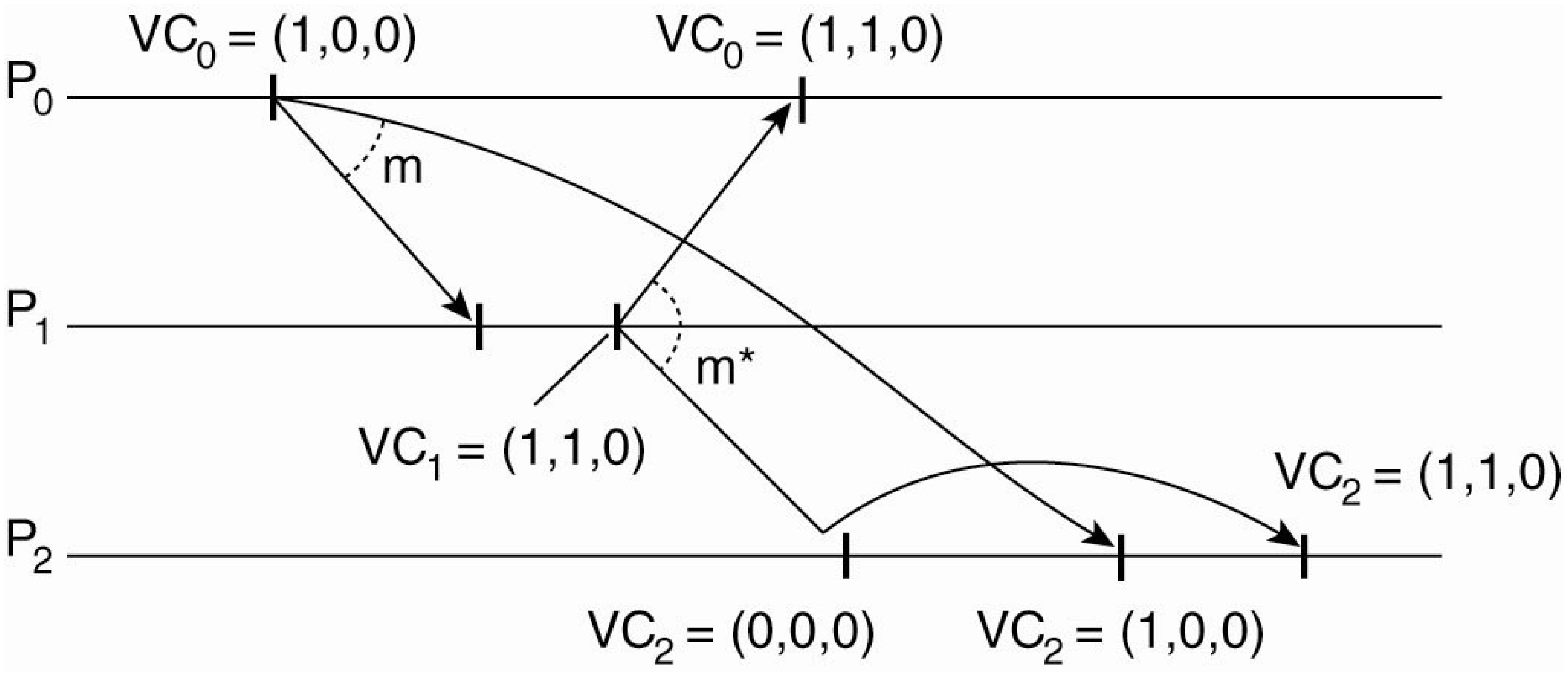


Figure 6-13. Enforcing causal communication.

Mutual Exclusion

- A Centralized Algorithm
- A Decentralized Algorithm
- A Distributed Algorithm
- A Token Ring Algorithm

Mutual Exclusion
A Centralized Algorithm (1)

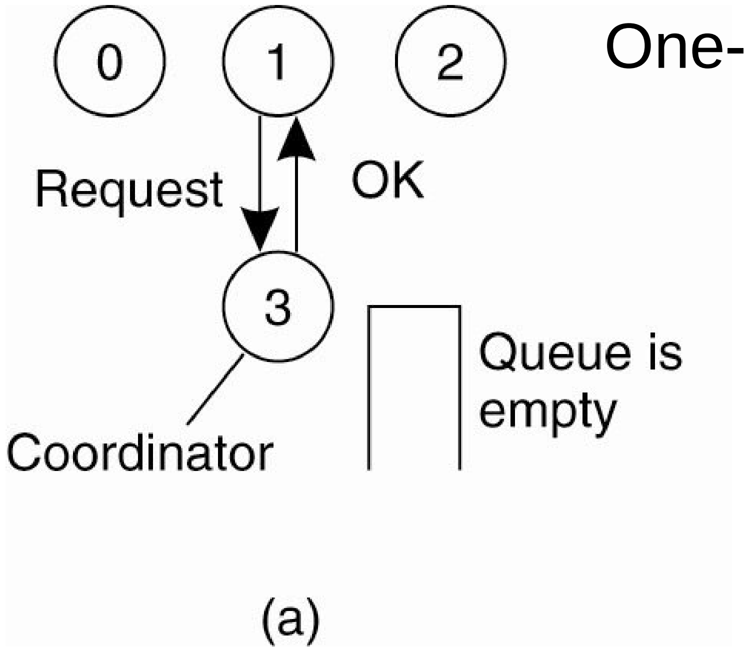


Figure 6-14. (a) Process 1 asks the coordinator for permission to access a hared resource. Permission is granted.

One-processor system

- One process is elected as the coordinator.
- Whenever a process wants to access a shared resource, it sends a request message to the coordinator stating which resource it wants to access and asking for permission.
- If no other process is currently accessing that resource, the coordinator sends back a reply granting permission, as shown in Fig.6-14(a).
- When the reply arrives, the requesting process can go ahead.

Mutual Exclusion
A Centralized Algorithm (2)

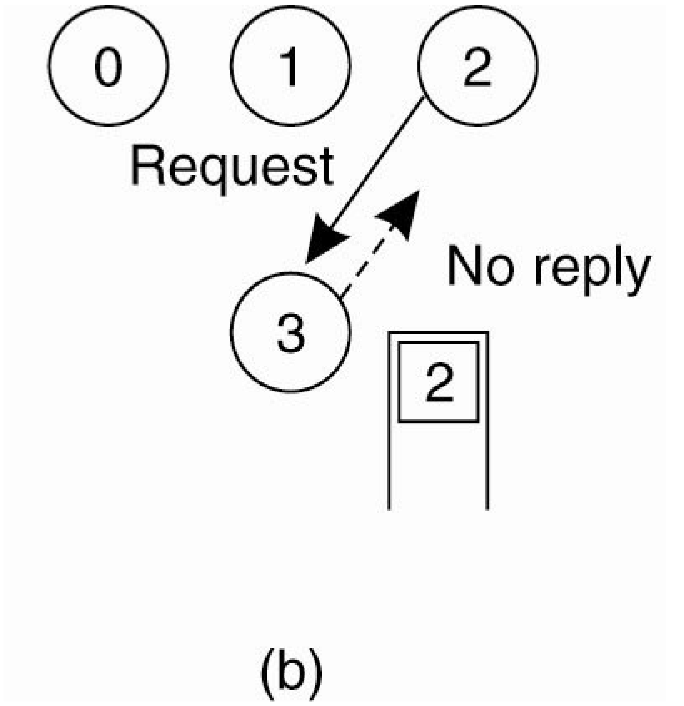


Figure 6-14. (b) Process 2 then asks permission to access the same resource. The coordinator does not reply.

- Now suppose that another process, 2 in Fig. 6-14(b), asks for permission to access the resource.
- The coordinator knows that a different process is already at the resource, so it cannot grant permission. The exact method used to deny permission is system dependent.
- In Fig. 6-14(b), the coordinator just refrains from replying, thus blocking process 2, which is waiting for a reply.
- Alternatively, it could send a reply saying "permission denied." Either way, it queues the request from 2 for the time being and waits for more messages.

Mutual Exclusion
A Centralized Algorithm (3)

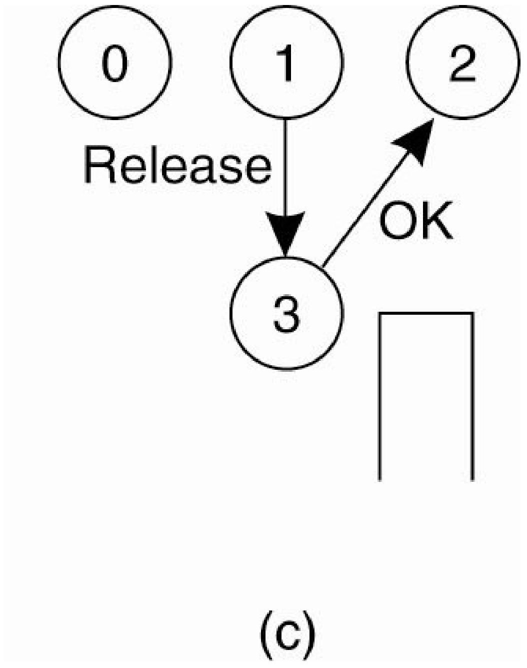


Figure 6-14. (c) When process 1 releases the resource, it tells the coordinator, which then replies to 2.

- When process 1 is finished with the resource, it sends a message to the coordinator releasing its exclusive access, as shown in Fig.6-14(c).
- The coordinator takes the first item off the queue of deferred requests and sends that process a grant message.
- If the process was still blocked (i.e., this is the first message to it), it unblocks and accesses the resource.
- If an explicit message has already been sent denying permission, the process will have to poll for incoming traffic or block later.
- Either way, when it sees the grant, it can go ahead as well.

A Distributed Algorithm (1)

Three different cases:

1. If the receiver is not accessing the resource and does not want to access it, it sends back an OK message to the sender.
2. If the receiver already has access to the resource, it simply does not reply. Instead, it queues the request.
3. 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.

A Distributed Algorithm (2)

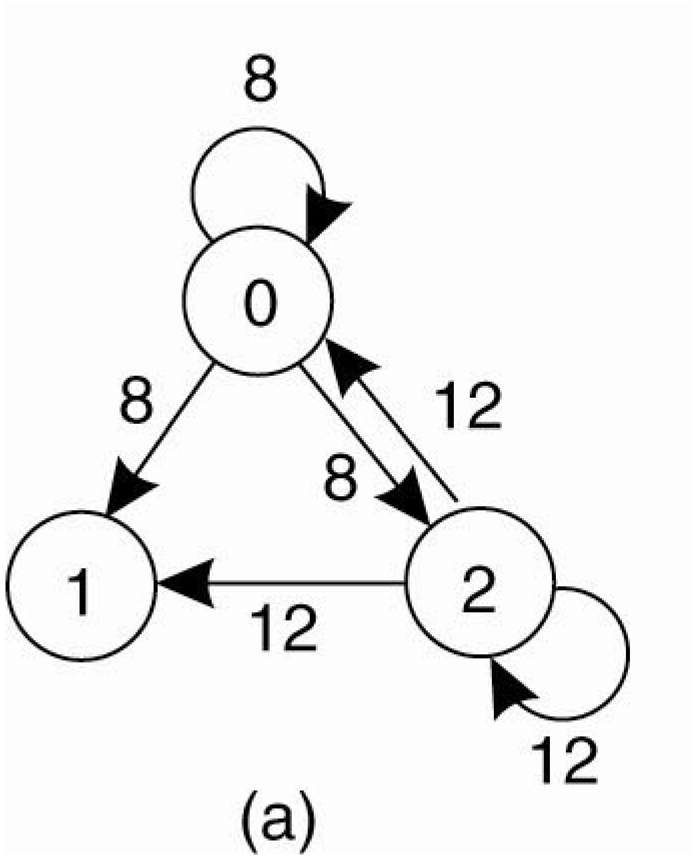


Figure 6-15. (a) Two processes want to access a shared resource at the same moment.

A Distributed Algorithm (3)

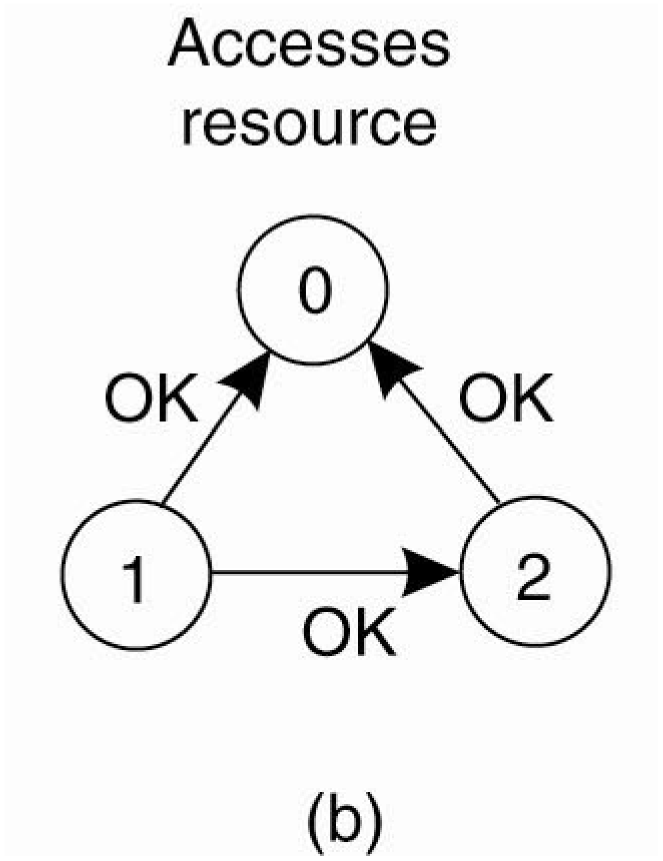


Figure 6-15. (b) Process 0 has the lowest timestamp, so it wins.

A Distributed Algorithm (4)

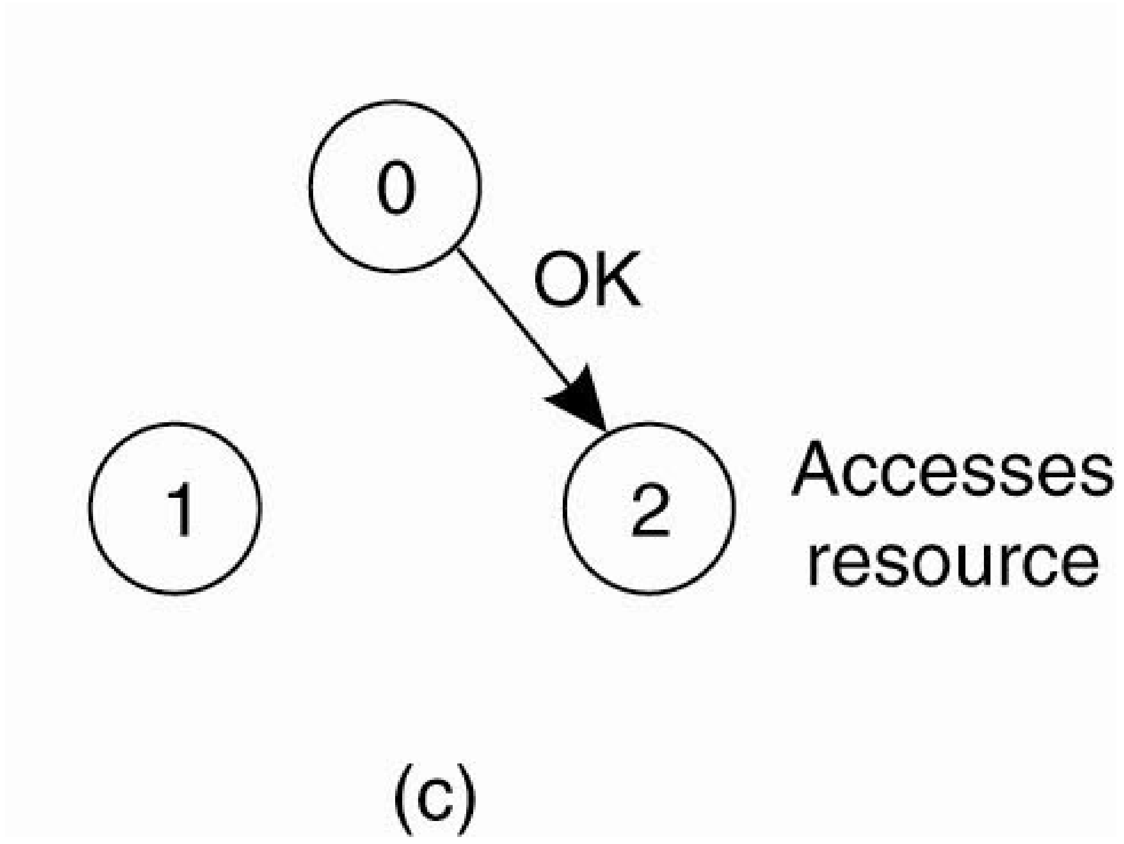


Figure 6-15. (c) When process 0 is done,
it sends an OK also, so 2 can now go ahead.

A Token Ring Algorithm

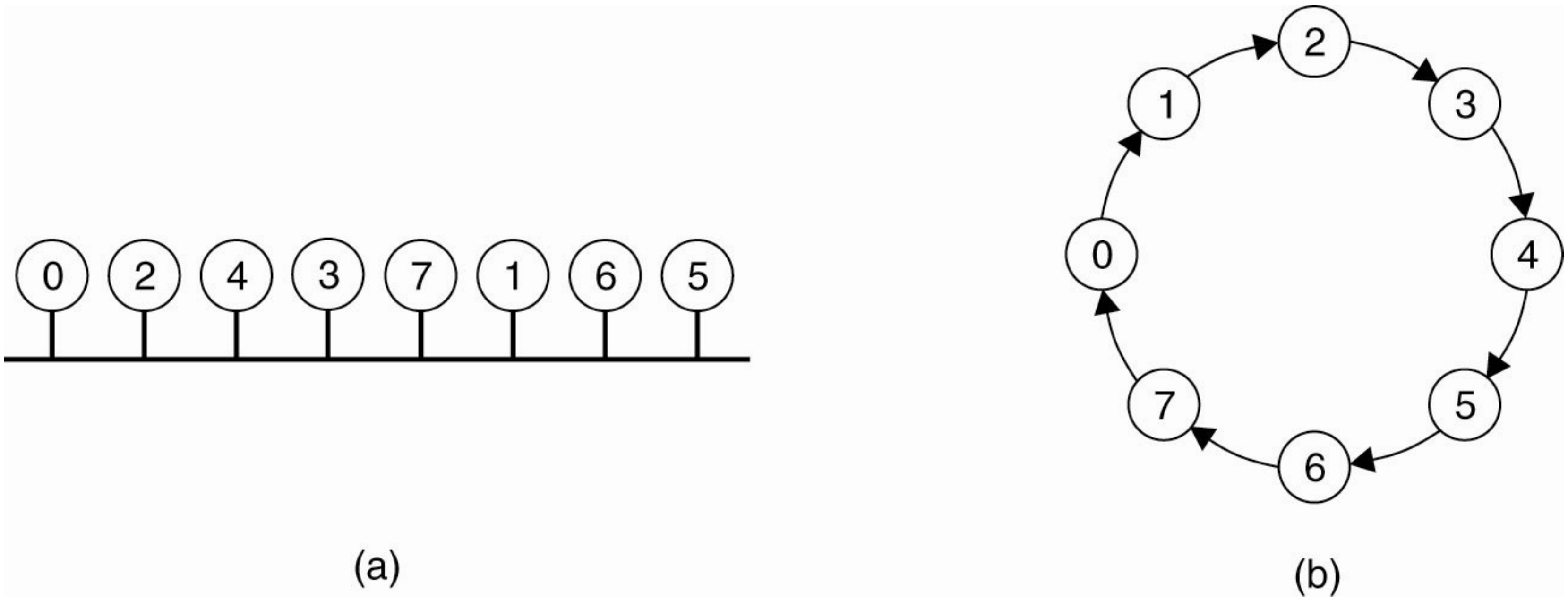
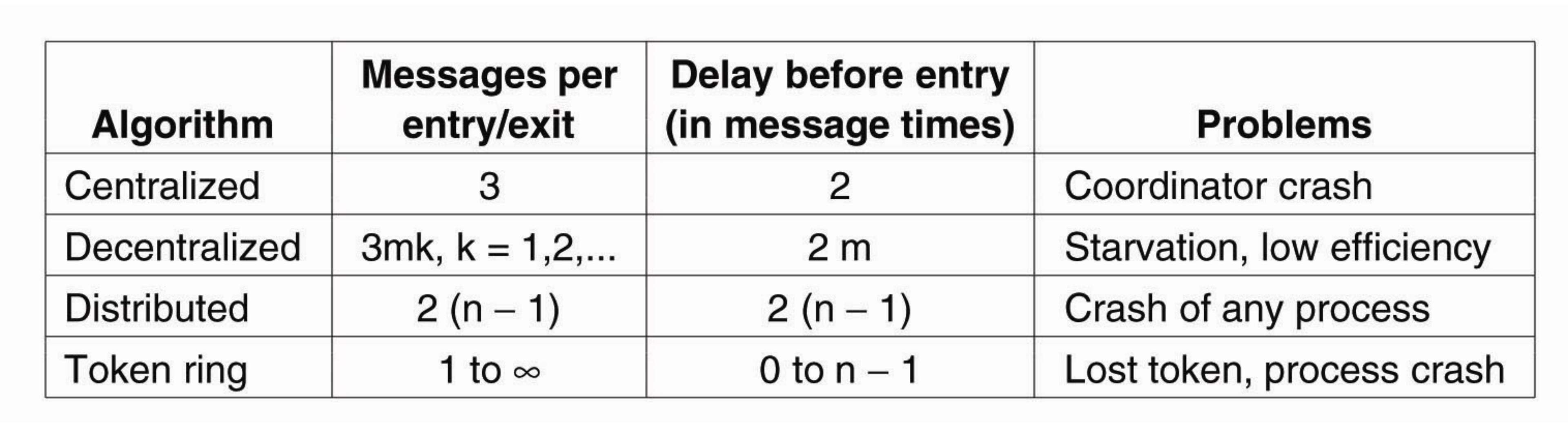


Figure 6-16. (a) An unordered group of processes on a network. (b) A logical ring constructed in software.

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
- A Ring Algorithm
- Elections in large-scale systems

Election Algorithms

The Bully Algorithm

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

The Bully Algorithm (1)

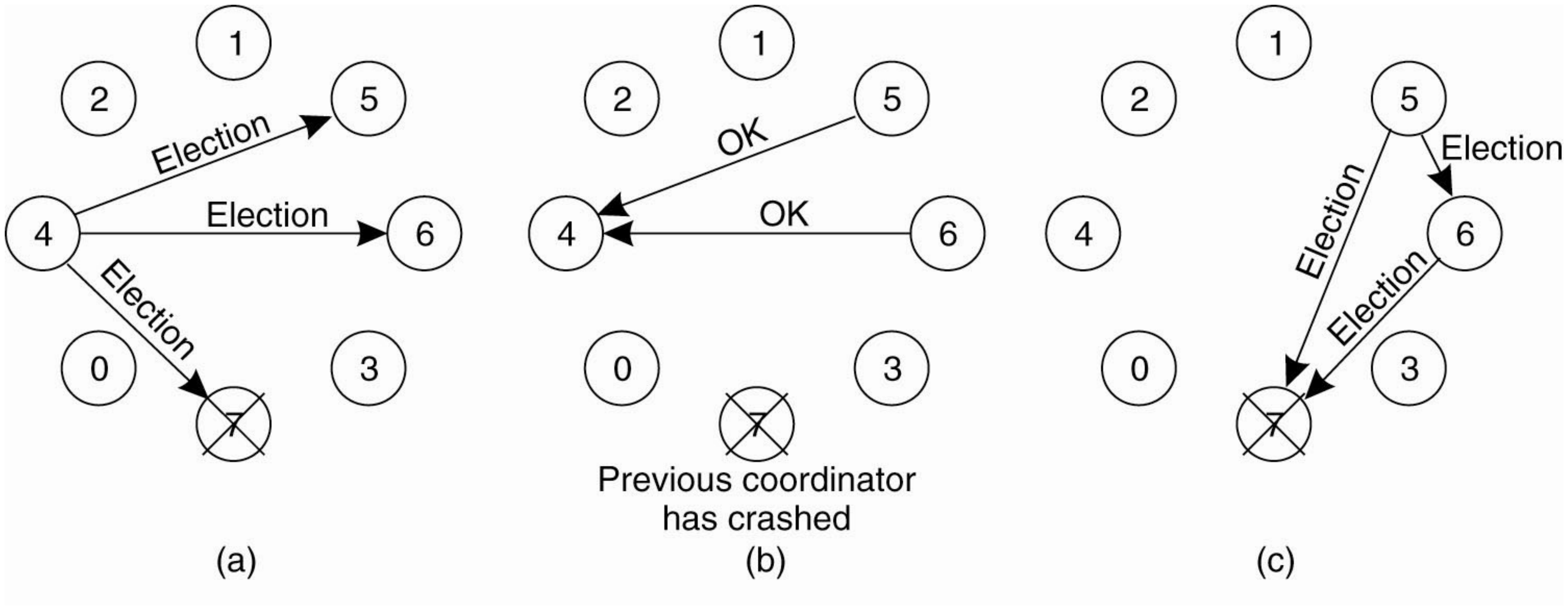


Figure 6-20. The bully election algorithm. (a) Process 4 holds an election. (b) Processes 5 and 6 respond, telling 4 to stop. (c) Now 5 and 6 each hold an election.

The Bully Algorithm (2)

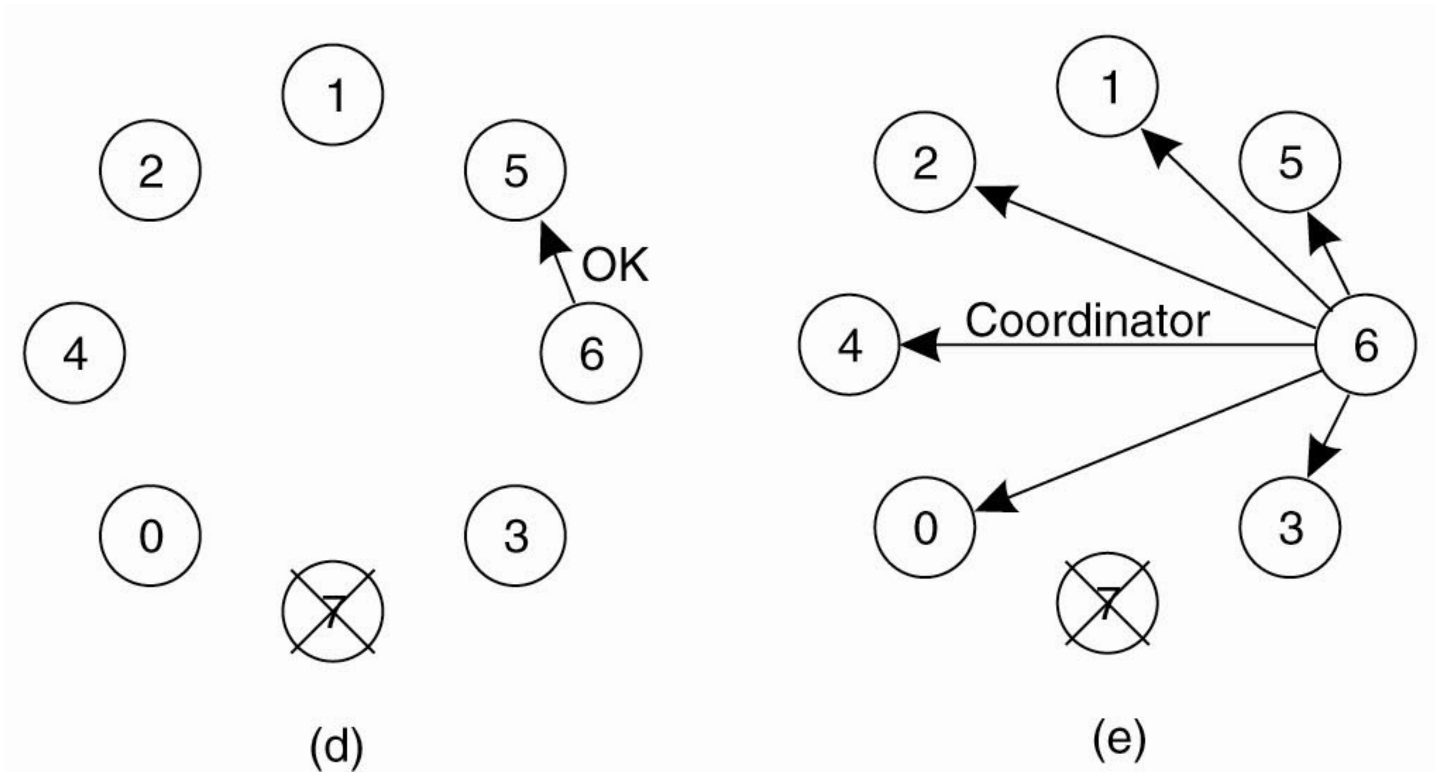


Figure 6-20. The bully election algorithm. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone.

A Ring Algorithm

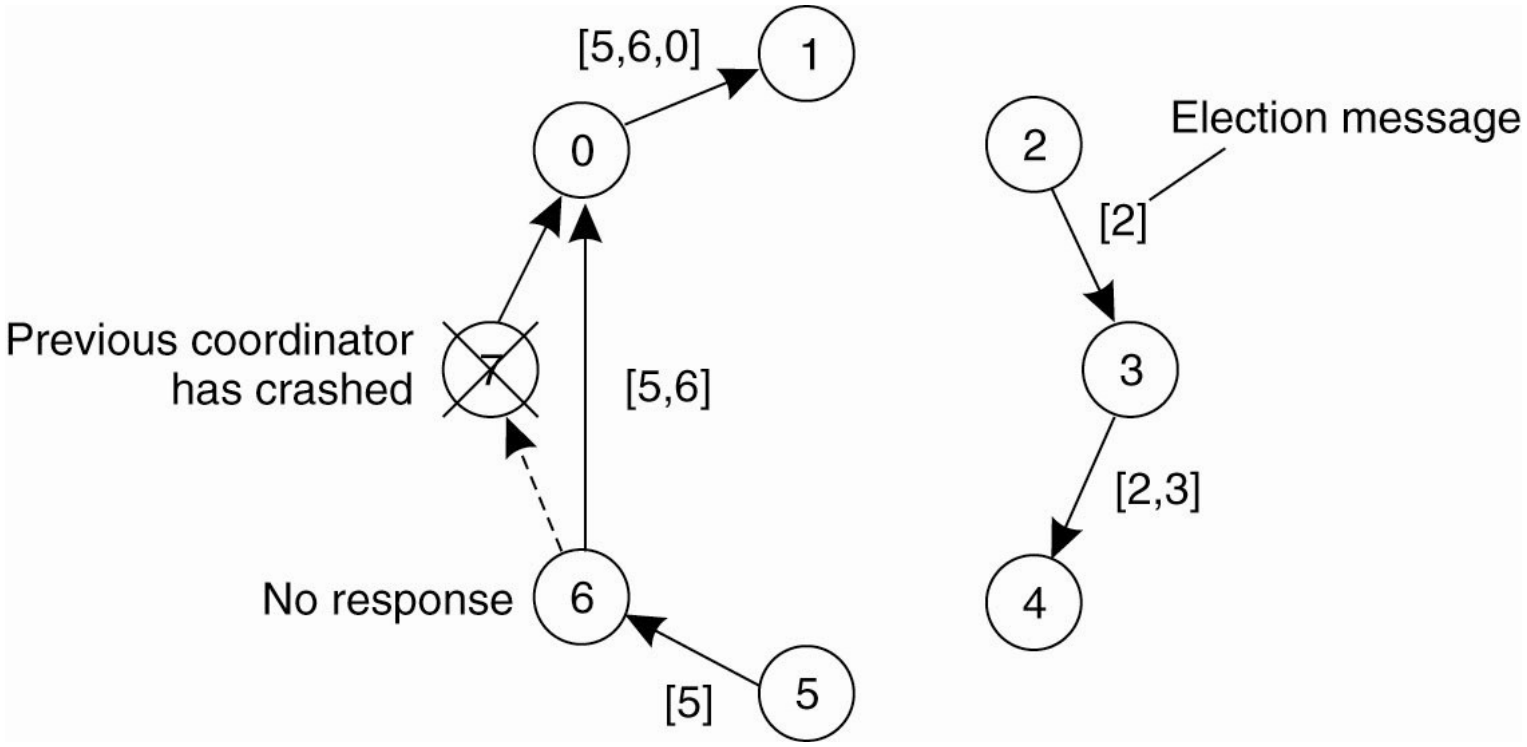


Figure 6-21. Election algorithm using a ring.

Elections in Wireless Environments (1)

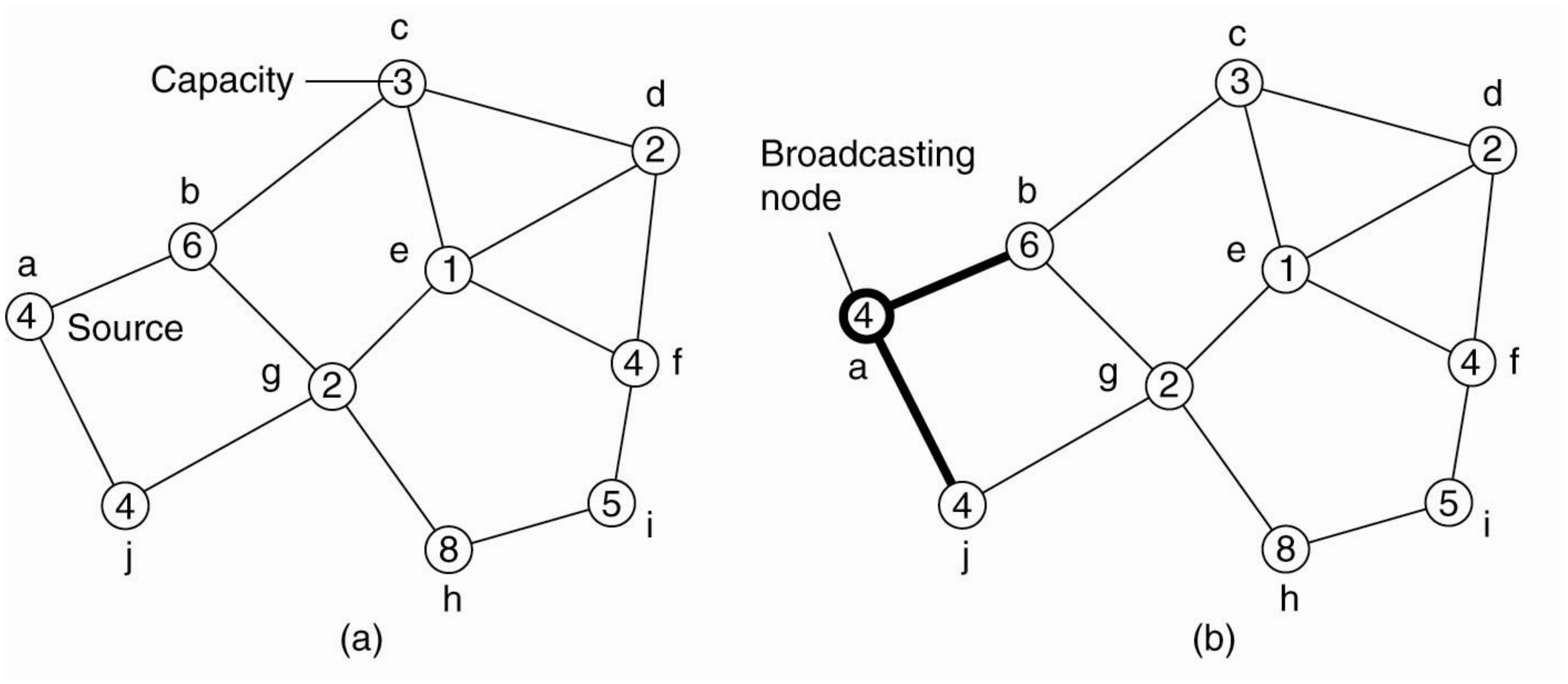


Figure 6-22. Election algorithm in a wireless network, with node a as the source. (a) Initial network. (b)–(e) The build-tree phase

Elections in Wireless Environments (2)

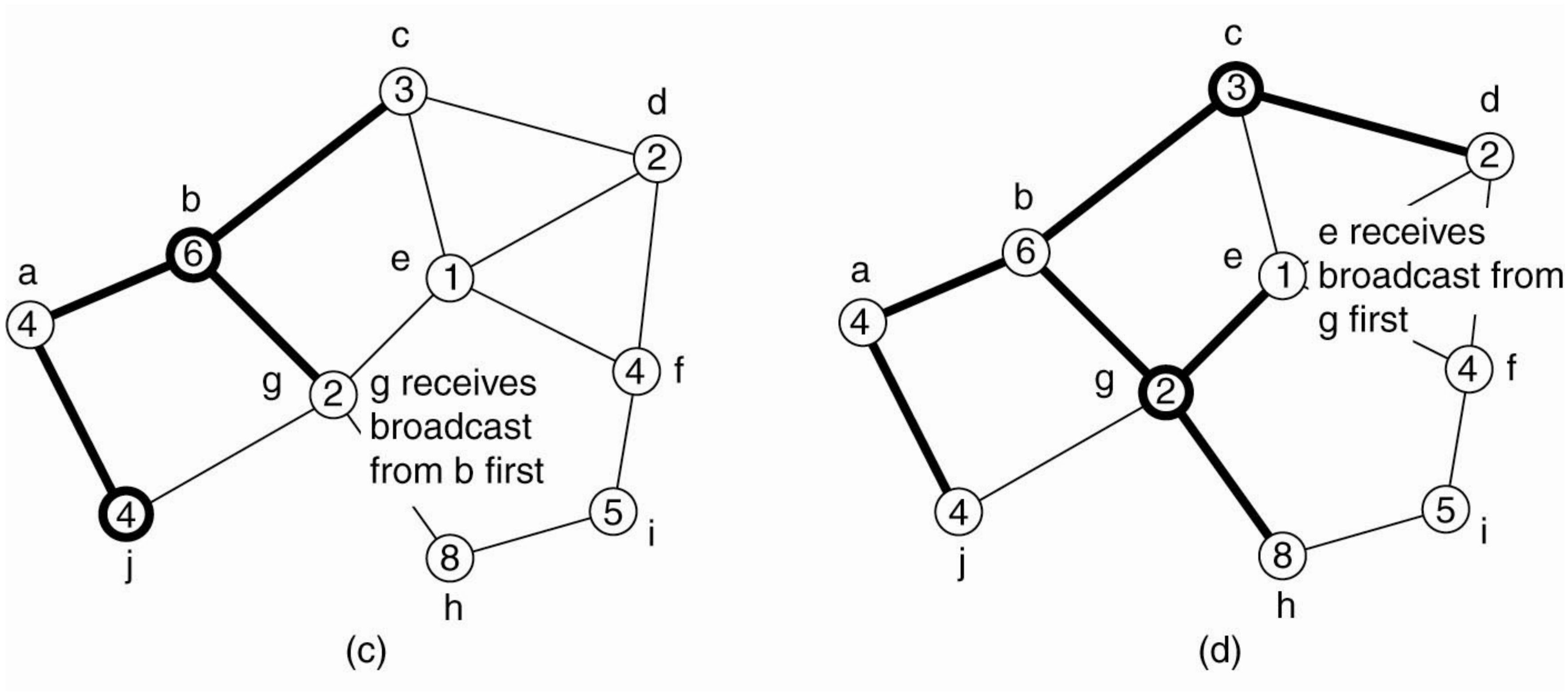
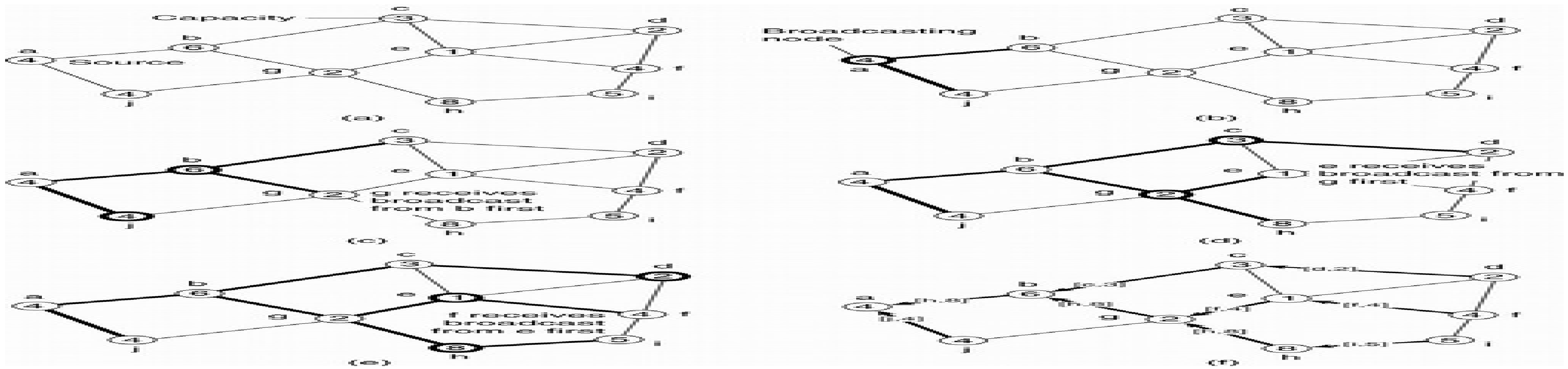

c
3
d
2
b
6
e
1
a
4
4
f
g
2
g receives broadcast from b first
4
j
8
5
i
h
(c)
c
3
d
2
b
6
e receives broadcast from g first
e
1
a
4
4
f
g
2
4
j
8
5
i
h
(d)

Figure 6-22. Election algorithm in a wireless network, with node a as the source. (a) Initial network. (b)–(e) The build-tree phase

Elections in Wireless Environments (3)



Capacity
Source
Broadcasting node
g receives broadcast from b first
e receives broadcast from g first
f receives broadcast from e first
(a)
(b)
(c)
(d)
(e)
(f)

Figure 6-22. (e) The build-tree phase.
(f) Reporting of best node to source.

Elections in Large-Scale Systems (1)

Requirements for superpeer selection:

1. Normal nodes should have low-latency access to superpeers.
2. Superpeers should be evenly distributed across the overlay network.
3. There should be a predefined portion of superpeers relative to the total number of nodes in the overlay network.
4. Each superpeer should not need to serve more than a fixed number of normal nodes.

Elections in Large-Scale Systems (2)

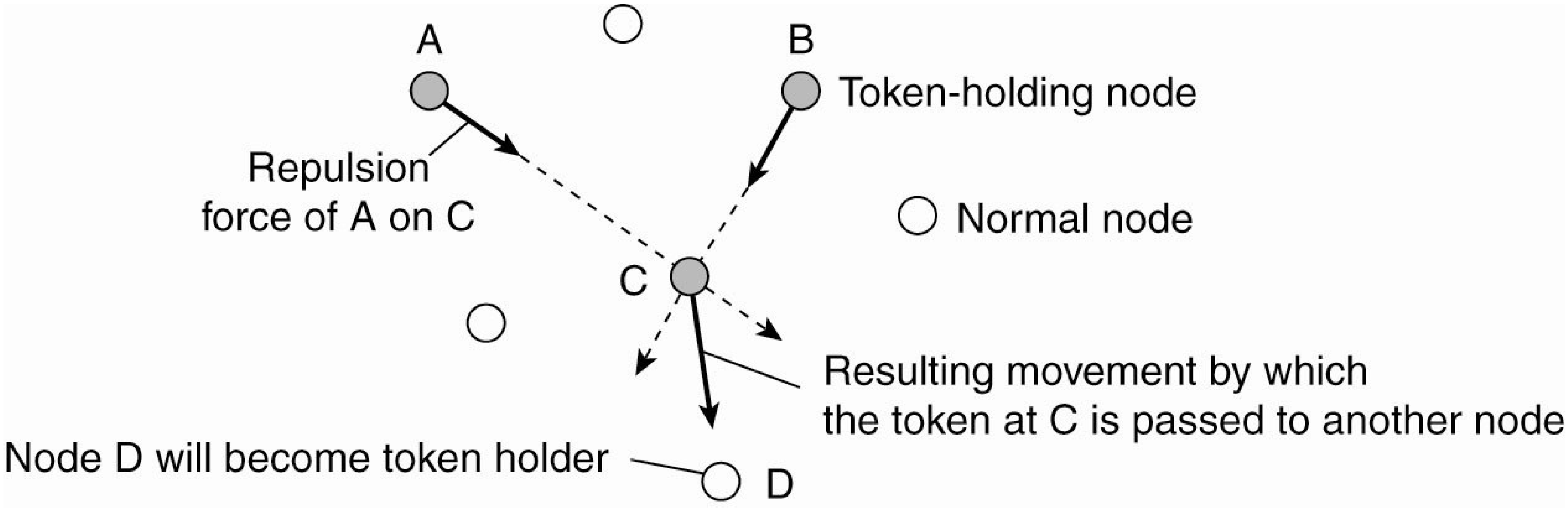

A
B
Token-holding node
Repulsion force of A on C
Normal node
C
Resulting movement by which the token at C is passed to another node
Node D will become token holder
D

Figure 6-23. Moving tokens in a two-dimensional space using repulsion forces.