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

Chapter 6
Coordination

Physical Clocks (1)

- All computers have a circuit to keep track of time using a quartz crystal
- However, quartz crystals at different computers often run at a slightly different speed
  - This leads to the so called clock skew between different machines
- Some systems (e.g., real-time systems) need external physical clock
  - Solar day: interval between two consecutive noons
  - Solar day varies due to many reasons
- Universal Coordinated Time (UTC): transitions of cesium 133 atom (pretty accurate).
  - Cannot be directly used as everyday clock. UTC second < Solar second
  - Solution: leap second whenever the difference is 800msec -> UTC

Clock Synchronization

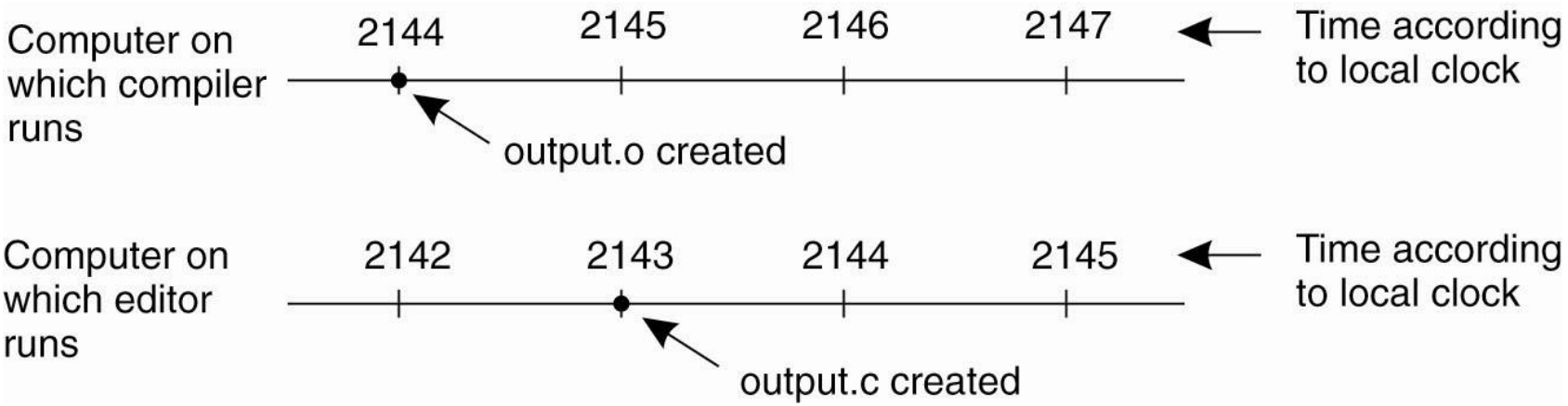


Figure 6-1. When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

Clock Synchronization

Precision

The goal is to keep the deviation between two clocks on any two machines within a specified bound, known as the precision π:

$$\forall t, \forall p, q : |C_p(t) - C_q(t)| \leq \pi$$

with $C_p(t)$ the computed clock time of machine p at UTC time t.

Accuracy

In the case of accuracy, we aim to keep the clock bound to a value α:

$$\forall t, \forall p : |C_p(t) - t| \leq \alpha$$

Synchronization

- Internal synchronization: keep clocks precise
- External synchronization: keep clocks accurate

Clock Drift

Clock specifications

- A clock comes specified with its maximum clock drift rate ρ.
- $F(t)$ denotes oscillator frequency of the hardware clock at time t
- F is the clock's ideal (constant) frequency $\Rightarrow$ living up to specifications:

$$\forall t : (1-\rho) \leq \frac{F(t)}{F} \leq (1+\rho)$$

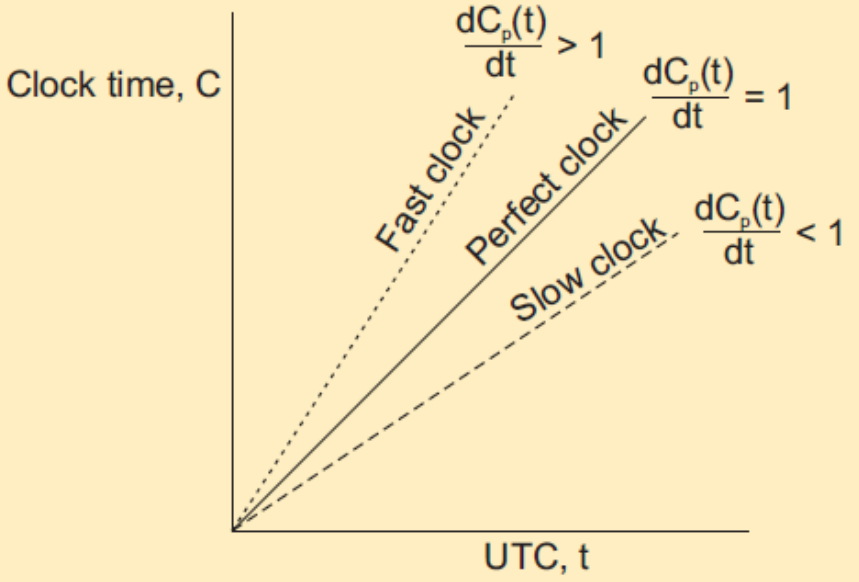
Observation

By using hardware interrupts we couple a software clock to the hardware clock, and thus also its clock drift rate:

$$C_p(t) = \frac{1}{F}\int_0^t F(t)dt \Rightarrow \frac{dC_p(t)}{dt} = \frac{F(t)}{F}$$

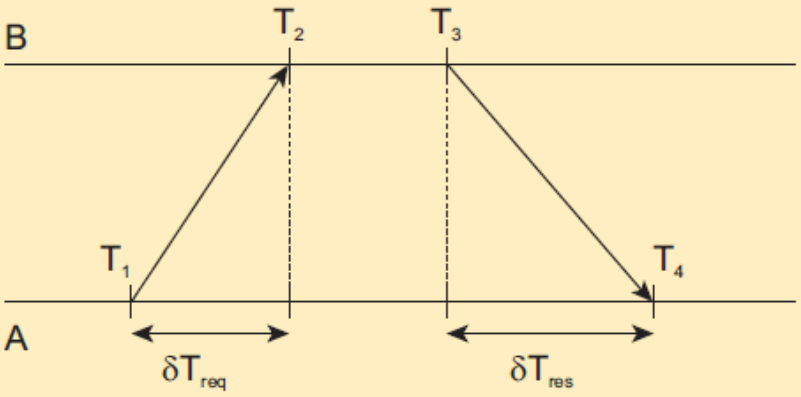
$$\Rightarrow \forall t : 1-\rho \leq \frac{dC_p(t)}{dt} \leq 1+\rho$$

Fast, perfect, slow clocks



Detecting and adjusting incorrect times

Getting the current time from a time server



Computing the relative offset θ and delay δ

Assumption: $\delta T_{req} = T_2 - T_1 \approx T_4 - T_3 = \delta T_{res}$

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

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

Network Time Protocol

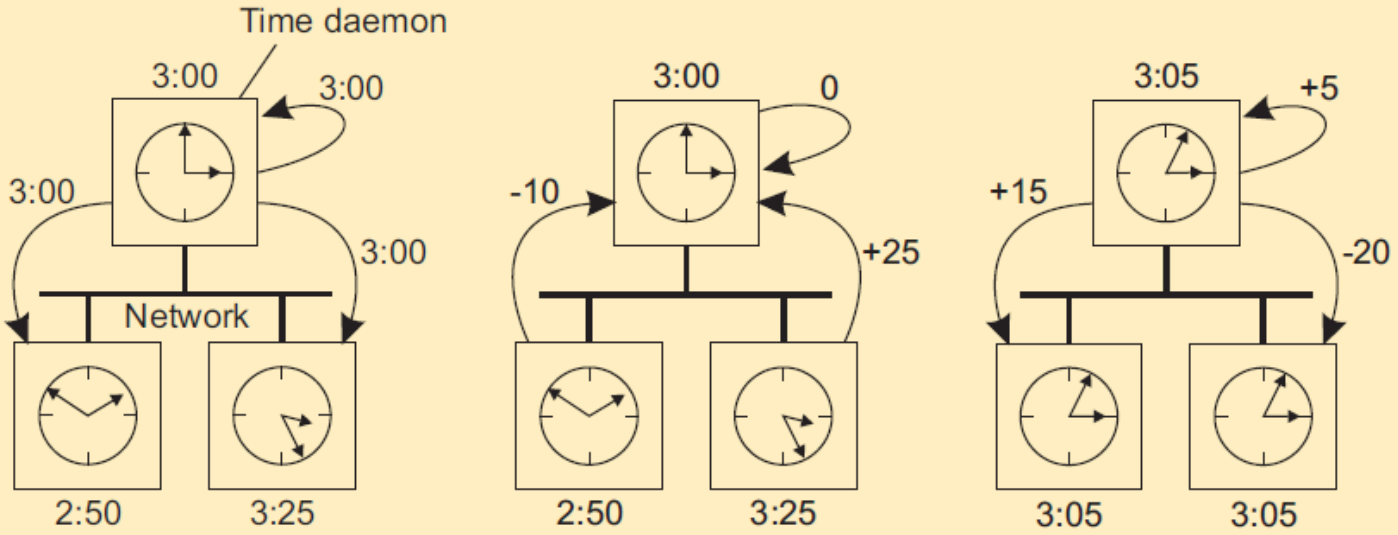
Collect eight (θ, δ) pairs and choose θ for which associated delay δ was minimal.

Keeping time without UTC

Principle

Let the time server scan all machines periodically, calculate an average, and inform each machine how it should adjust its time relative to its present time.

Using a time server



Fundamental

You'll have to take into account that setting the time back is never allowed ⇒ smooth adjustments (i.e., run faster or slower).

Clock Synchronization in Wireless Networks

In traditional distributed systems, we can deploy many time servers

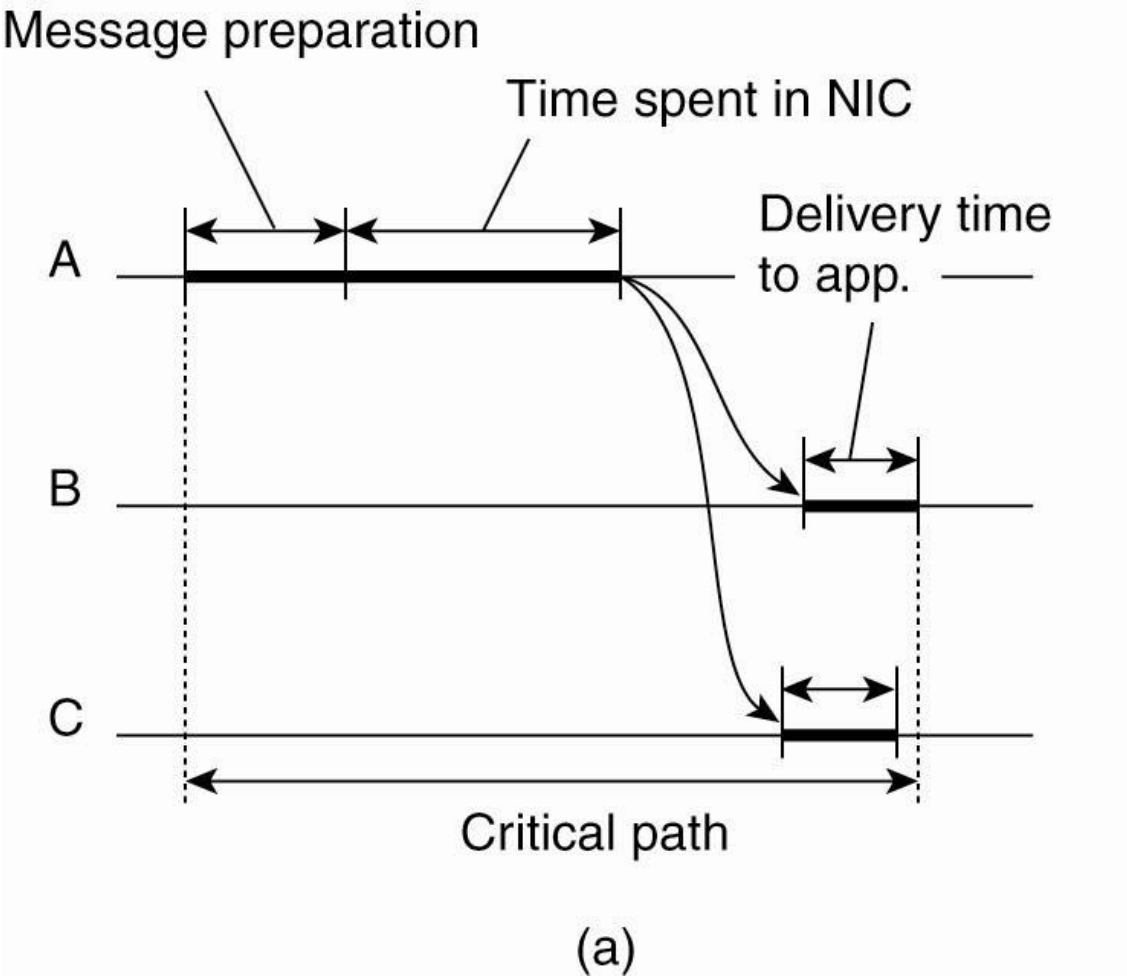
– that can easily contact each other for efficient information dissemination

- However, in wireless networks, communication becomes expensive and unreliable
- RBS (Reference Broadcast Synchronization) is a clock synchronization protocol

  – where a sender broadcasts a reference message that will allow its receivers to adjust their clocks

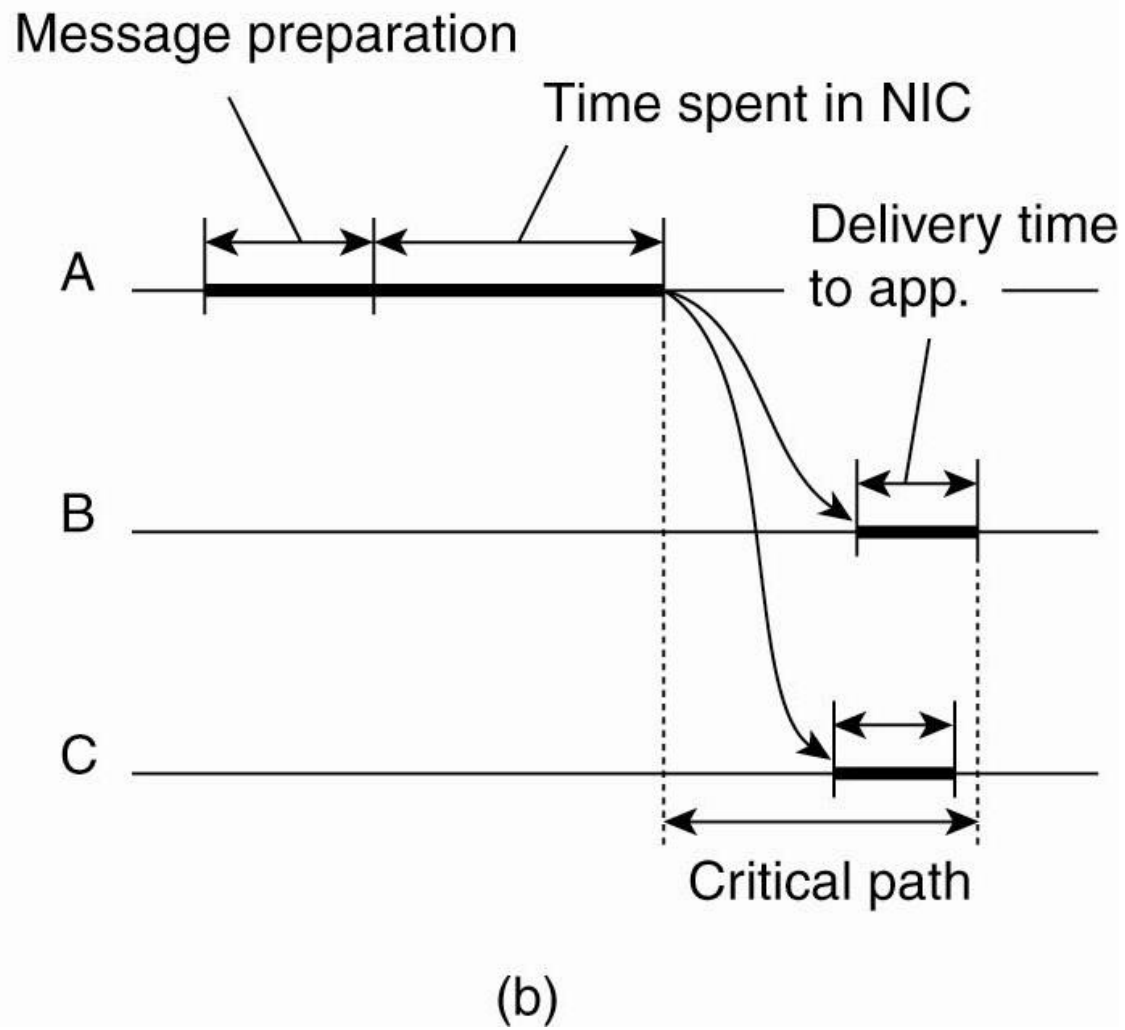
Reference Broadcast Synchronization

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



Reference Broadcast Synchronization(2)

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



Lamport's Logical Clocks

The "happens-before" relation $\rightarrow$ can be observed directly in two situations:

- If *a* and *b* are events in the same process, and *a* occurs before *b*, then $a \rightarrow b$ 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 $a \rightarrow b$
- This introduces a partial ordering of events in a system with concurrently operating processes.

Logical Clock

Problem

How do we maintain a global view on the system's behavior that is consistent with the happened-before relation?

Attach a timestamp $C(e)$ to each event e, satisfying the following properties:

- P1 If a and b are two events in the same process, and $a \rightarrow b$, then we demand that $C(a) < C(b)$.
- P2 If a corresponds to sending a message m, and b to the receipt of that message, then also $C(a) < C(b)$.

Problem

How to attach a timestamp to an event when there's no global clock $\Rightarrow$ maintain a consistent set of logical clocks, one per process.

Logical Clocks: solution

Each process P_i maintains a local counter C_i and adjusts this counter

1. For each new event that takes 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.

Notes

- Property P1 is satisfied by (1); Property P2 by (2) and (3).
- It can still occur that two events happen at the same time. Avoid this by breaking ties through process IDs.

Lamport's Logical Clocks (2)

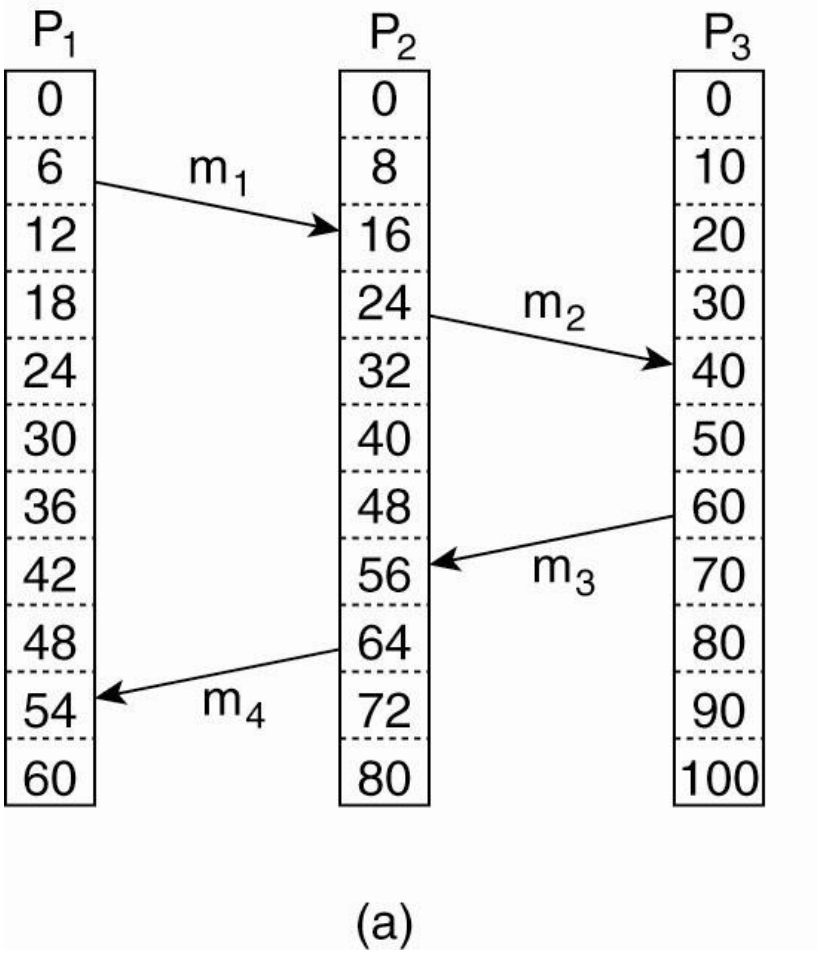


Figure 6-9. (a) Three processes, each with its own clock. The clocks run at different rates.

Lamport's Logical Clocks (3)

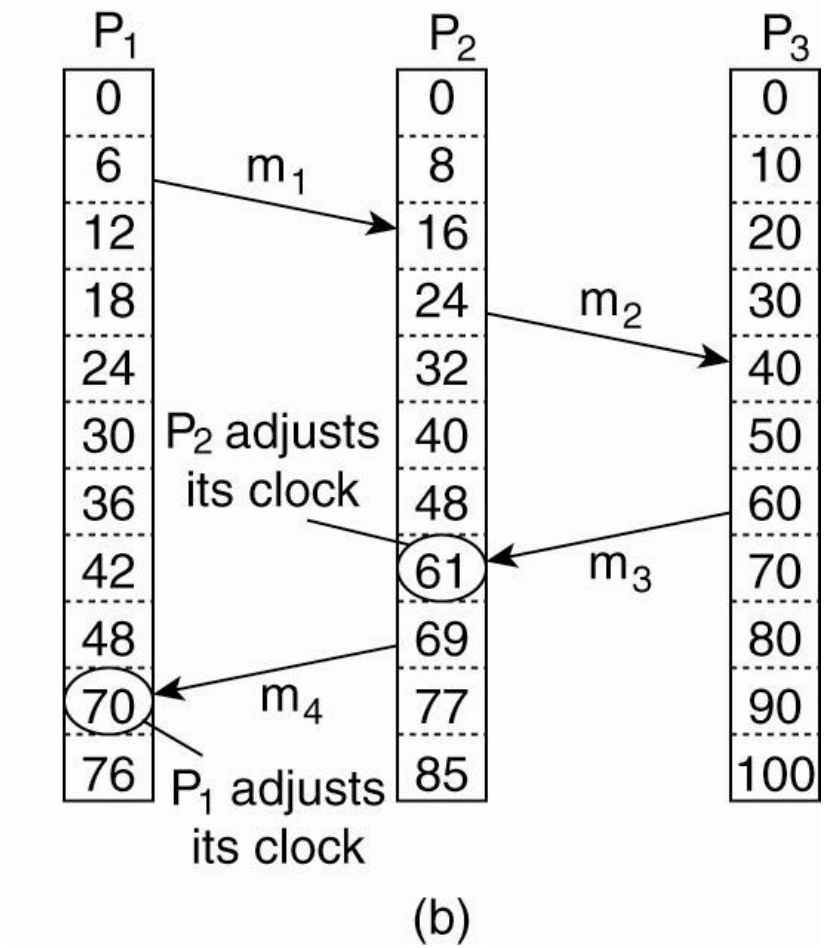


Figure 6-9. (b) Lamport's algorithm corrects the clocks.

Lamport's Logical Clocks (4)

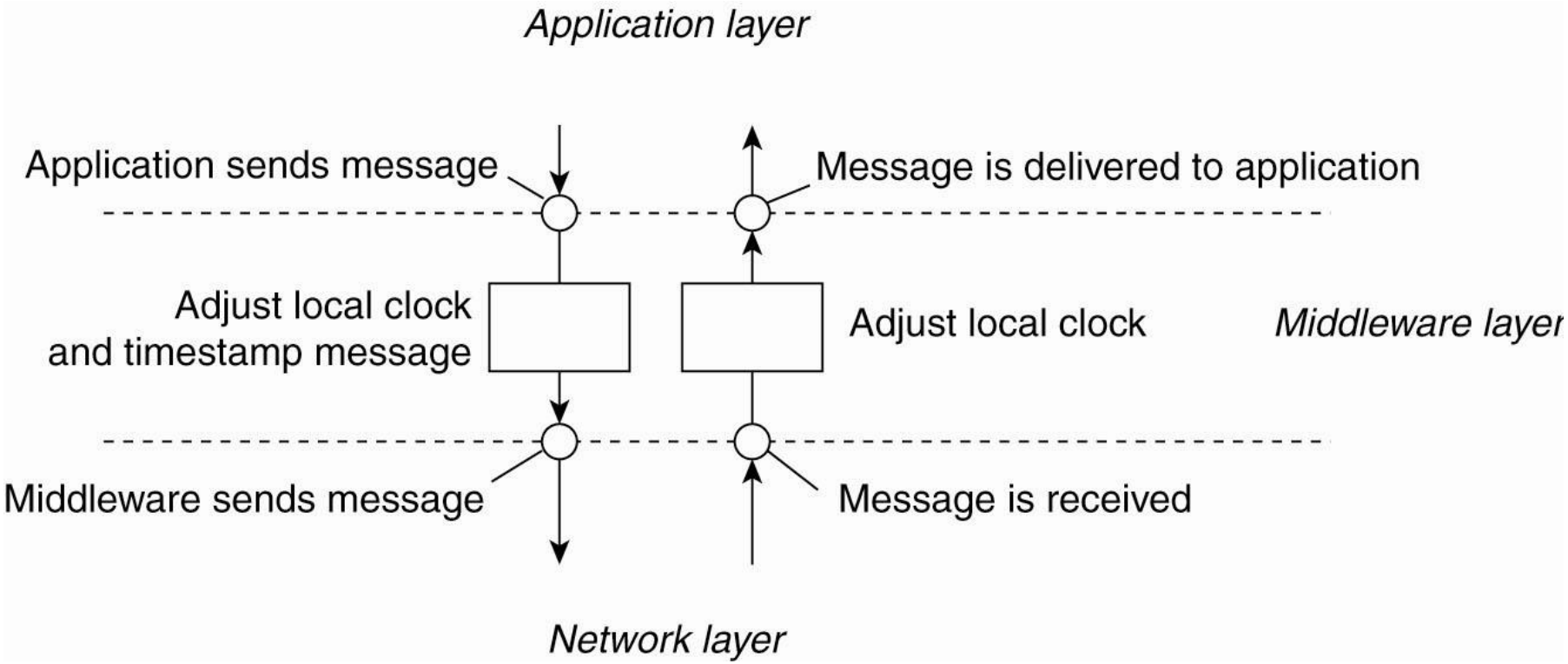
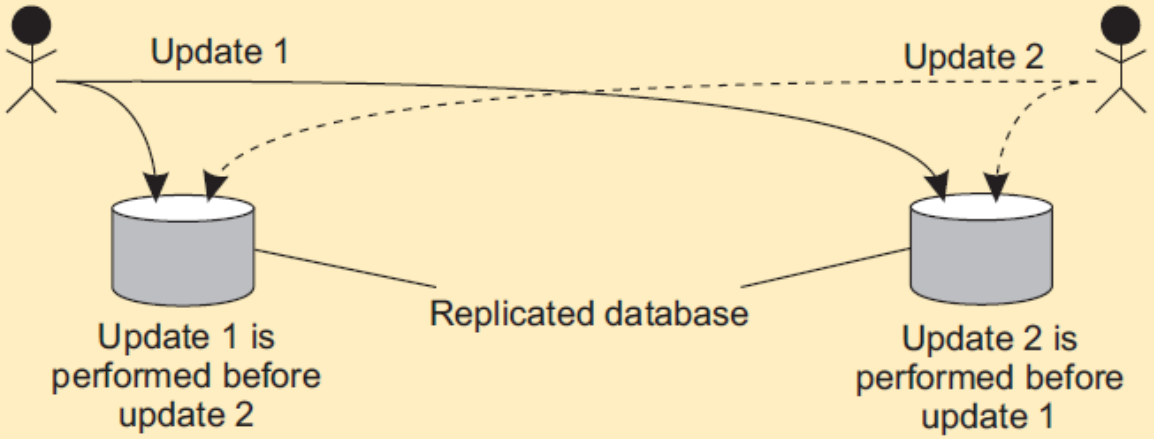


Figure 6-10. The positioning of Lamport's logical clocks in distributed systems.

Example: Totally Ordered Multicasting

Concurrent updates on a replicated database are seen in the same order everywhere

- P_1 adds \$100 to an account (initial value: \$1000)
- P_2 increments account by 1%
- There are two replicas



Result

In absence of proper synchronization:
replica #1 ← \$1111, while replica #2 ← \$1110.

Example: Total-ordered multicast

Solution

- Process P_i sends timestamped message m_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 m_i to its application if:

(1) m_i is at the head of $queue_j$
(2) for each process P_k, there is a message m_k in $queue_j$ with a larger timestamp.

Note

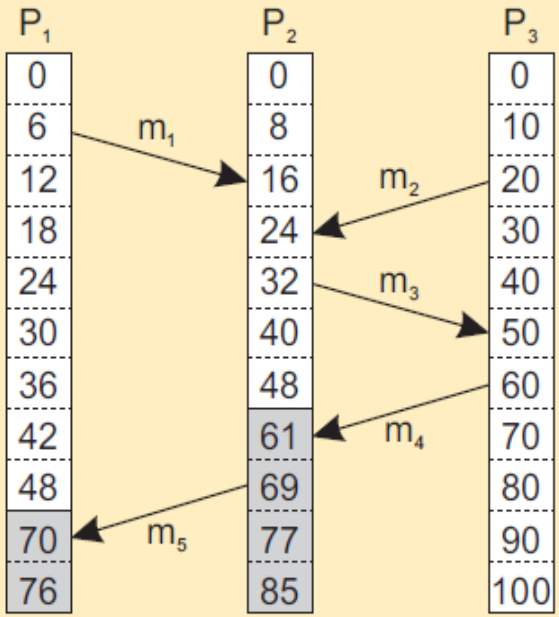
We are assuming that communication is reliable and FIFO ordered.

Vector Clocks (1)

Observation

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

Concurrent message transmission using logical clocks



Observation

Event a: m_1 is received at $T = 16$;
Event b: m_2 is sent at $T = 20$.

Note

We cannot conclude that a causally precedes b.

Causal dependency

Definition

We say that b may causally depend on a if $ts(a) < ts(b)$, with:

- for all k, $ts(a)[k] \leq ts(b)[k]$ and
- there exists at least one index k' for which $ts(a)[k'] < ts(b)[k']$

Precedence vs. dependency

- We say that a causally precedes b.
- b may causally depend on a, as there may be information from a that is propagated into b.

Vector Clocks (2)

Solution: each P_i maintains a vector VC_i

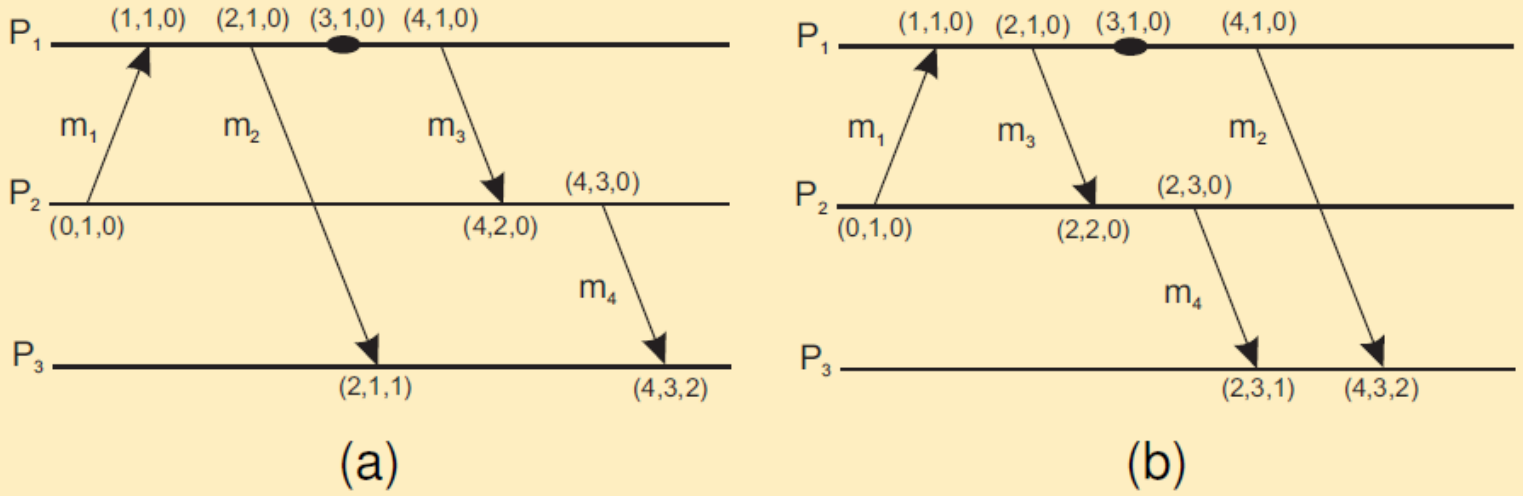
- $VC_i[i]$ is the local logical clock at process P_i.
- If $VC_i[j] = k$ then P_i knows that k events have occurred at P_j.

Maintaining vector clocks

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 step 1.
3. Upon the receipt of a message m, process P_j sets $VC_j[k] \leftarrow \max\{VC_j[k], ts(m)[k]\}$ for each k, after which it executes step 1 and then delivers the message to the application.

Vector clocks: Example

Capturing potential causality when exchanging messages



Analysis

Situation	$ts(m_2)$	$ts(m_4)$	$ts(m_2) < ts(m_4)$	$ts(m_2) > ts(m_4)$	**Conclusion**
(a)	(2,1,0)	(4,3,0)	Yes	No	m_2 may causally precede m_4
(b)	(4,1,0)	(2,3,0)	No	No	m_2 and m_4 may conflict

Causally ordered multicasting

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 $VC_j[j]$).

P_j postpones delivery of m until:

1. $ts(m)[i] = VC_j[i] + 1$
2. $ts(m)[k] \leq VC_j[k]$ for all $k \neq i$

Enforcing Causal Communication

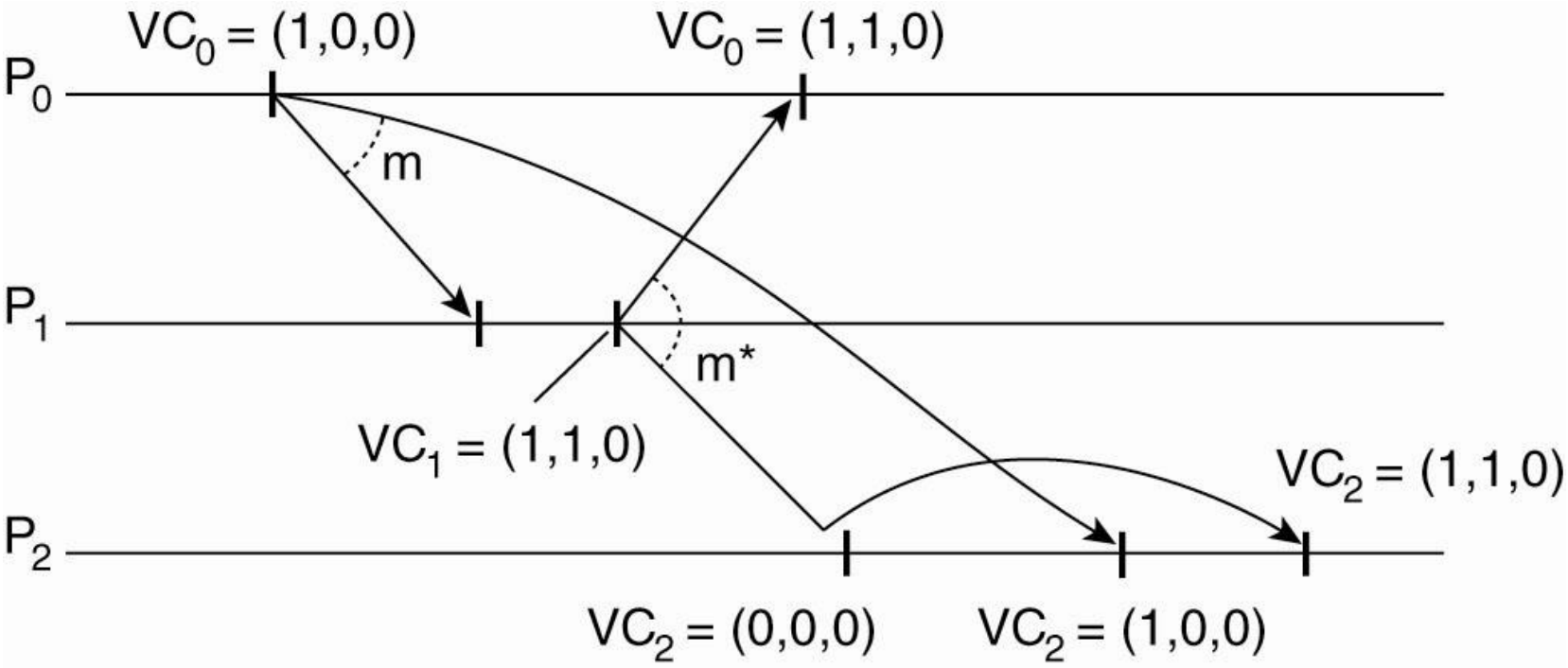


Figure 6-13. Enforcing causal communication.

Mutual Exclusion
A Centralized Algorithm (1)

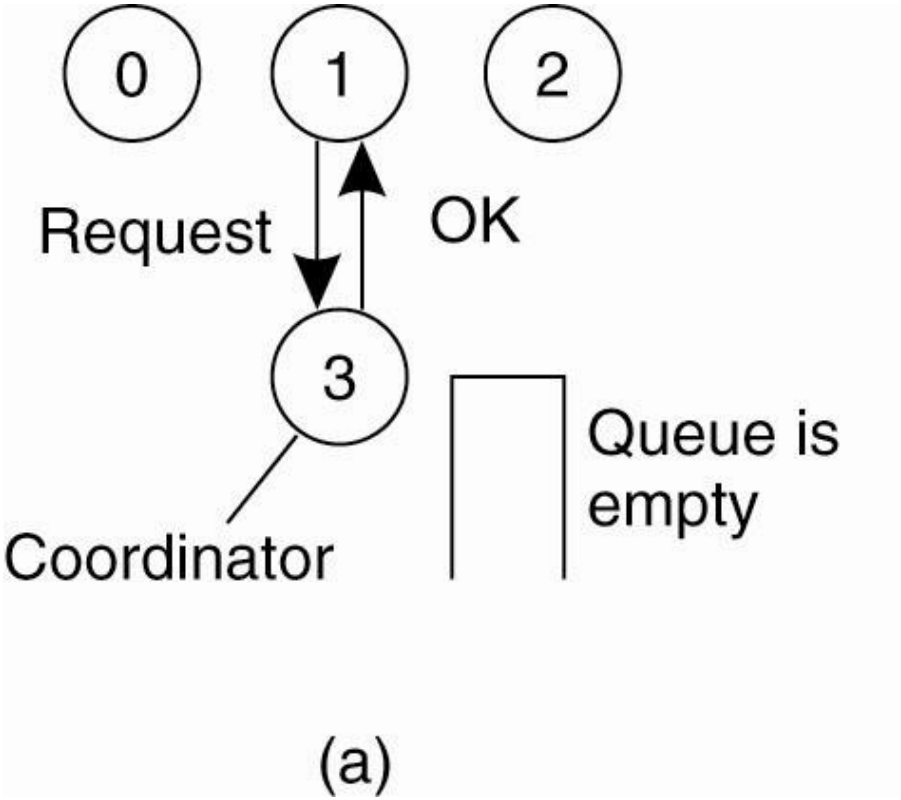


Figure 6-14. (a) Process 1 asks the coordinator for permission to access a hared resource. Permission is granted.

Mutual Exclusion
A Centralized Algorithm (2)

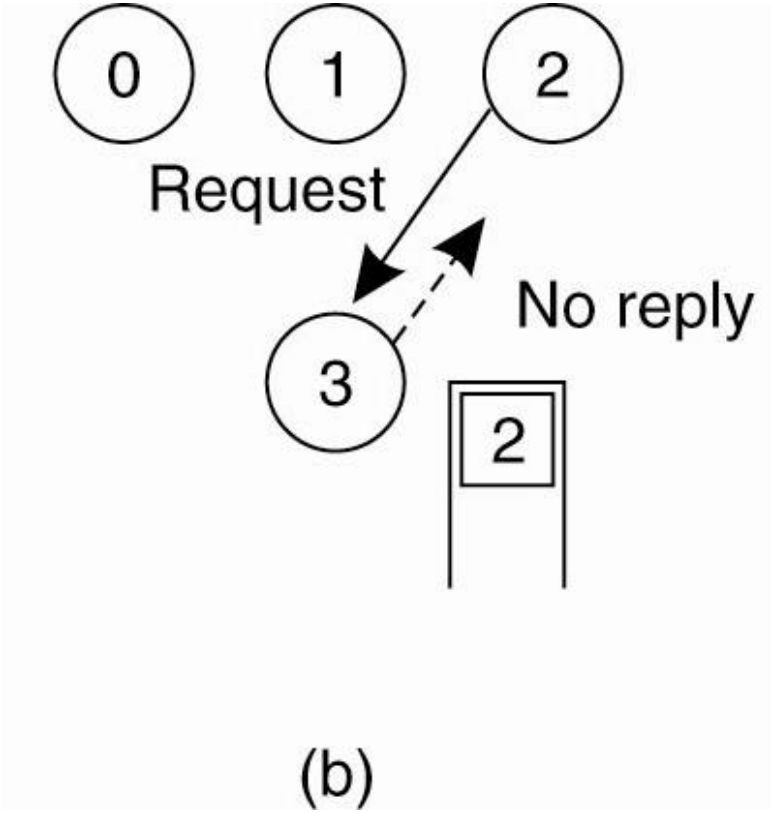


Figure 6-14. (b) Process 2 then asks permission to access the same resource. The coordinator does not reply.

Mutual Exclusion
A Centralized Algorithm (3)

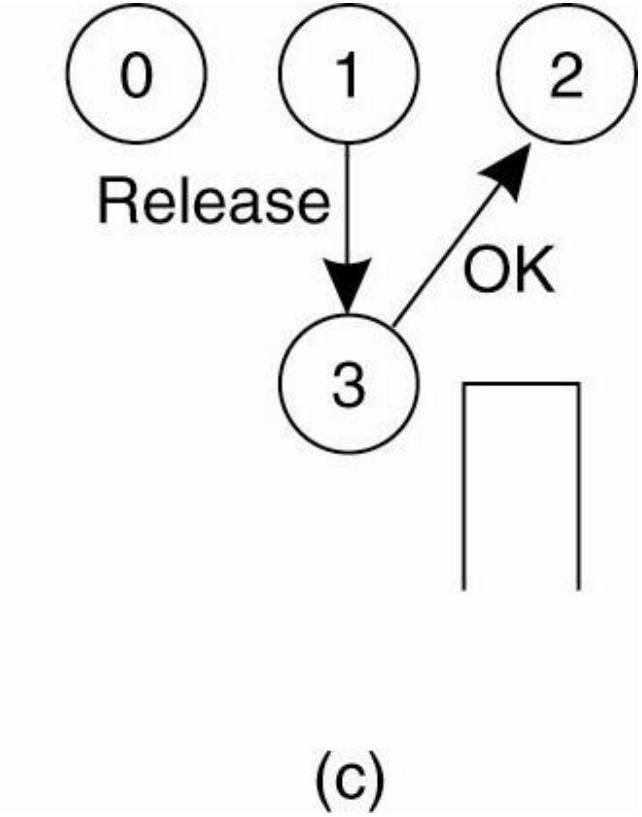


Figure 6-14. (c) When process 1 releases the resource, it tells the coordinator, which then replies to 2.

A Distributed Algorithm Ricart & Agrawala (1)

- When a process wants to access a shared resource, it builds a messages containing:
  - Name of the resource, Its process number, and the current time
  - Then, sends the message to all other processes, even to itself

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.

Ricart & Agrawala (2)

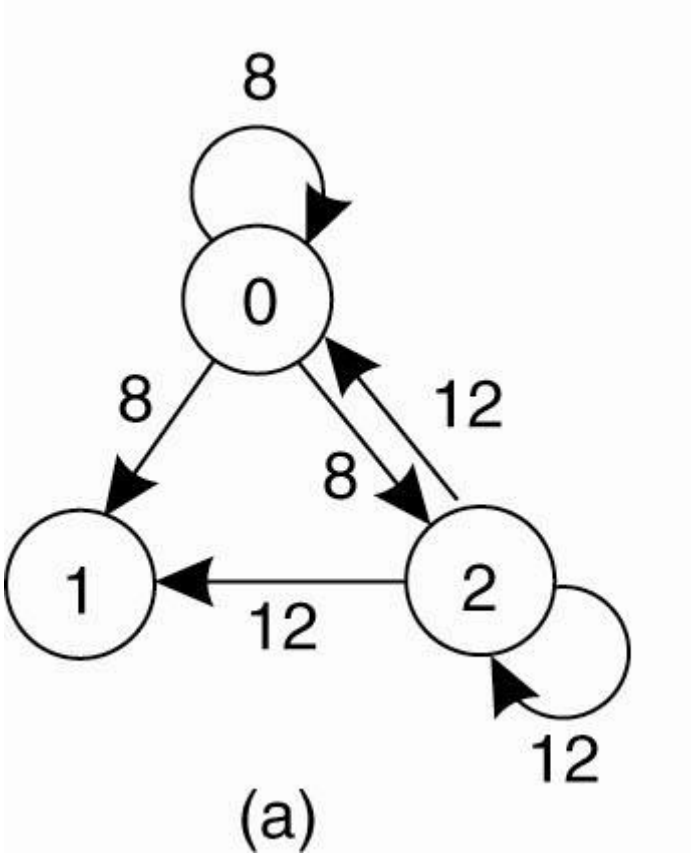


Figure 6-15. (a) Two processes want to access a shared resource at the same moment.

Ricart & Agrawala (3)

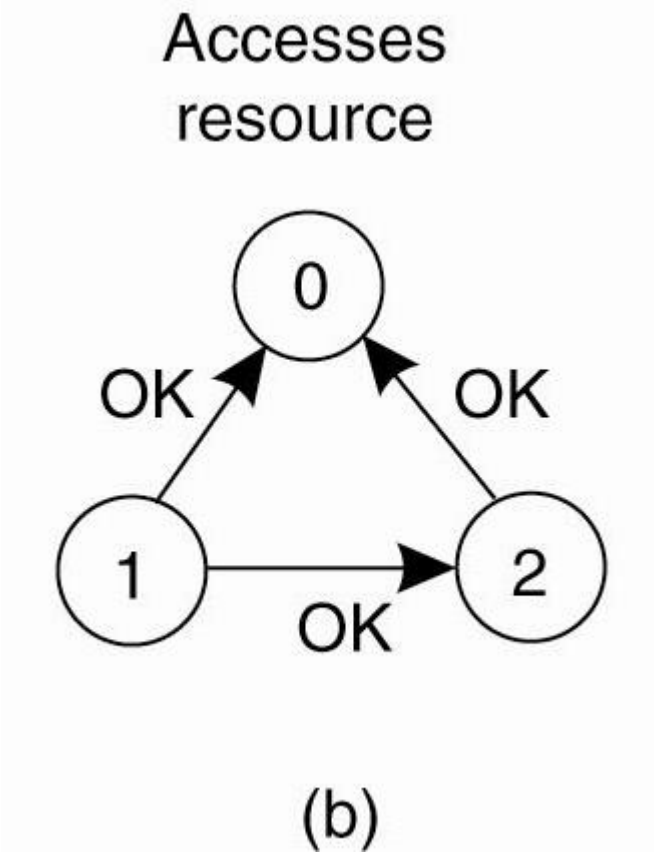


Figure 6-15. (b) Process 0 has the lowest timestamp, so it wins.

Ricart & Agrawala (4)

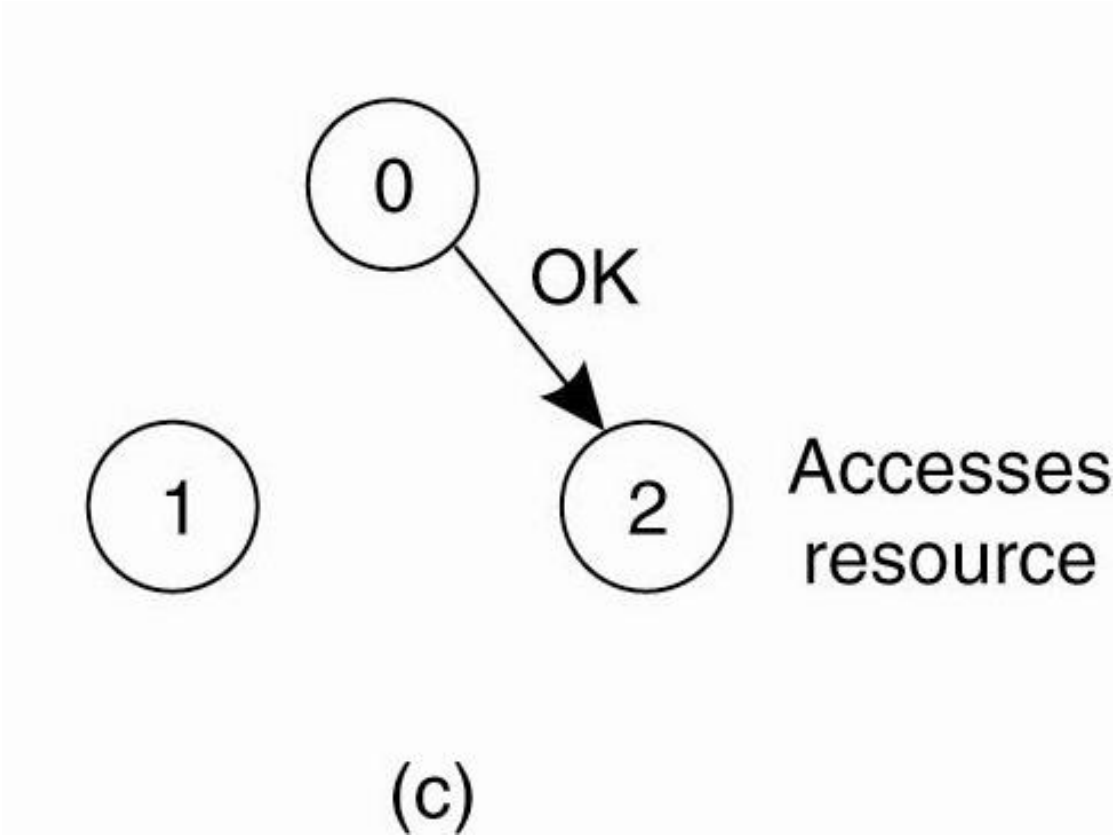


Figure 6-15. (c) When process 0 is done, it sends an OK also, so 2 can now go ahead.

A Token Ring Algorithm

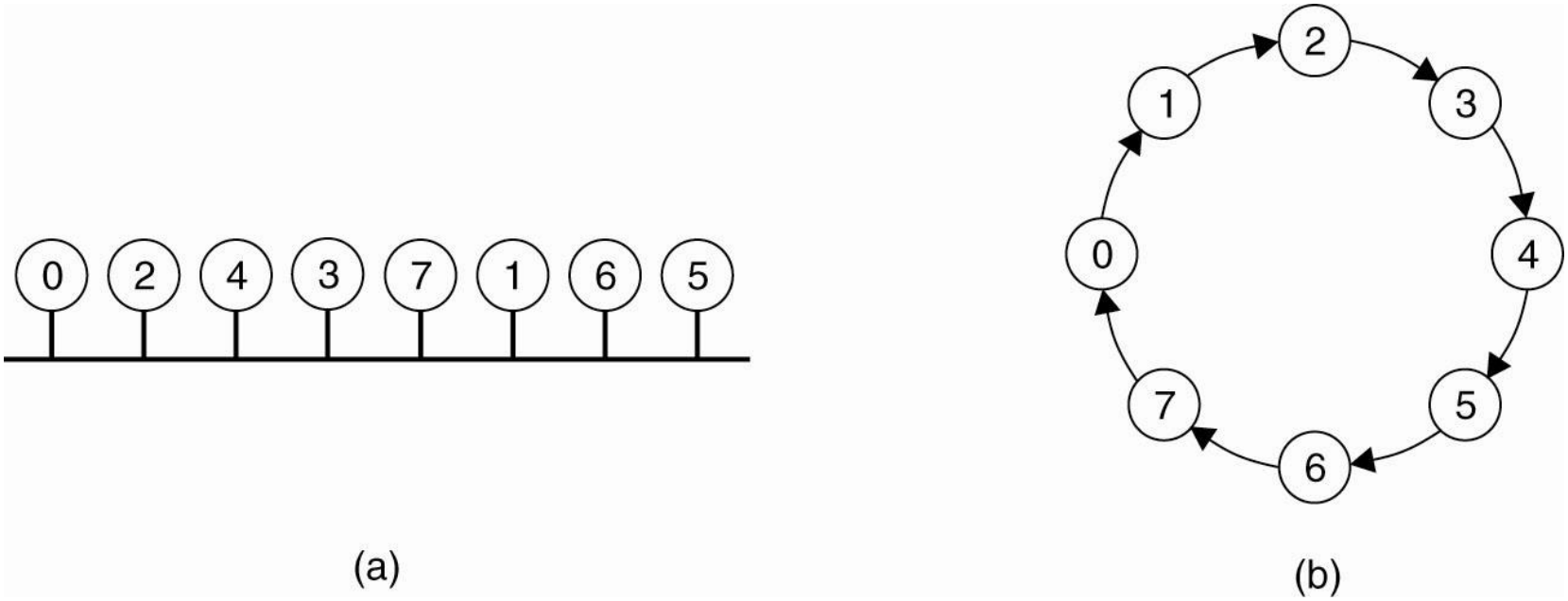


Figure 6-16. (a) An unordered group of processes on a network. (b) A logical ring constructed in software.

Election algorithms

- An algorithm requires that some process acts as a coordinator. The question is how to select this special process **dynamically**.
- In many systems the coordinator is chosen by hand (e.g. file servers). This leads to **centralized** solutions ➔ single point of failure.

- If a coordinator is chosen dynamically, to what extent can we speak about a centralized or distributed solution?
- Is a fully distributed solution, i.e. one without a coordinator, always more robust than any centralized/coordinated solution?

Assumptions

- All processes have unique id's
- All processes know id's of all processes in the system (but not if they are up or down)
- Election means identifying the process with the highest id that is up

Election Algorithms

When a process P notices that the coordinator is no longer responding to requests, it initiates an election:

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)

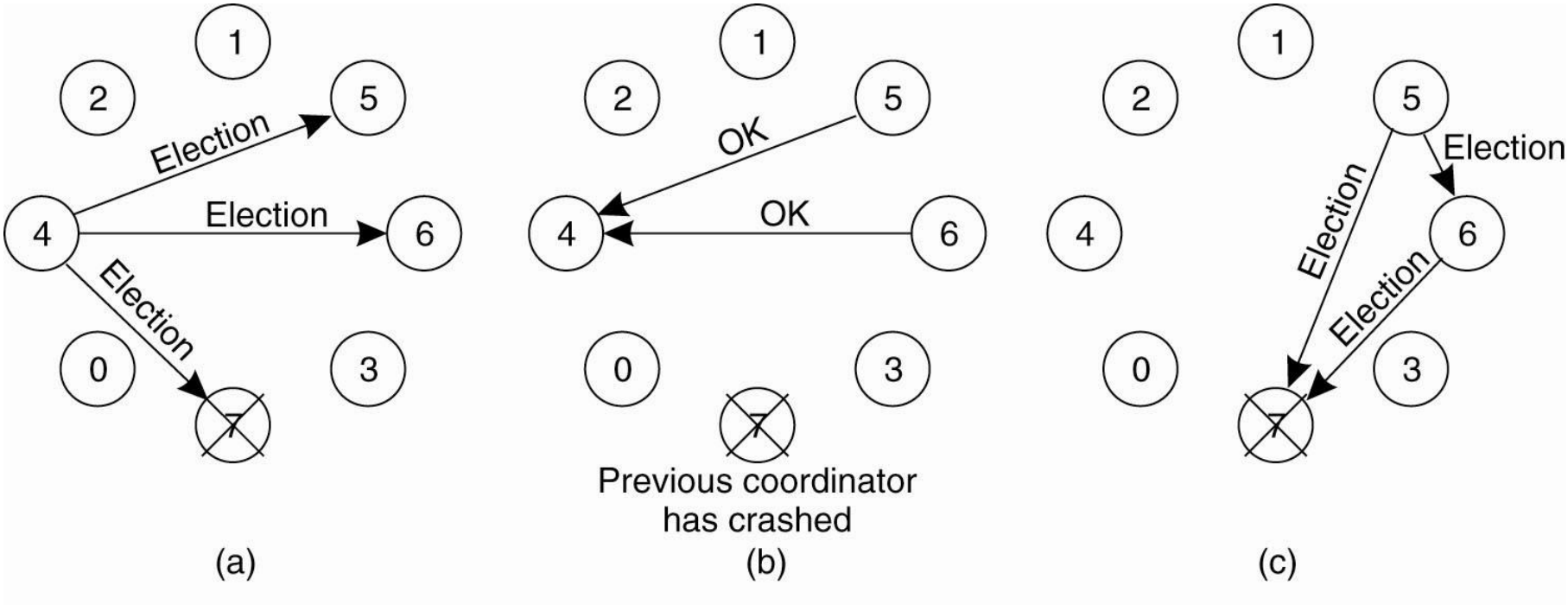


Figure 6-20. The bully election algorithm. (a) Process 4 holds an election. (b) Processes 5 and 6 respond, telling 4 to stop. (c) Now 5 and 6 each hold an election.

The Bully Algorithm (2)

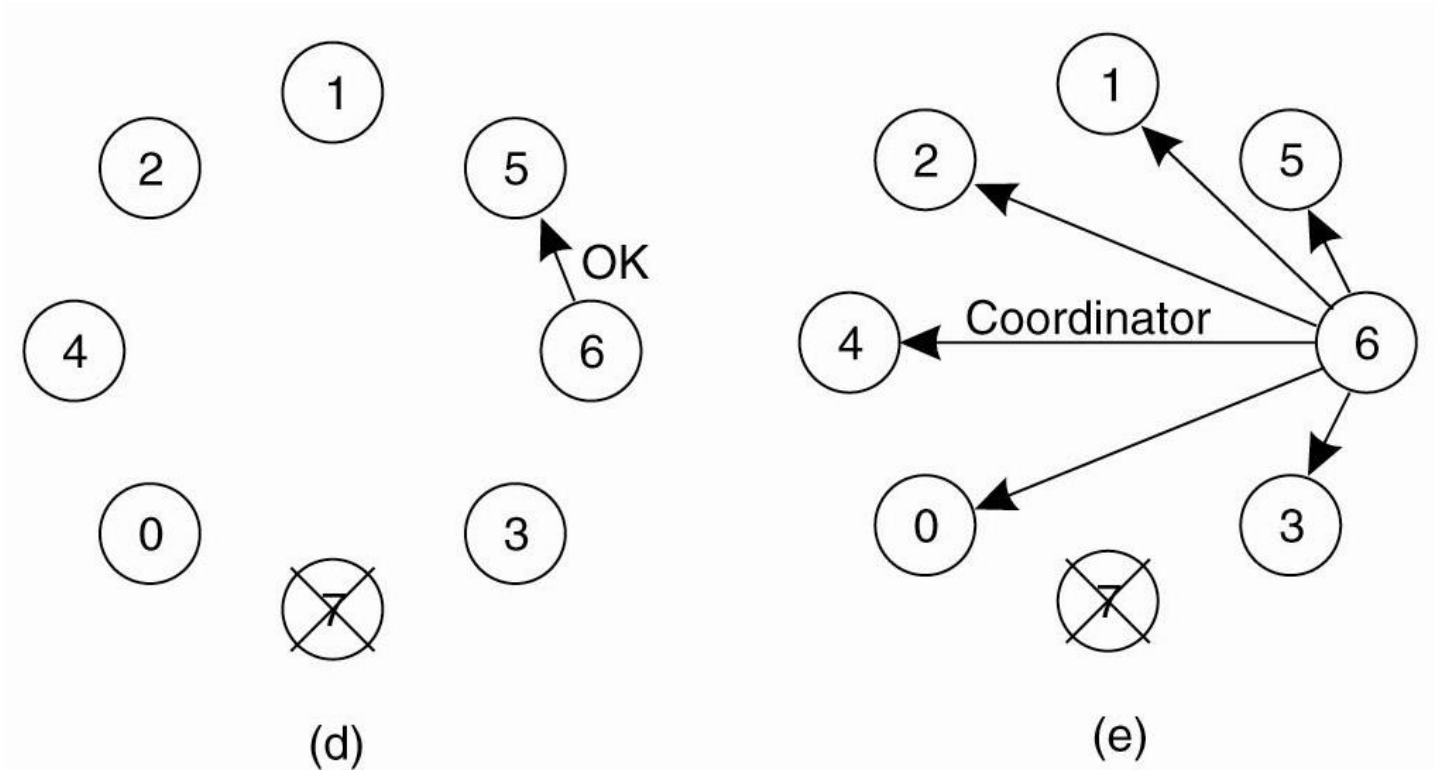
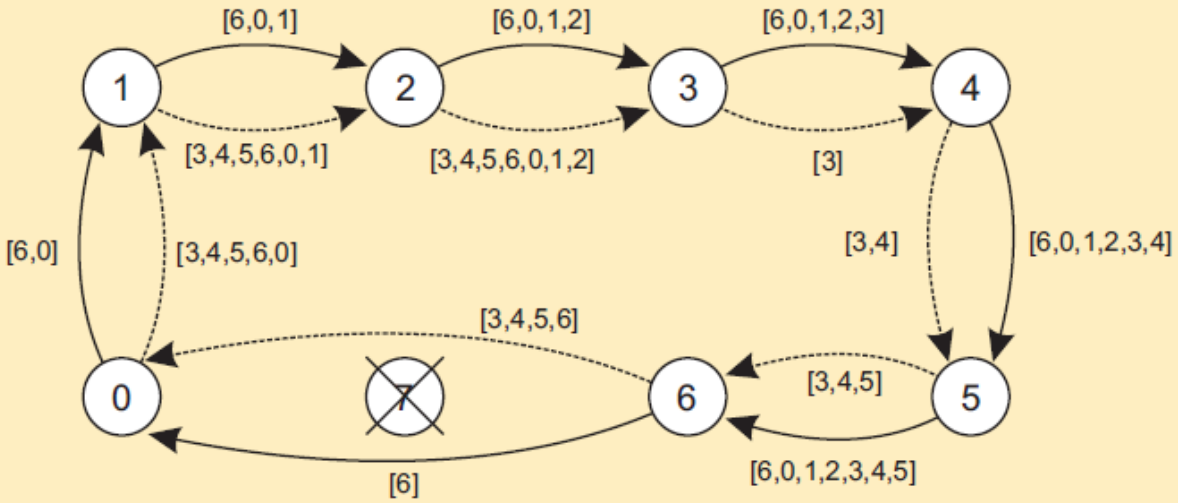


Figure 6-20. The bully election algorithm. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone.

A Ring Algorithm

Election algorithm using a ring



- The solid line shows the election messages initiated by P_6
- The dashed one the messages by P_3

Elections in Wireless Environments (1)

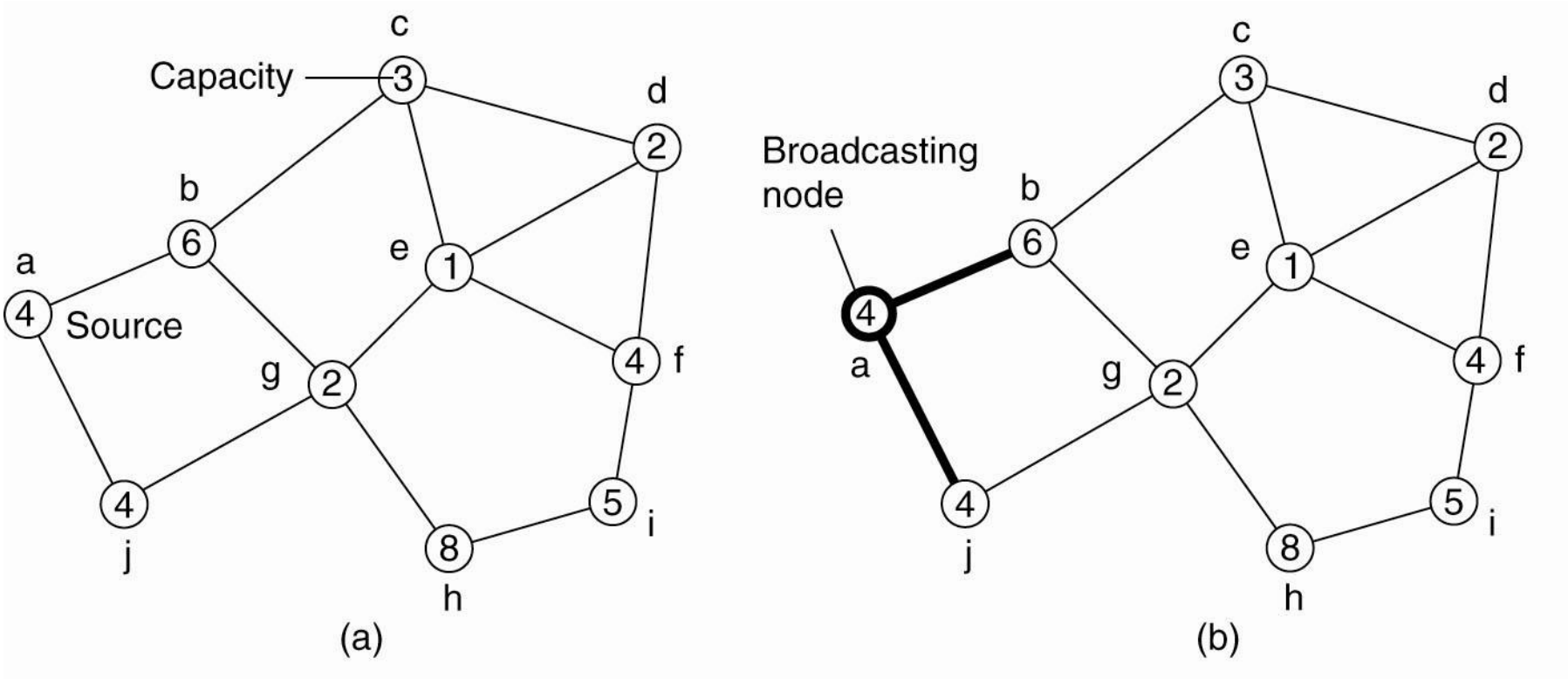


Figure 6-22. Election algorithm in a wireless network, with node a as the source. (a) Initial network. (b)–(e) The build-tree phase

Elections in Wireless Environments (2)

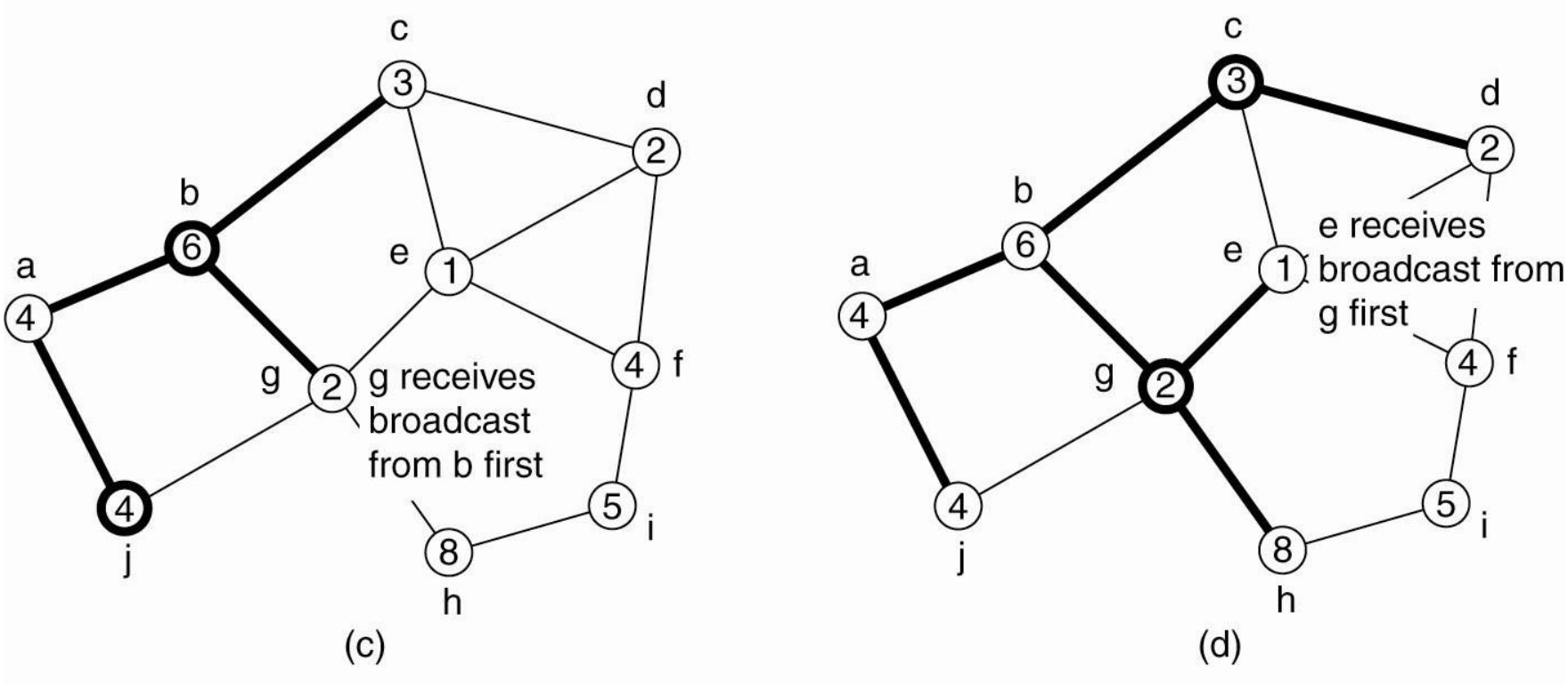


Figure 6-22. Election algorithm in a wireless network, with node a as the source. (a) Initial network. (b)–(e) The build-tree phase

Elections in Wireless Environments (3)

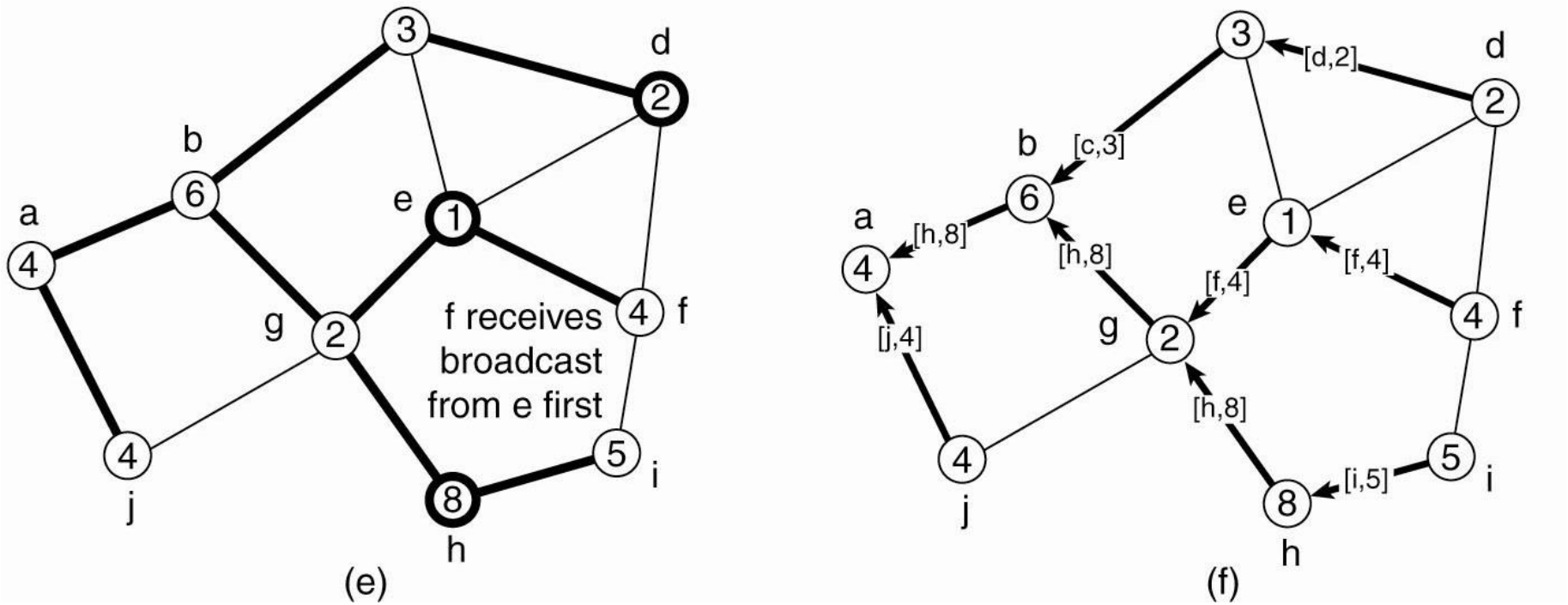


Figure 6-22. (e) The build-tree phase.
(f) Reporting of best node to source.

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)

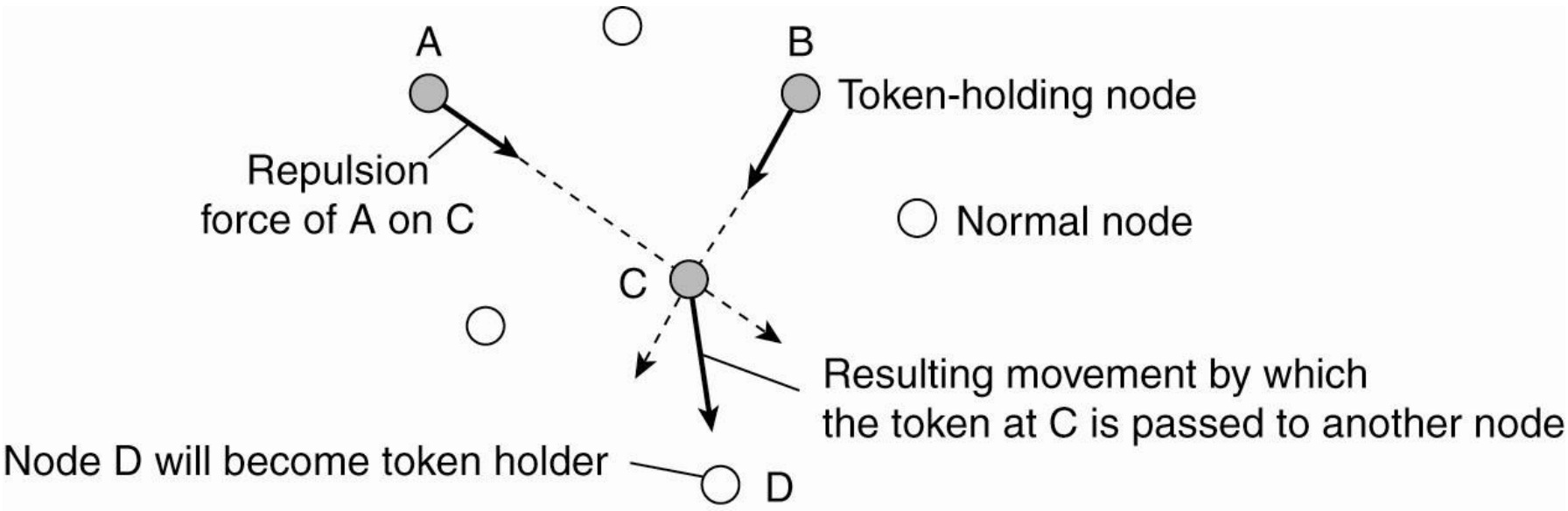


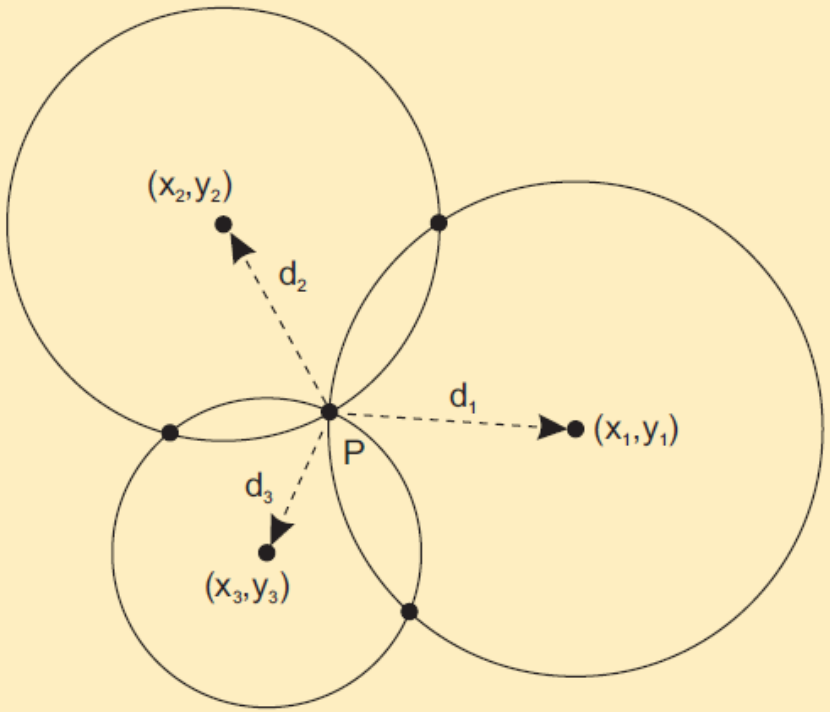
Figure 6-23. Moving tokens in a two-dimensional space using repulsion forces.

Global Positioning System

Observation

A node P needs $d+1$ landmarks to compute its own position in a d-dimensional space. Consider two-dimensional case.

Computing a position in 2D



Solution

P needs to solve three equations in two unknowns (x_P, y_P):

$$d_i = \sqrt{(x_i - x_P)^2 + (y_i - y_P)^2}$$

Global Positioning System (1)

Assuming that the clocks of the satellites are accurate and synchronized

- It takes a while before a signal reaches the receiver
- The receiver's clock is definitely out of sync with the satellite

Basics

- Δ_r: unknown deviation of the receiver's clock.
- x_r, y_r, z_r: unknown coordinates of the receiver.
- T_i: timestamp on a message from satellite i
- $\Delta_i = (T_{now} - T_i) + \Delta_r$: measured delay of the message sent by satellite i.
- Measured distance to satellite i: $c \times \Delta_i$ (c is speed of light)
- Real distance: $d_i = c\Delta_i - c\Delta_r = \sqrt{(x_i - x_r)^2 + (y_i - y_r)^2 + (z_i - z_r)^2}$

Observation

4 satellites $\Rightarrow$ 4 equations in 4 unknowns (with Δ_r as one of them)

WiFi-based location services

Basic idea

- Assume we have a database of known access points (APs) with coordinates
- Assume we can estimate distance to an AP
- Then: with 3 detected access points, we can compute a position.

War driving: locating access points

- Use a WiFi-enabled device along with a GPS receiver, and move through an area while recording observed access points.
- Compute the centroid: assume an access point AP has been detected at N different locations $\{\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_N\}$, with known GPS location.
- Compute location of AP as $\vec{x}_{AP} = \frac{\sum_{i=1}^{N} \vec{x}_i}{N}$.

Problems

- Limited accuracy of each GPS detection point $\vec{x}_i$
- An access point has a nonuniform transmission range
- Number of sampled detection points N may be too low.

Global Positioning Of Nodes (2)

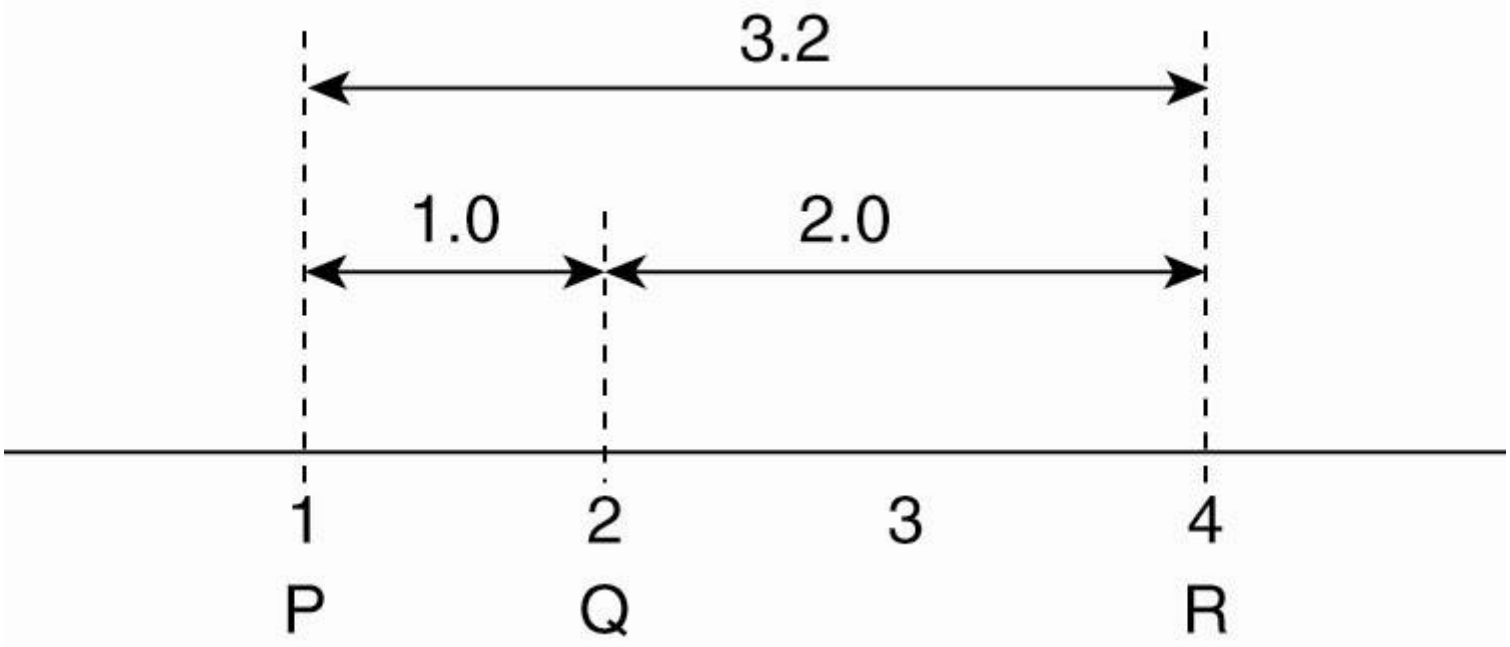


Figure 6-19. Inconsistent distance measurements in a one-dimensional space.