DISTRIBUTED SYSTEMS

Module-3

(Chapter 6)

COORDINATION

- The emphasis is on how processes can **synchronize** and **coordinate** their actions.

  Example: Access to a shared resource by multiple processes

- **Process synchronization:** One process waits for another to complete its operation.
- **Coordination:** The goal is to manage the interactions and dependencies between activities in a distributed system. **Coordination encapsulates synchronization.**
- Coordination in distributed systems is often much more difficult compared to that in uniprocessor or multiprocessor systems.
- The coordination problems and solutions are discussed.

6.1 CLOCK SYNCHRONIZATION

- In a centralized system, time is unambiguous. When a process wants to know the time, it simply makes a call to the operating system.
- If two processes A and B obtain time from OS in that order, then,

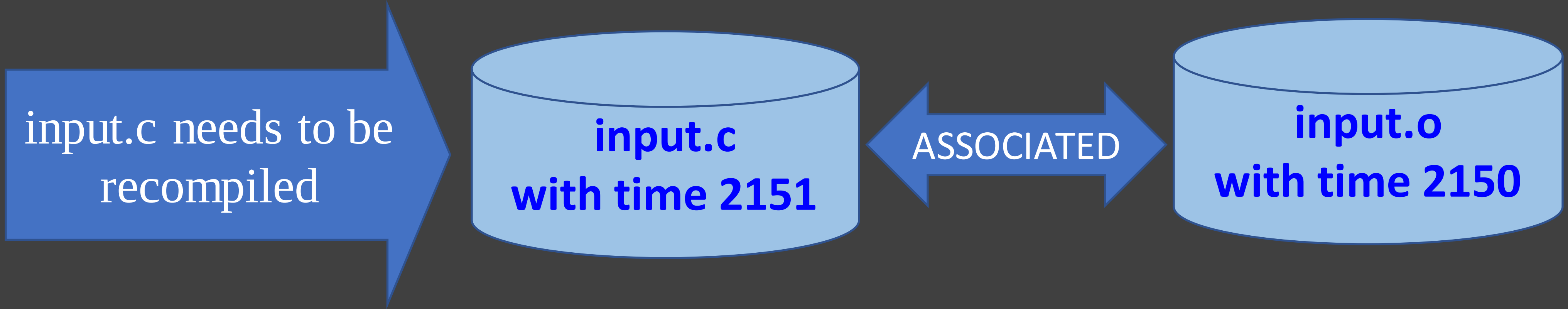
Process B’s time > = Process A’s time

- This is difficult to achieve in Distributed System (DS)

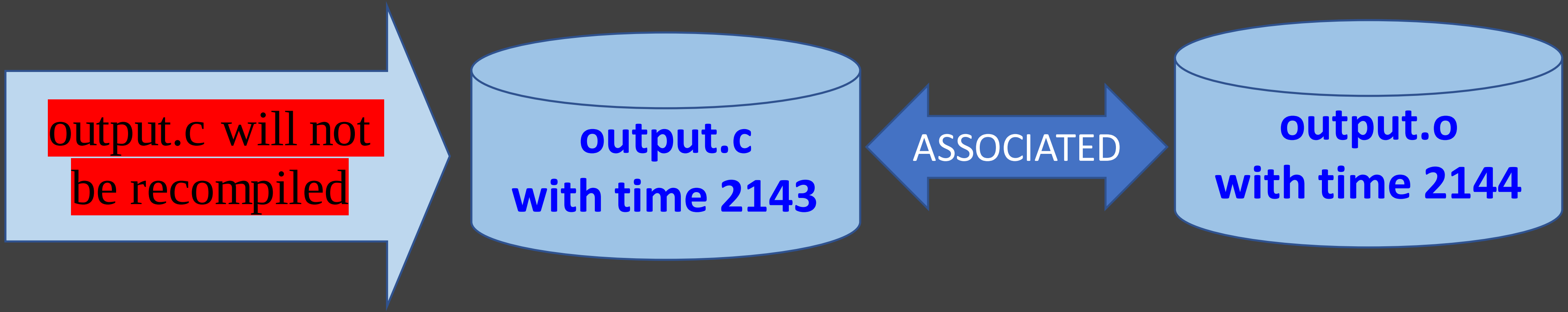
Example: If (Think) there is lack of global time in Unix *make* program.

In Unix large programs are split up into multiple source files, so that a change to one source file requires only one file to be recompiled, not all the files.

- **Working of *make*:** 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.



- **In DS where there is 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 Figure 6.1. Make will not call the compiler resulting into erroneous binary program.



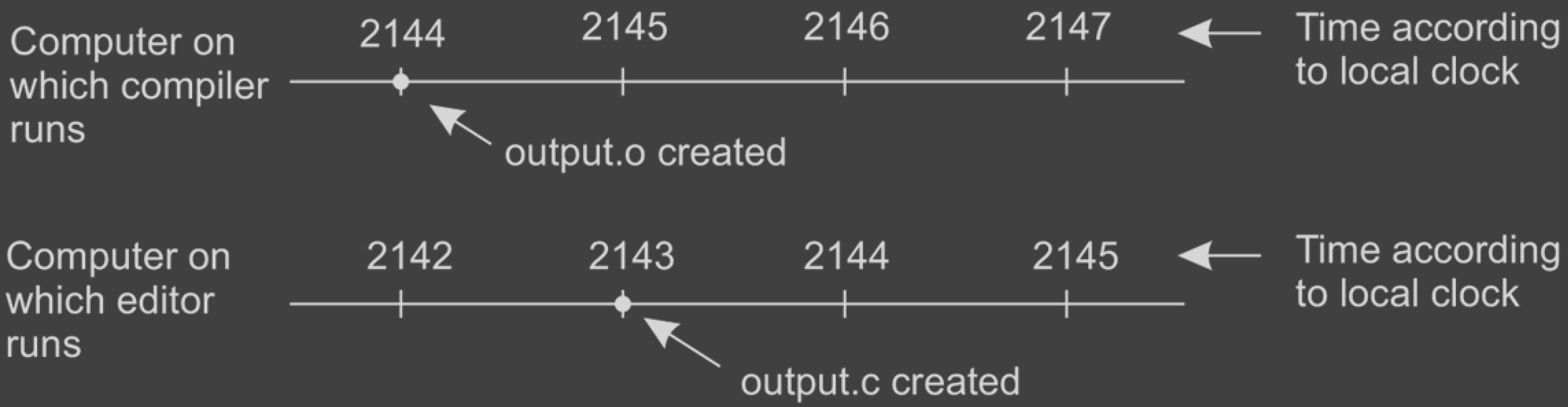

Computer on which compiler runs
2144
2145
2146
2147
Time according to local clock
output.o created
Computer on which editor runs
2142
2143
2144
2145
Time according to local clock
output.c created

Figure 6.1: When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

Physical clocks

- Computers have a circuit for keeping track of time. This is termed as timer, which is machined quartz crystal. Each crystal has two registers, a counter and a holding register.
- Each oscillation of the crystal decrements the counter by one. When the counter gets to zero, an interrupt is generated and the counter is reloaded from the holding register.
- In this way, it is possible to program a timer to generate an interrupt 60 times a second, or at any other desired frequency. Each interrupt is called one clock tick.
- When the system is booted, it usually asks the user to enter the date and time, which is then converted to the number of ticks after some known starting date and stored in memory.

- At every clock tick, the interrupt service procedure adds one to the time stored in memory. In this way, the (software) clock is kept up to date.
- With a single computer and a single clock, it does not matter much if this clock is off by a small amount. Since all processes on the machine use the same clock, they will still be internally consistent.
- With multiple CPUs, each with its own clock, the situation changes.
- Although the frequency at which a crystal oscillator runs is usually fairly stable, it is impossible to guarantee that the crystals in different computers all run at exactly the same frequency.

- When a system has n computers, all n crystals will run at slightly different rates, causing the (software) clocks gradually to get out of sync and give different values when read out. This difference in time values is called clock skew.
- As a consequence of this clock skew, programs can fail. (As in above *make* example)
- In some systems (e.g., real-time systems), the actual clock time is important. In such cases external physical clocks are needed.
- For reasons of efficiency and redundancy, multiple physical clocks are generally considered desirable, which yields two problems:

  (1) How do we synchronize them with real-world clocks, and

  (2) How do we synchronize the clocks with each other?
- The basis for keeping global time is called Universal Coordinated Time, but is abbreviated as UTC. UTC is the basis of all modern civil timekeeping and is a worldwide standard.

Clock synchronization algorithms

- If one machine has a UTC receiver, the goal becomes keeping all the other machines synchronized to it.

- If no machines have UTC receivers, each machine keeps track of its own time, and the goal is to keep all the machines together as well as possible.

- When the UTC time is t, denote by Cp(t) the value of the software clock on machine p. The goal of clock synchronization algorithms is to keep the deviation between the respective clocks of any two machines in a distributed system, within a specified bound, known as the **precision *π***:

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

- **Precision** refers to the deviation of clocks only between machines that are part of a distributed system.

- When considering an external reference point, like UTC, we speak of **accuracy**, aiming to keep it bound to a value ***α*****:**

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

- The whole idea of clock synchronization is that we keep clocks ***precise***, referred to as **internal synchronization** or ***accurate***, known as **external synchronization**.

- **Clock drift:** Because of variations in frequency and temperature, clocks on different machines will show different values for time. This is called clock drift.

- **Clock drift rate:** The difference per unit of time from a perfect reference clock.

- The specifications of a hardware clock include its maximum clock drift rate $\boldsymbol{\rho}$.

  If F(t) denotes the actual oscillator frequency of the hardware clock at time t and F its ideal (constant) frequency, then a hardware clock is living up to its specifications if,

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

- By using hardware interrupts we are directly coupling a software clock to the hardware clock, and thus also its clock drift rate. In particular, we have that,

$$C_p(t) = \frac{1}{F}\int_0^t F(t)dt, \text{ and thus: } \frac{dC_p(t)}{dt} = \frac{F(t)}{F}$$

which brings us to our ultimate goal, namely keeping the software clock drift rate also bounded to ***ρ*** :

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

Slow, perfect, and fast clocks are shown in Figure 6.4.

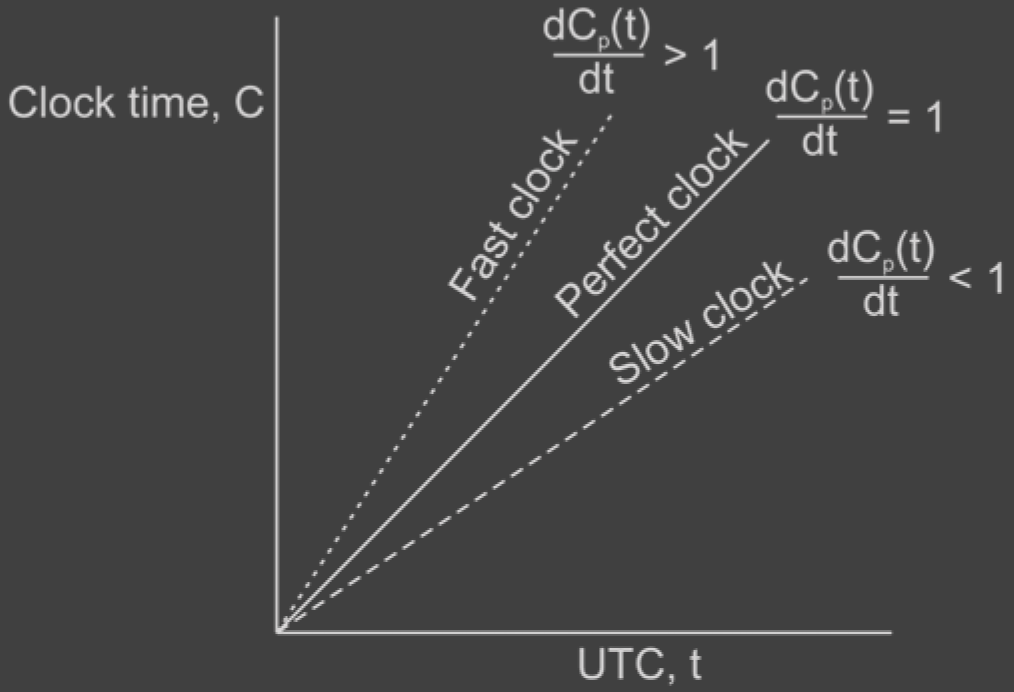


Figure 6.4: The relation between clock time and UTC when clocks tick at different rates.

Network Time Protocol

- Clients contact a time server, which provides the accurate current time as it can be equipped with a UTC receiver or an accurate clock.

- **The issue:** Message delays will have outdated the reported time. This issue can be solved by finding a good estimation for these delays.

- In the situation as depicted in Figure 6.5, 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.

- Let us assume that the propagation delays from A to B is roughly the same as B to A, meaning that

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

In that case, A can estimate its offset relative to B as,

$$\theta = T_3 + \frac{(T_2 - T_1) + (T_4 - T_3)}{2} - T_4 = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}$$

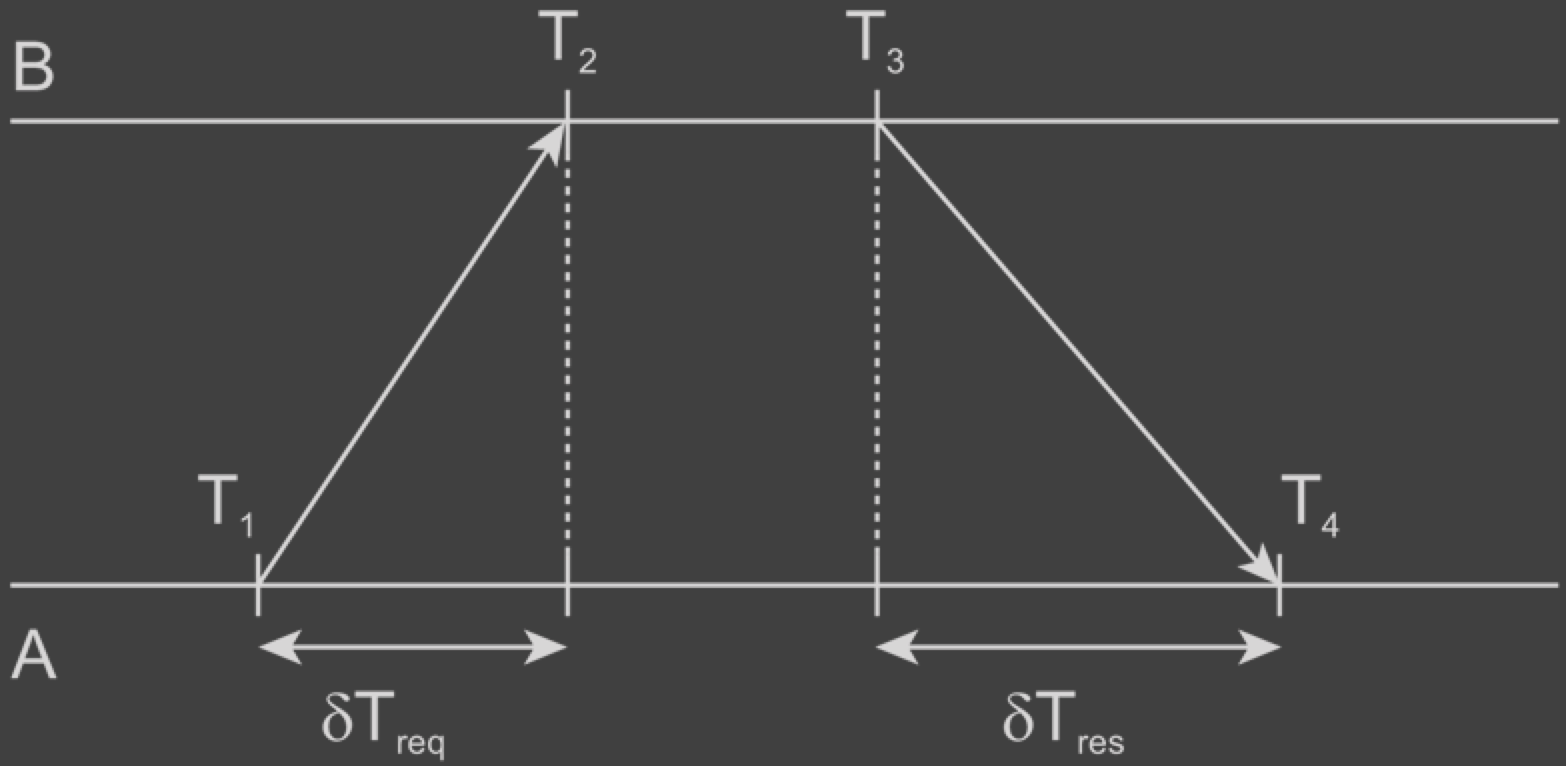


Figure 6.5: Getting the current time from a time server.

- In the case of the Network Time Protocol (NTP), this protocol is set up pairwise between servers.

- In other words, B will also probe A for its current time. The offset θ is computed as given above, along with the estimation δ for the delay:

$$\delta = \frac{(T_4 - T_1) - (T_3 - T_2)}{2}$$

- Eight pairs of (θ, δ) values are buffered, finally taking the minimal value found for δ as the best estimation for the delay between the two servers, and subsequently the associated value θ as the most reliable estimation of the offset.

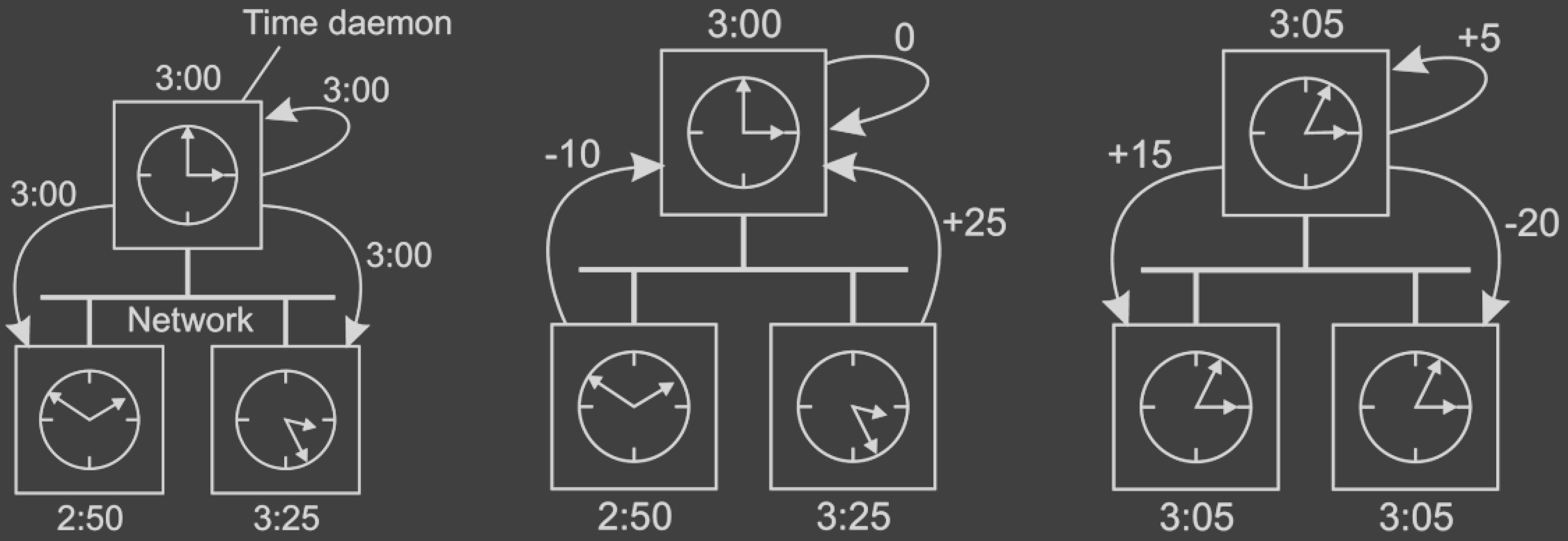
The Berkeley algorithm

- 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.

- This method is suitable for a system in which no machine has a UTC receiver. The time daemon's time must be set manually by the operator periodically.

- In Figure 6.6(a) at 3:00, the time daemon tells the other machines its time and asks for theirs. In Figure 6.6(b) they respond with how far ahead or behind the time daemon they are. Armed with these numbers, the time daemon computes the average and tells each machine how to adjust its clock [see Figure 6.6(c)].



(a) (b) (c)

Figure 6.6: (a) The time daemon asks all the other machines for their clock values. (b) The machines answer. (c) The time daemon tells everyone how to adjust their clock.

Clock synchronization in wireless networks

- In wireless networks, notably sensor networks, nodes are resource constrained and multihop routing is expensive. Algorithms needs to be optimized for energy consumption. Hence there is need of different clock synchronization algorithms.

- **Reference Broadcast Synchronization (RBS) :** Clock synchronization protocol.

- **First:** The protocol does not assume that there is a single node with an accurate account of the actual time available. Instead of aiming to provide all nodes UTC time, it aims at internally synchronizing the clocks, just as the Berkeley algorithm does.

- **Second:** Only the receivers synchronize, keeping the sender out of the loop.

- In RBS, a sender broadcasts a reference message that will allow its receivers to adjust their clocks.

- In a sensor network the time to propagate a signal to other nodes is roughly constant, provided no multihop routing is assumed.

- Propagation time in this case is measured from the moment that a message leaves the network interface of the sender.

- As a consequence, two important sources for variation in message transfer no longer play a role in estimating delays: the time spent to construct a message, and the time spent to access the network.

- This principle is shown in Figure 6.7.

- Furthermore, as wireless networks are based on a contention protocol, there is generally no saying how long it will take before a message can actually be transmitted.

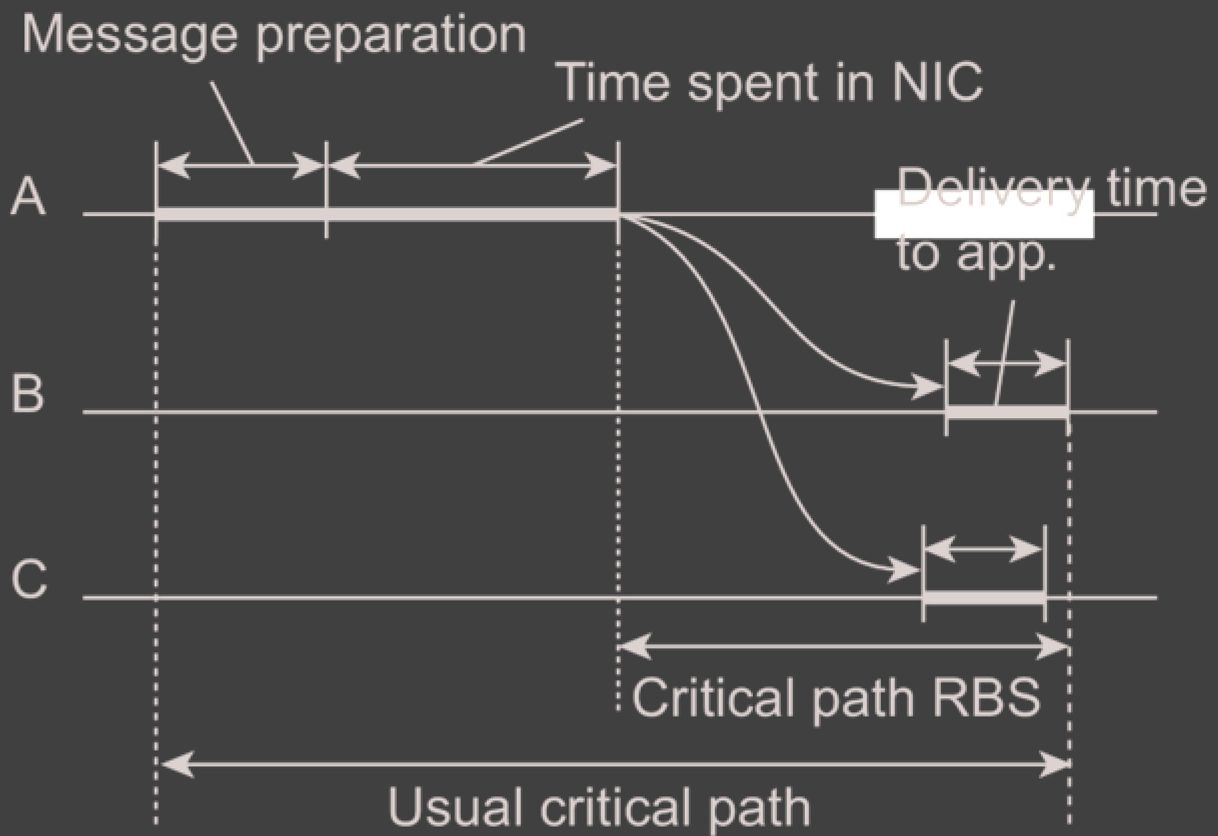


Figure 6.7: The usual critical path and the one used in RBS in determining network delays.

- The idea underlying RBS is simple: When a node broadcasts a reference message m, each node p simply records the time $T_{p,m}$ that it received m. Note that $T_{p,m}$ is read from p's local clock.

- Ignoring clock skew, two nodes p and q can exchange each other's delivery times in order to estimate their mutual, relative offset:

$$Offset[p,q] = \frac{\sum_{k=1}^{M}(T_{p,k} - T_{q,k})}{M}$$

where M is the total number of reference messages sent.

- This information is important: node p will know the value of q's clock relative to its own value. Moreover, if it simply stores these offsets, there is no need to adjust its own clock, which saves energy.

6.2 LOGICAL CLOCKS

- To achieve synchronization in a DS it may be sufficient that every node in a distributed system agrees on *a* current time and they keep track of each other's events. This is achieved by using logical clocks.

- **Lamport's logical clocks**

- To synchronize logical clocks, Lamport defined a relation called happens before. The expression $a \rightarrow b$ is read "event *a* happens before event *b*" and means that all processes agree that first event *a* occurs, then afterward, event *b* occurs.

- The happens-before relation can be observed directly in two situations:
  1. If *a* and *b* are events in the same process, and *a* occurs before *b*, then *a* → *b* is true.

  2. 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* is also true. A message cannot be received before it is sent, or even at the same time it is sent, since it takes a finite, nonzero amount of time to arrive.

- Happens-before is a transitive relation, so if *a* → *b* and *b* → *c*, then *a* → *c*.

- If two events, x and y, happen in different processes that do not exchange messages (not even indirectly via third parties), then *x* → *y* is not true, but neither is *y* → *x*. These events are said to be **concurrent**, which simply means that nothing can be said (or need be said) about when the events happened or which event happened first.

- What we need is a way of measuring a notion of time such that for every event, *a*, we can assign it a time value *C(a)* on which all processes agree. These time values must have the property that if $a \rightarrow b$, then $C(a) < C(b)$.

- If a and b are two events within the same process and a occurs before b, then $C(a) < C(b)$.

- Similarly, if a is the sending of a message by one process and b is the reception of that message by another process, then C(a) and C(b) must be assigned in such a way that everyone agrees on the values of C(a) and C(b) with $C(a) < C(b)$.

- In addition, the clock time, C, must always go forward (increasing), never backward (decreasing). Corrections to time can be made by adding a positive value, never by subtracting one.

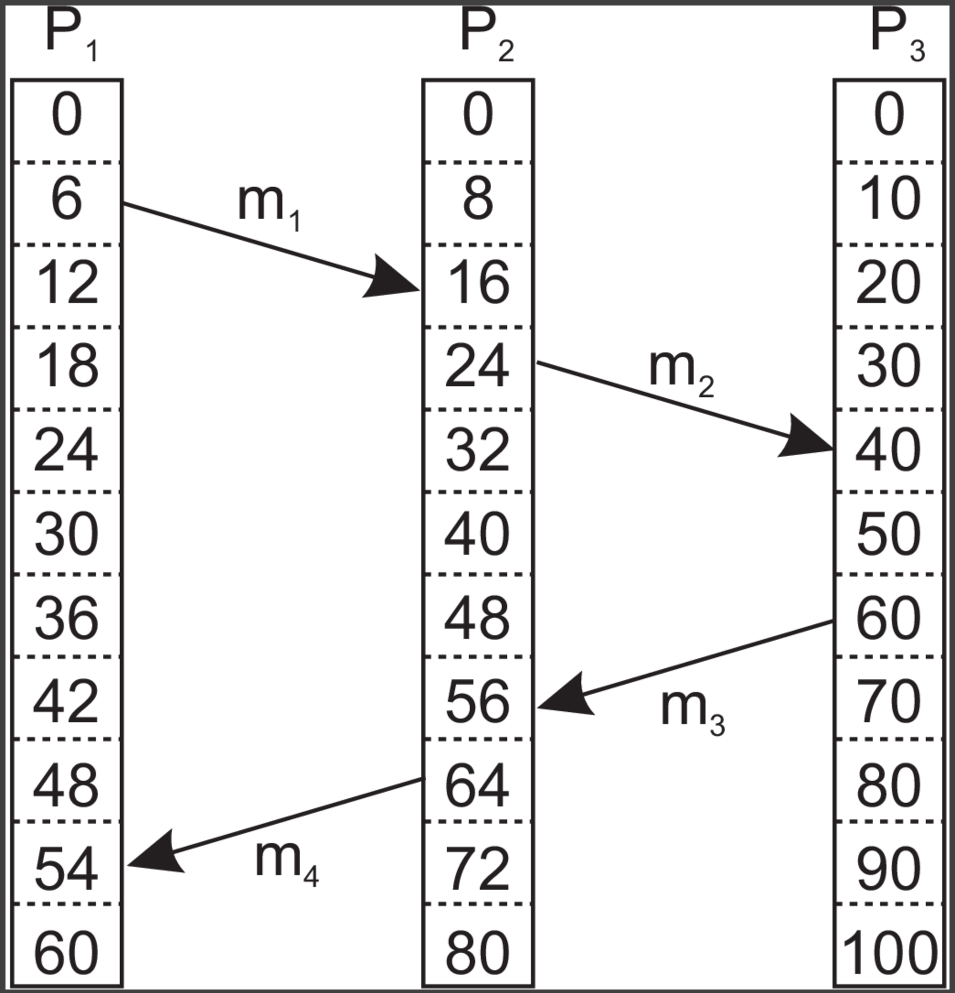
Lamport’s algorithm for assigning times to events.

- The three processes, as shown in Figure 6.8, run on different machines, each with its own clock.

- The clock is implemented as a software counter: the counter is incremented by a specific value every T time units. However, the value by which a clock is incremented differs per process.

- The clock in process P1 is incremented by 6 units, 8 units in process P2, and 10 units in process P3, respectively.

- 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.

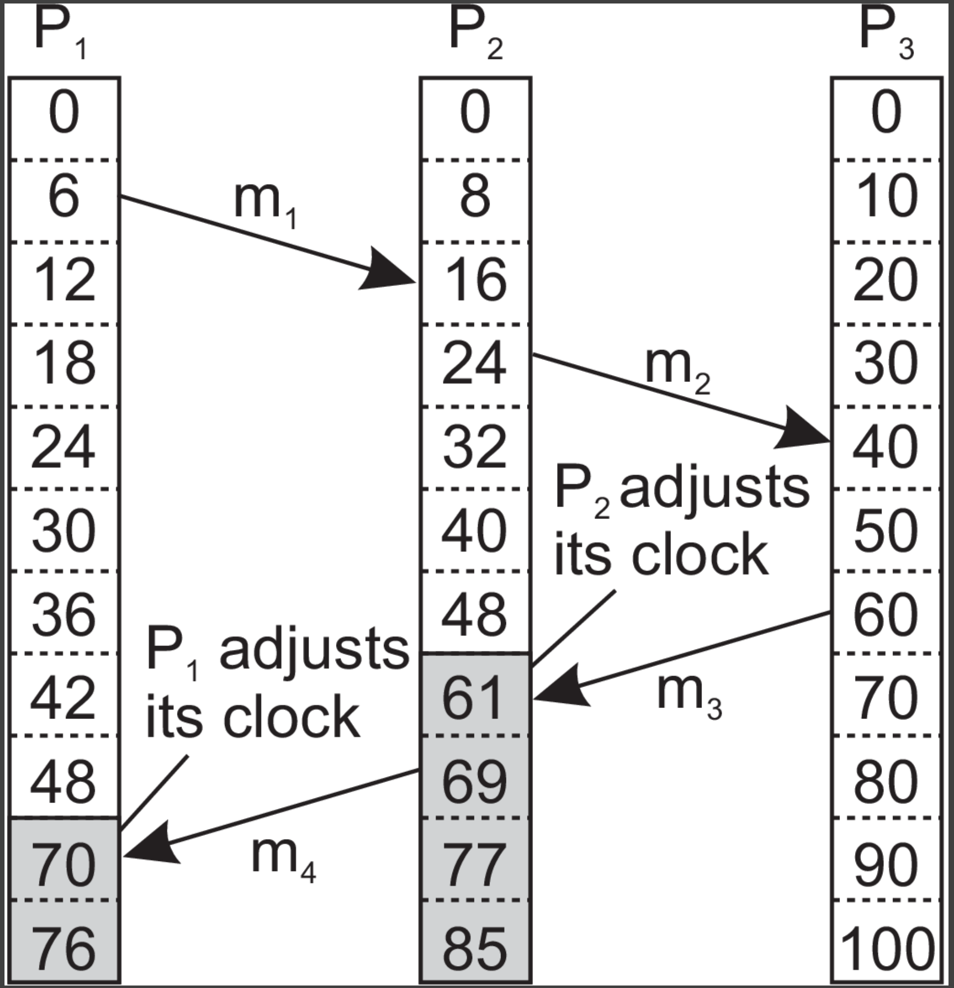


(a)

Figure 6.8: (a) Three processes, each with its own (logical) clock. The clocks run at different rates.

- Now consider message m3. It leaves process P3 at 60 and arrives at P2 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 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 Figure 6.8, we see that m3 now arrives at 61. Similarly, m4 arrives at 70.



(b)

Figure 6.8: (b) Lamport’s algorithm corrects their values.

- It is important to distinguish three different layers of software, the network, a middleware layer, and an application layer, as shown in Figure 6.9. What follows is typically part of the middleware layer.

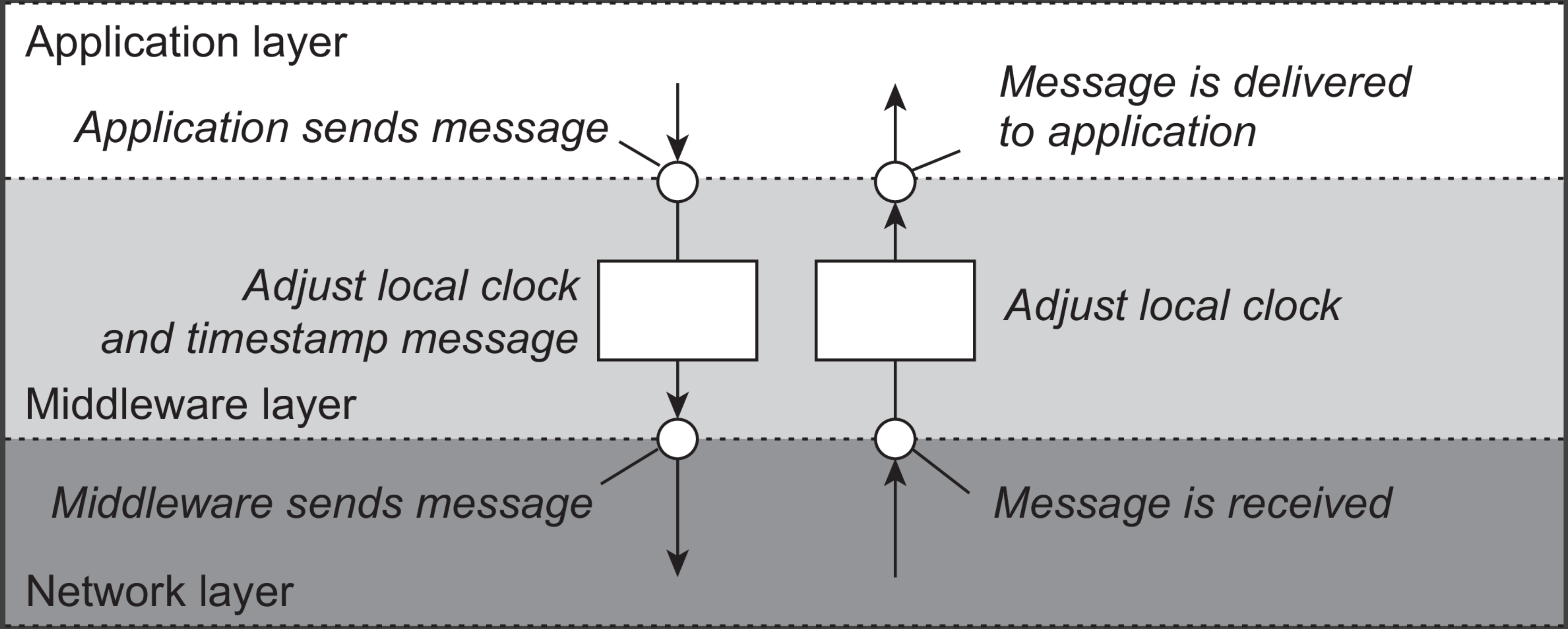


Figure 6.9: The positioning of Lamport's logical clocks in distributed systems.

- To implement Lamport’s logical clocks, each process Pi maintains a local counter Ci. These counters are updated according to the following steps:

1. Before executing an event (i.e., sending a message over the network, delivering a message to an application, or some other internal event), Pi increments Ci: Ci 🡐 Ci + 1.

2. When process Pi sends a message m to process Pj, it sets m’s timestamp ts(m) equal to Ci after having executed the previous step.

3. Upon the receipt of a message m, process Pj adjusts its own local counter as Cj 🡐 max{ Cj, ts(m) } after which it then executes the first step and delivers the message to the application.

Example of Lamport's logical clocks: Total-ordered multicasting

- Consider the situation in which a database has been replicated across several sites. For example, to improve query performance, a bank may place copies of an account database in two different cities, say New York and San Francisco. A query is always forwarded to the nearest copy. The database update needs care.

Update 1: Assume a customer in San Francisco wants to add $100 to his account, which currently contains $1,000.

Update 2: At the same time, a bank employee in New York initiates an update by which the customer's account is to be increased with 1 percent interest.

- Both updates should be carried out at both copies of the database. However, due to communication delays in the underlying network, the updates may arrive in the order as shown in Figure 6.10.

San Francisco – To add $100 New York – To add 1% interest

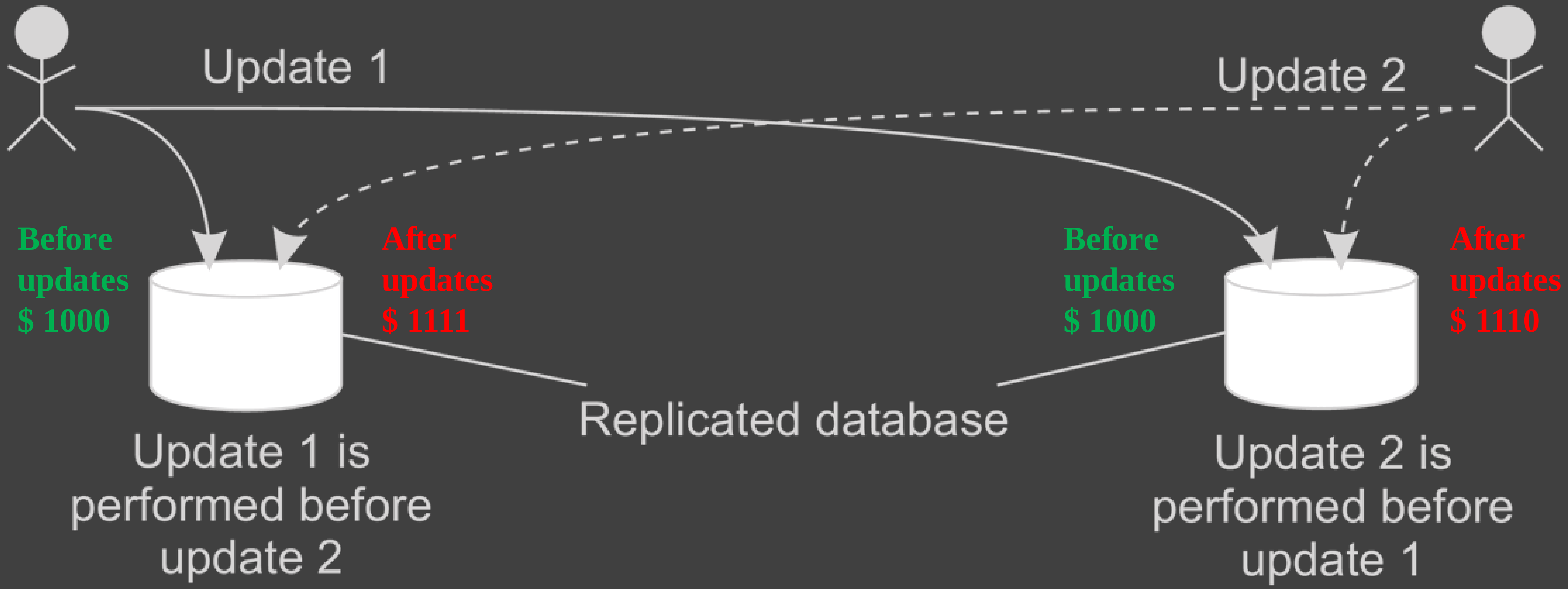


Figure 6.10: Updating a replicated database and leaving it in an inconsistent state.

- The customer's update operation is performed in San Francisco before the interest update. In contrast, the copy of the account in the New York replica is first updated with the 1 percent interest, and after that with the $100 deposit. Consequently, the San Francisco database will record a total amount of $1,111, whereas the New York database records $1,110.

- The important issue is that both copies should be exactly the same. In general, situations such as these require a total-ordered multicast.

- **Total-ordered multicast: A** multicast operation by which all messages are delivered in the same order to each receiver.

- Lamport's logical clocks can be used to implement total-ordered multicasts in a completely distributed fashion.

- Consider a group of processes multicasting messages to each other. Each message is always timestamped with the current (logical) time of its sender.

- When a message is multicast, it is conceptually also sent to the sender. In addition, we assume that messages from the same sender are received in the order they were sent, and that no messages are lost.

- When a process receives a message, it is put into a local queue, ordered according to its timestamp.

- The receiver multicasts an acknowledgment to the other processes. Note that if we follow Lamport’s algorithm for adjusting local clocks, the timestamp of the received message is lower than the timestamp of the acknowledgment.

- The interesting aspect of this approach is that all processes will eventually have the same copy of the local queue (provided no messages are removed).

- A process can deliver a queued message to the application it is running only when that message is at the head of the queue and has been acknowledged by each other process.

- At that point, the message is removed from the queue and handed over to the application; the associated acknowledgments can simply be removed.

- Because each process has the same copy of the queue, all messages are delivered in the same order everywhere. In other words, we have established total-ordered multicasting.

- Lamport clocks can be used to achieve mutual exclusion.

VECTOR CLOCKS

- With Lamport clocks, nothing can be said about the relationship between two events a and b by merely comparing their time values C(a) and C(b), respectively. In other words, if C(a) < C(b), then this does not necessarily imply that a indeed happened before b. Something more is needed for that.

- Consider the messages as sent by the three processes shown in Figure 6.12. Denote by Tsnd(mi) the logical time at which message mi was sent, and likewise, by Trcv(mi) the time of its receipt.

- By construction, we know that for each message Tsnd(mi) < Trcv(mi).

- But what can we conclude in general from Trcv(mi) < Tsnd(mj) for different messages mi and mj?

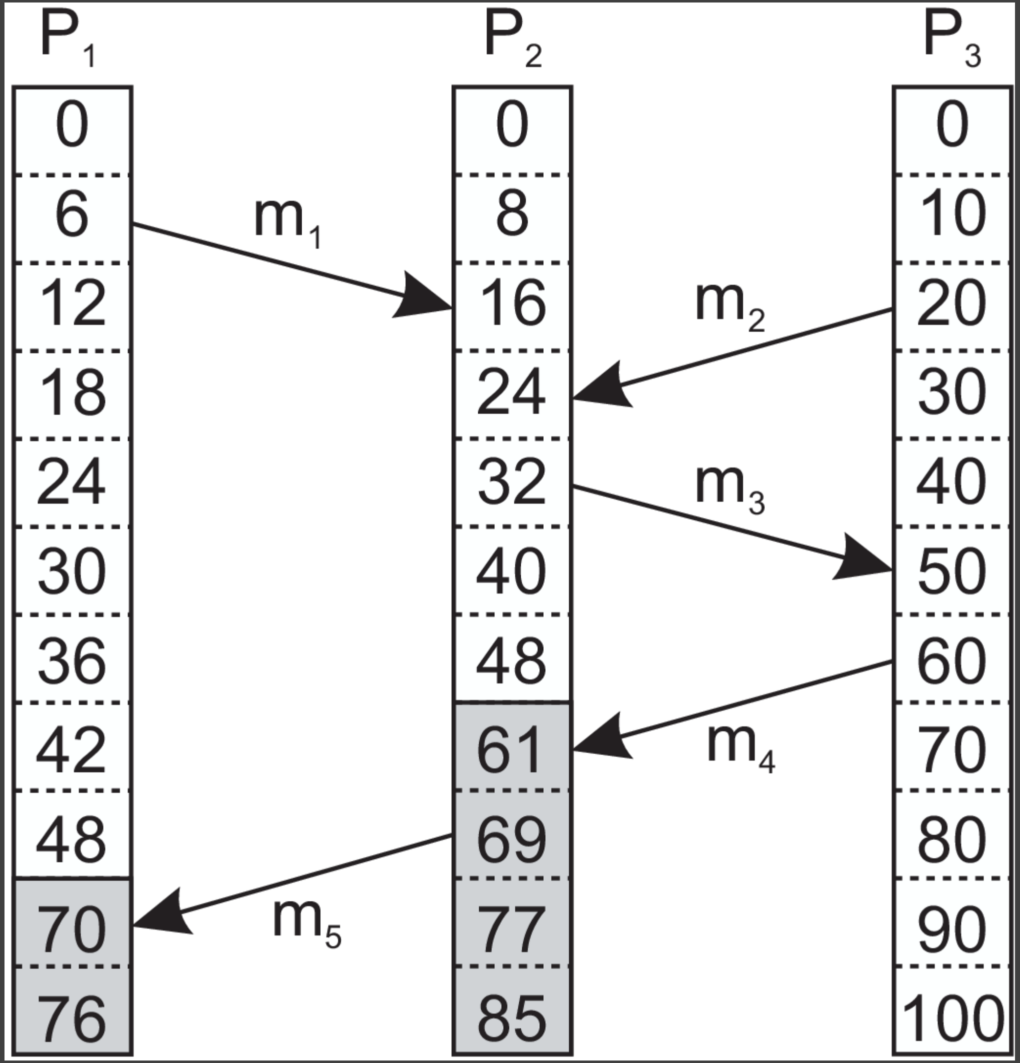


Figure 6.12: Concurrent message transmission using logical clocks.

- In the case for which $m_i = m_1$ and $m_j = m_3$, we know that these values correspond to events that took place at process P_2, meaning that m_3 was indeed sent after the receipt of message m_1.

- This may indicate that the sending of message m_3 depended on what was received through message m_1 . At the same time, we also know that $T_{rcv}(m_1) < T_{snd}(m_2)$.

- However, as far as we can tell from Figure 6.12, the sending of m_2 has nothing to do with the receipt of m_1.

- The problem is that Lamport clocks do not capture causality. In practice, causality is captured by means of vector clocks.

- Tracking causality is simple if we assign each event a unique name such as the combination of a process ID and a locally incrementing counter: pk is the kth event that happened at process P. The problem then boils down to keeping track of causal histories.
- For example, if two local events happened at process P, then the causal history H(p2) of event p2 is { p1, p2 }.
- Now assume that process P sends a message to process Q (which is an event at P and thus recorded as pk from some k), and that at the time of arrival (and event for Q), the most recent causal history of Q was { q1 }.
- To track causality, P also sends its most recent causal history (assume it was { p1, p2 }, extended with p3 expressing the sending of the message).
- Upon arrival, Q records the event (q2), and merges the two causal histories into a new one: { p1, p2, p3, q1, q2 }.

- Checking whether an event p causally precedes an event q can be done by checking whether $H(p) \subset H(q)$ (i.e., it should be a proper subset). In fact, with our notation, it even suffices to check whether $p \in H(q)$, assuming that q is always the last local event in $H(q)$.

- Problem with causal histories : Their representation is not very efficient. However, there is no need to keep track of all successive events from the same process: the last one will do.

- If we subsequently assign an index to each process, we can represent a causal history as a vector, in which the jth entry represents the number of events that happened at process Pj.

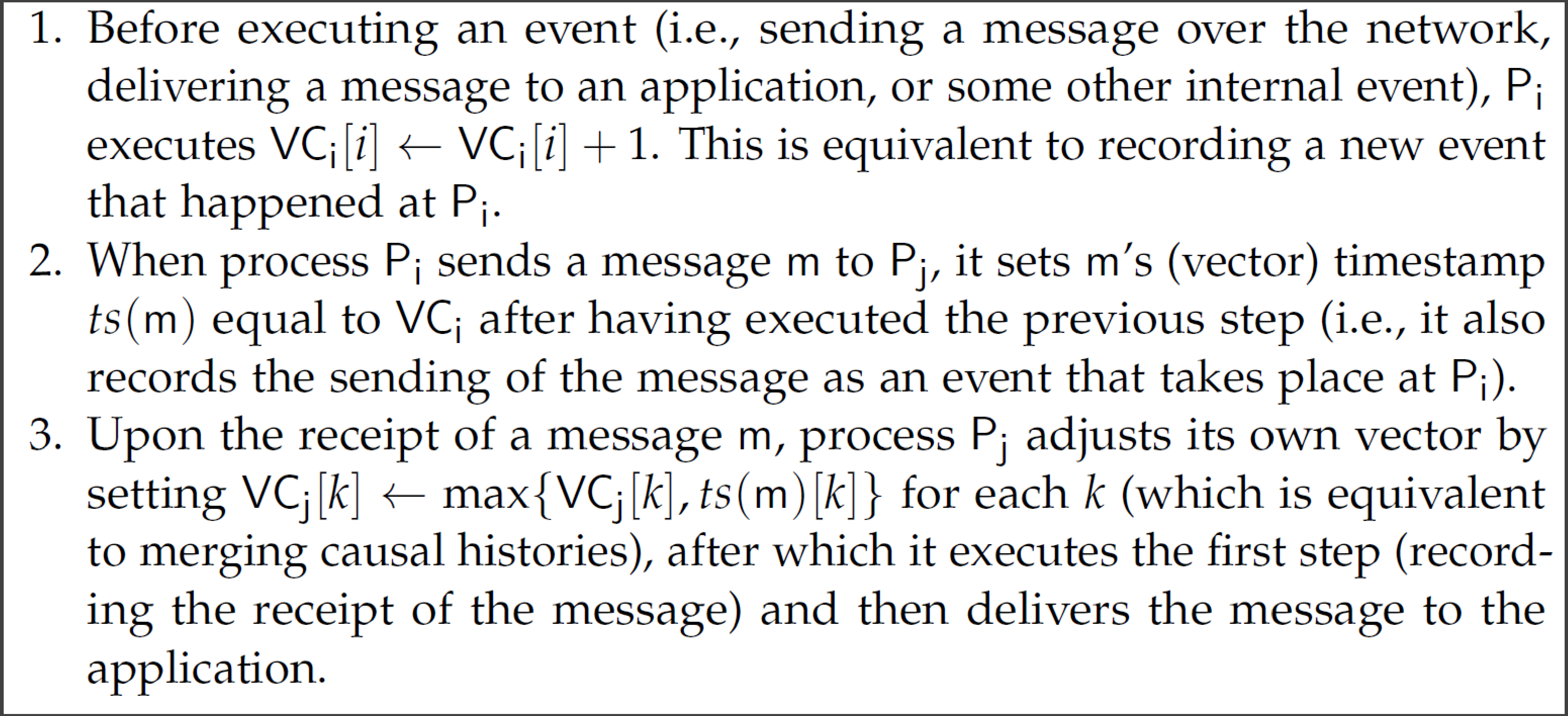
- Causality can then be captured by means of vector clocks (VC), which are constructed by letting each process P_i maintain a vector VC_i with the following two properties:

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.N this's , V

- The first property is maintained by incrementing $VC_i[i]$ at the occurrence of each new event that happens at process P_i .

- The second property is maintained by piggybacking vectors along with messages that are sent.

- In particular, the following steps are performed:



1. Before executing an event (i.e., sending a message over the network, delivering a message to an application, or some other internal event), P_i executes $VC_i[i] \leftarrow VC_i[i] + 1$. This is equivalent to recording a new event that happened at P_i.
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 (i.e., it also records the sending of the message as an event that takes place at P_i).
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 (which is equivalent to merging causal histories), after which it executes the first step (recording the receipt of the message) and then delivers the message to the application.

- Note that if an event a has timestamp ts(a), then ts(a)[i] - 1 denotes the number of events processed at Pi that causally precede a.

- As a consequence, when Pj receives a message m from Pi with timestamp ts(m), it knows about the number of events that have occurred at Pi that causally preceded the sending of m.

- More important, however, is that Pj is also told how many events at other processes have taken place, known to Pi, before Pi sent message m.

- In other words, timestamp ts(m) tells the receiver how many events in other processes have preceded the sending of m, and on which m may causally depend.

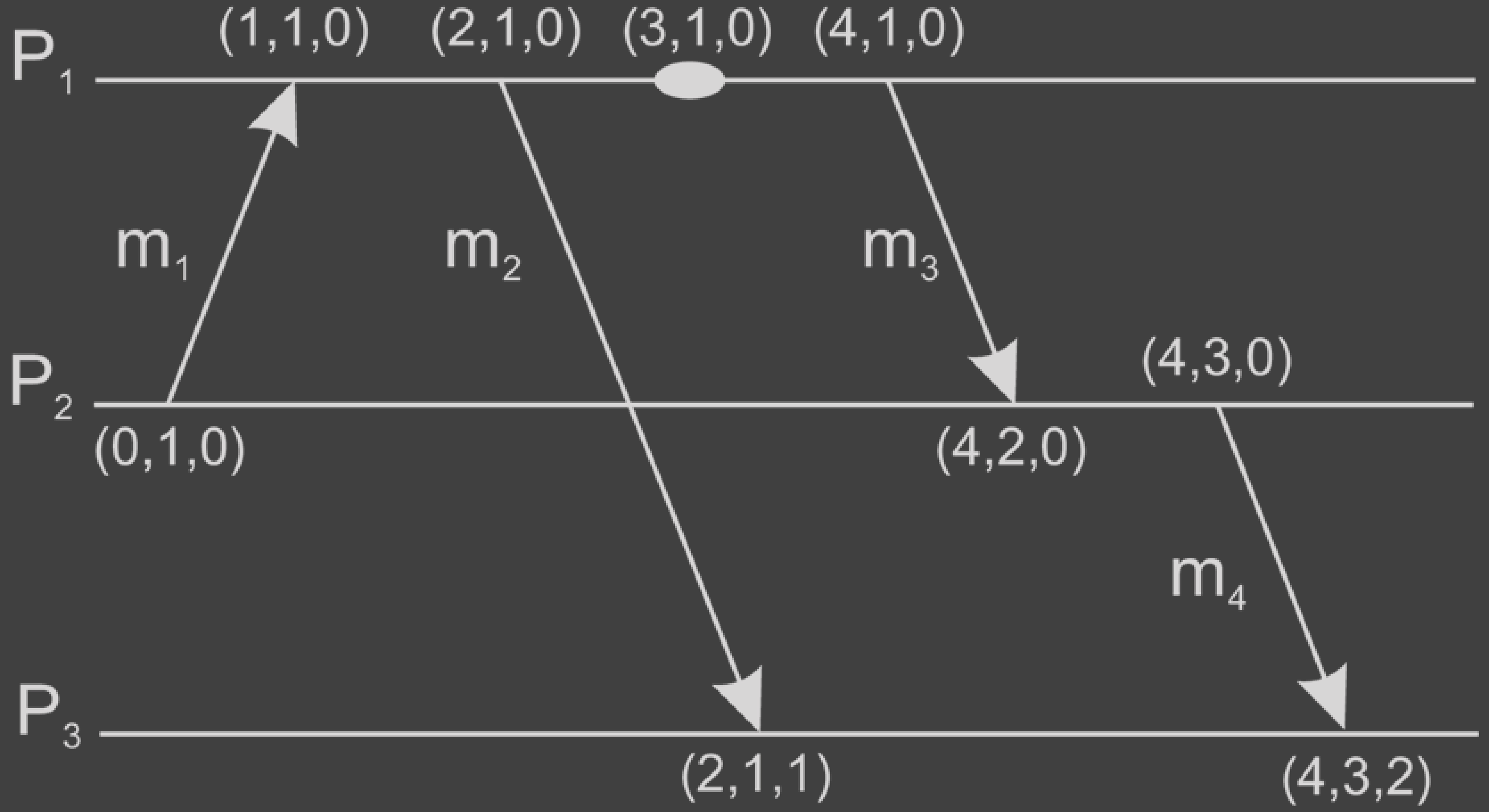
- To see what this means, consider Figure 6.13 which shows three processes.

- In Figure 6.13(a), P2 sends a message m1 at logical time VC2 = (0, 1, 0) to process P1. Message m1 thus receives timestamp ts(m1) = (0, 1, 0).

- Upon its receipt, P1 adjusts its logical time to VC1 ← (1, 1, 0) and delivers it.

- Message m2 is sent by P1 to P3 with timestamp ts(m2) = (2, 1, 0).

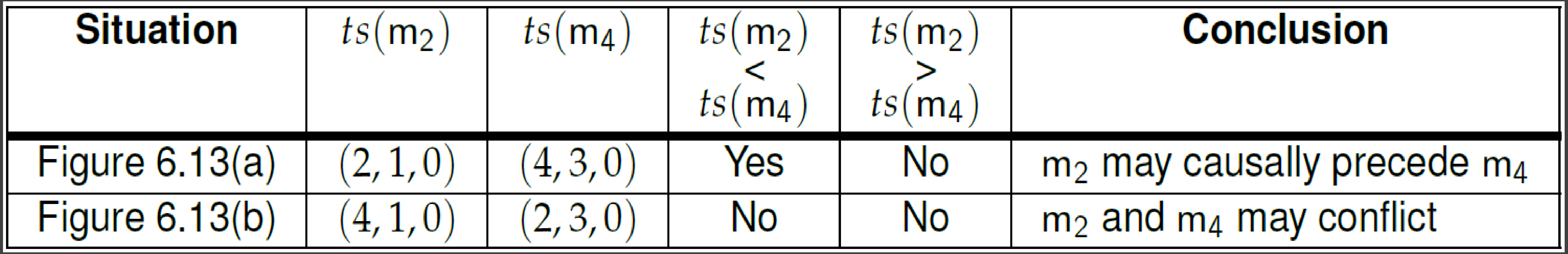
- Before P1 sends another message, m3, an event happens at P1, eventually leading to timestamping m3 with value (4, 1, 0). After receiving m3, process P2 sends message m4 to P3, with timestamp ts(m4) = (4, 3, 0).



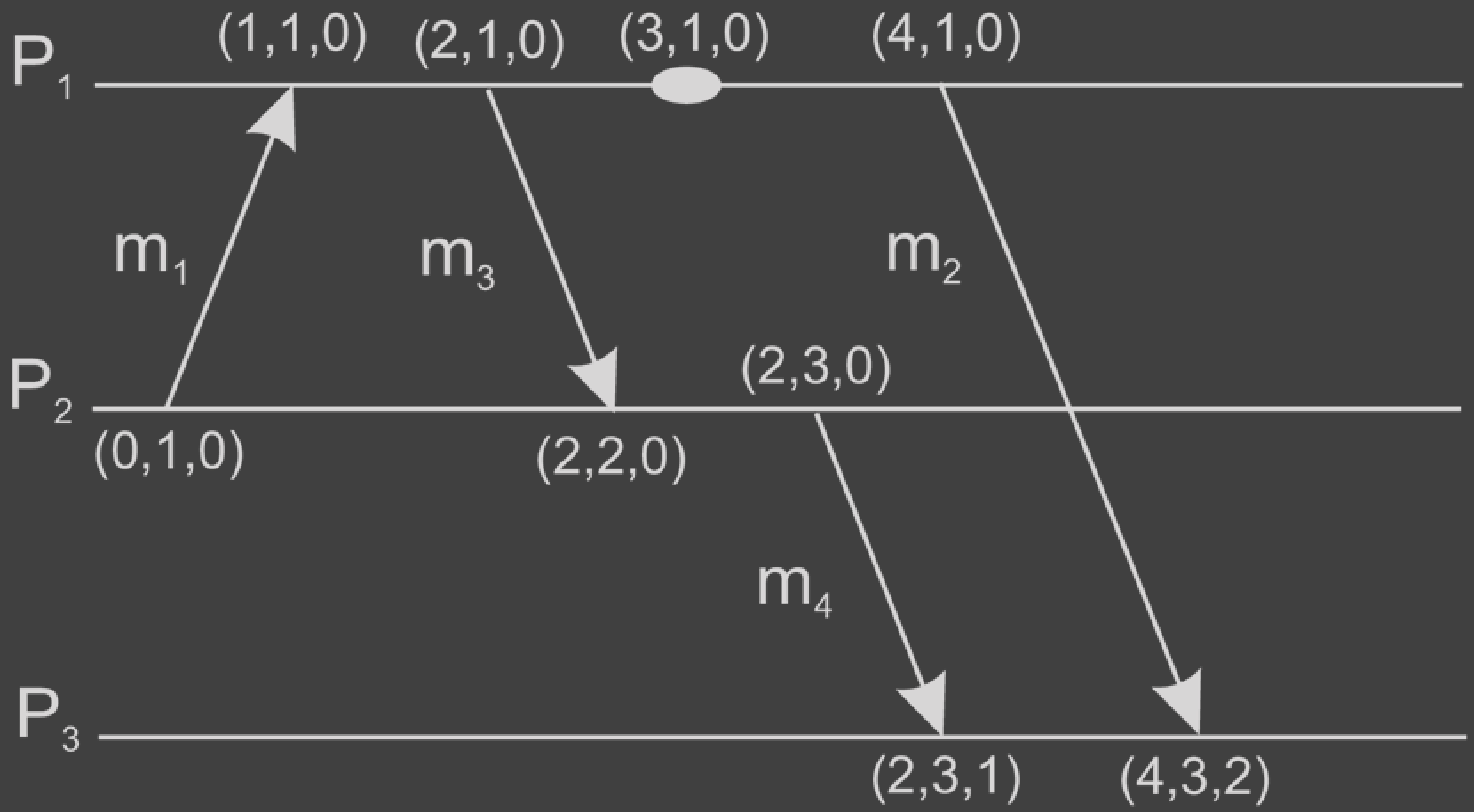
(a)

Figure 6.13: Capturing potential causality when exchanging messages.

- Now consider the situation shown in Figure 6.13(b).
- Here, we have delayed sending message m2 until after message m3 has been sent, and after the event had taken place.
- It is not difficult to see that ts(m2) = (4, 1, 0), while ts(m4) = (2, 3, 0). Compared to Figure 6.13(a), we have the following situation:



Situation	$ts(m_2)$	$ts(m_4)$	$ts(m_2) < ts(m_4)$	$ts(m_2) > ts(m_4)$	**Conclusion**
Figure 6.13(a)	$(2,1,0)$	$(4,3,0)$	Yes	No	m_2 may causally precede m_4
Figure 6.13(b)	$(4,1,0)$	$(2,3,0)$	No	No	m_2 and m_4 may conflict



(b)

Figure 6.13: Capturing potential causality when exchanging messages.

- We use the notation $ts(a) < ts(b)$ if and only if for all k, $ts(a)[k] \leq ts(b)[k]$ and there is at least one index k' for which $ts(a)[k'] < ts(b)[k']$.

- Thus, by using vector clocks, process P3 can detect whether m4 may be causally dependent on m2, or whether there may be a potential conflict.

- Note, by the way, that without knowing the actual information contained in messages, it is not possible to state with certainty that there is indeed a causal relationship, or perhaps a conflict.

6.3 MUTUAL EXCLUSION

Simultaneous access to same resource by multiple processes will corrupt the resource or make it inconvenient. A mutual exclusion is a mechanism to prevent simultaneous access to a shared resource.

Overview

Distributed mutual exclusion algorithms can be classified into two different categories.

1) Token-based approaches
2) Permission-based approaches

Token-based solutions

- In token-based solutions mutual exclusion is achieved by passing a special message between the processes, known as a token.
- There is only one token available and who ever has that token is allowed to access the shared resource.
- When finished, the token is passed on to a next process. If a process having the token is not interested in accessing the resource, it passes it on.
- Key properties: These solutions avoid starvation and deadlock.
- Drawback: when the token is lost a mechanism has to be present to ensure that a new token is created and it is also the only one token.

Permission-based approach

- In this case, a process wanting to access the resource first requires the permission from other processes. Different ways are there to obtain permission.

A centralized algorithm

<u>Simulation of mutual exclusion in a DS in a one-processor system</u>

- 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 Figure 6.15(a). When the reply arrives, the requester can go ahead.

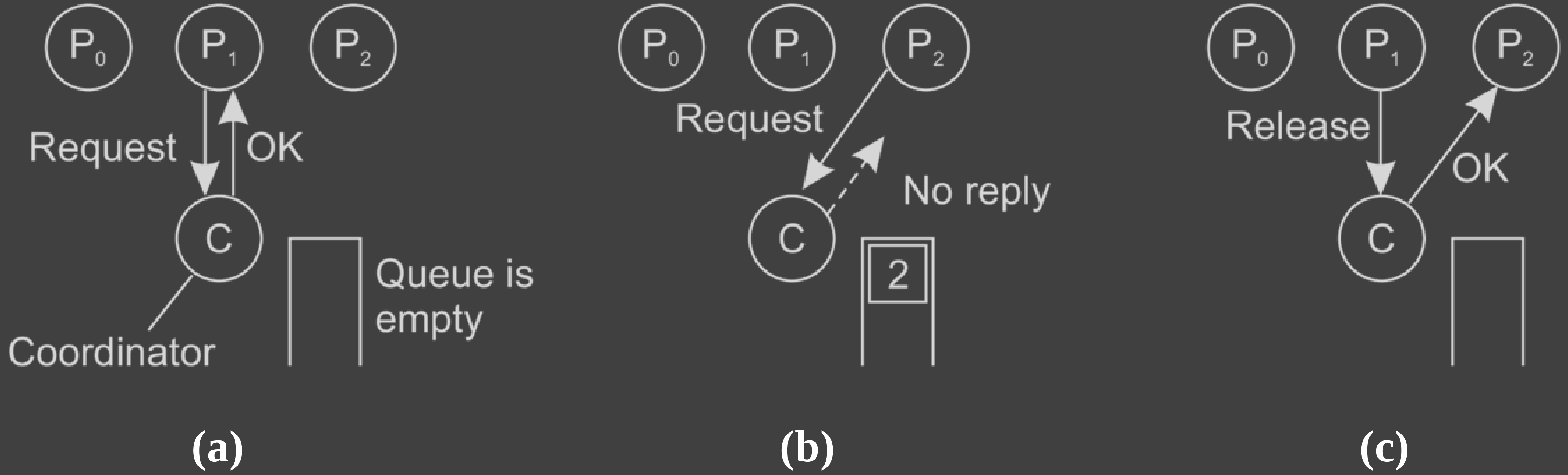


Figure 6.15: (a) Process P1 asks for permission to access a shared resource. Permission is granted. (b) Process P2 asks permission to access the same resource, but receives no reply. (c) When P1 releases the resource, the coordinator replies to P2.

- Now suppose that another process, P2 in Figure 6.15(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 Figure 6.15(b) the coordinator just refrains from replying, thus blocking process P2, which is waiting for a reply.
- Alternatively, it could send a reply saying "permission denied."
- Either way, it queues the request from P2 for the time being and waits for more messages.

- When process P1 is finished with the resource, it sends a message to the coordinator releasing its exclusive access, as shown in Figure 6.15(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.

The algorithm guarantees,

Mutual exclusion: The coordinator lets only one process at a time access the resource.

Fairness: The requests are granted in the order in which they are received.

No starvation: No process ever waits forever.

Easier implementation: It requires only three messages per use of resource (request, grant, release).

Shortcomings of Centralized Approach

- The coordinator is a single point of failure, so if it crashes, the entire system may go down.

- If processes normally block after making a request, they cannot distinguish a dead coordinator from “permission denied” since in both cases no message comes back.

- In addition, in a large system, a single coordinator can become a performance bottleneck.

A distributed algorithm

By using Lamport’s original solution for distributed mutual exclusion, Ricart and Agrawala have provided this algorithm.

Working

- When a process wants to access a shared resource, it builds a message containing the name of the resource, its process number, and the current (logical) time.
- It then sends the message to all other processes, conceptually including itself. The sending of messages is assumed to be reliable; that is, no message is lost.
- When a process receives a request message from another process, the action it takes depends on its own state with respect to the resource named in the message. Three different cases have to be clearly distinguished:

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.

If the incoming message has a lower timestamp, the receiver sends back an OK message. If its own message has a lower timestamp, the receiver queues the incoming request and sends nothing.

- After sending out requests asking permission, a process sits back and waits until everyone else has given permission. As soon as all the permissions are in, it may go ahead. When it is finished, it sends OK messages to all processes in its queue and deletes them all from the queue. If there is no conflict, it clearly works.

- However, suppose that two processes try to simultaneously access the resource, as shown in Figure 6.16(a).

- Process P0 sends everyone a request with timestamp 8, while at the same time, process P2 sends everyone a request with timestamp 12. P1 is not interested in the resource, so it sends OK to both senders.

- Processes P0 and P2 both see the conflict and compare timestamps. P2 sees that it has lost, so it grants permission to P0 by sending OK.

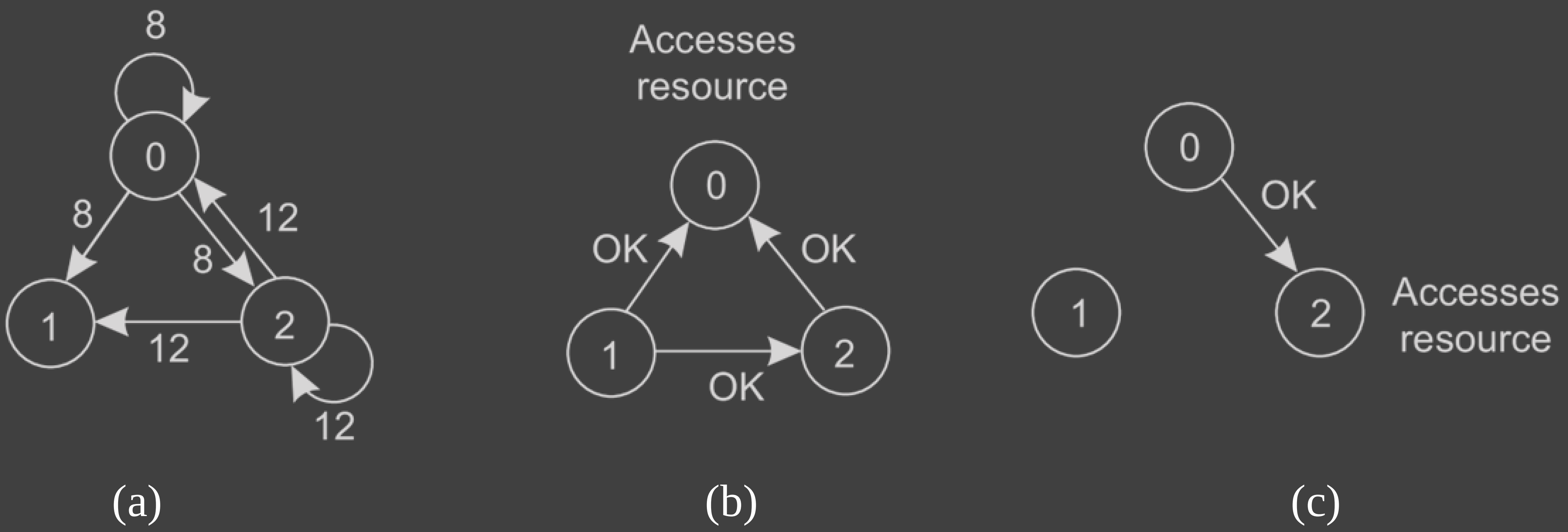


Figure 6.16:
(a) Two processes want to access a shared resource at the same moment.
(b) P0 has the lowest timestamp, so it wins.
(c) When process P0 is done, it sends an OK also, so P2 can now go ahead.

- Process P0 now queues the request from P2 for later processing and accesses the resource, as shown in Figure 6.16(b).

- When it is finished, it removes the request from P2 from its queue and sends an OK message to P2, allowing the latter to go ahead, as shown in Figure 6.16(c).

- The algorithm works because in the case of a conflict, the lowest timestamp wins and everyone agrees on the ordering of the timestamps.

- Mutual exclusion is guaranteed without deadlock or starvation. If the total number of processes is N, then the number of messages that a process needs to send and receive before it can enter its critical section is $2 \times (N - 1)$: N - 1 request messages to all other processes, and subsequently N - 1 OK messages, one from each other process.

Shortcomings

- Unfortunately, this algorithm has N points of failure. If any process crashes, it will fail to respond to requests.

- Either a multicast communication primitive must be used, or each process must maintain the group membership list itself, including processes entering the group, leaving the group, and crashing. The method works best with small groups of processes that never change their group memberships.

- All processes are involved in all decisions concerning accessing the shared resource, which may impose a burden on processes running on resource-constrained machines.

- **Improvement:** The algorithm can be modified to grant permission for resource usage when a process has collected permission from a simple majority of the other processes, rather than from all of them.

A token-ring algorithm

- An approach to achieve mutual exclusion in a DS as shown in Figure 6.17.

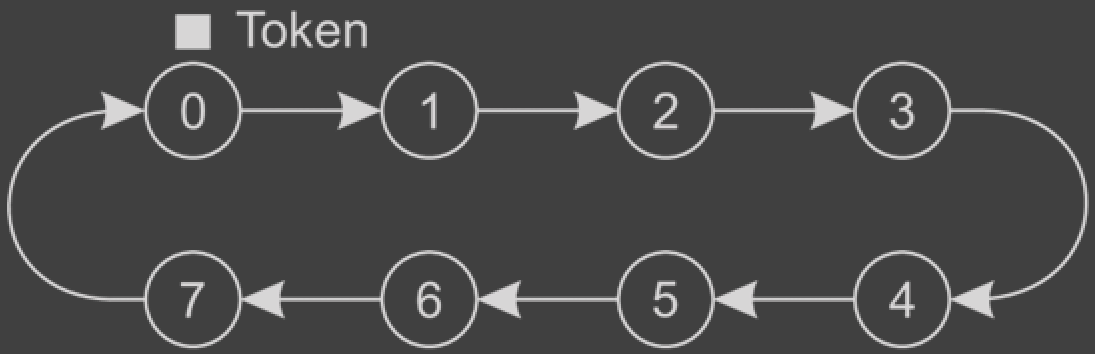


Figure 6.17: An overlay network constructed as a logical ring with a token circulating between its members.

- An overlay network is constructed in the form of a logical ring in which each process is assigned a position in the ring. Each process knows who is next in line after itself.

- When the ring is initialized, process P0 is given a token. The token circulates around the ring. Assuming there are N processes, the token is passed from process Pk to process P(k+1) mod N in point-to-point messages.

- When a process acquires the token from its neighbor, it checks to see if it needs to access the shared resource. If so, the process goes ahead, does all the work it needs to, and releases the resources.

- After it has finished, it passes the token along the ring. It is not permitted to immediately enter the resource again using the same token.

- If a process is handed the token by its neighbor and is not interested in the resource, it just passes the token along. As a consequence, when no processes need the resource, the token just circulates around the ring.

The algorithm guarantees,

Mutual exclusion: Only one process has the token at any instant, so only one process can actually get to the resource.

No starvation: Since the token circulates among the processes in a well-defined order, starvation cannot occur. Once a process decides it wants to have access to the resource, at worst it will have to wait for every other process to use the resource.

Issue with algorithm

Loss of token: The token holder may crash or the message containing the token may get lost.

Possible conflict: Unavailability of token does not mean that it has been lost; somebody may still be using it.

A decentralized algorithm

- A voting algorithm proposed by Lin et al.

- Each resource is assumed to be replicated N times. Every replica has its own coordinator for controlling the access by concurrent processes.

- Whenever a process wants to access the resource, it will simply need to get a majority vote from $m > N/2$ coordinators.

- **Coordinator crash:** It is assumed that when a coordinator crashes, it recovers quickly but will have forgotten any vote it gave before it crashed.

- **Coordinator reset:** Coordinator may reset itself and may forget the previous permission given by it. As a consequence, it may incorrectly grant this permission again to another process after its recovery.

Implementation of decentralized algorithm

- We use a system in which a resource is replicated N times and assume that the resource is known under its unique name rname. We can then assume that the i-th replica is named $rname_i$ which is then used to compute a unique key using a known hash function.

- Every process can generate the N keys given a resource's name, and subsequently look up each node responsible for a replica (and controlling access to that replica) using some commonly used naming system.

- If permission to access the resource is denied (i.e., a process gets less than m votes), it is assumed that it will back off for some randomly chosen time, and make a next attempt later.

Problem with this scheme

If many nodes want to access the same resource, it turns out that the utilization rapidly drops. In that case, there are so many nodes competing to get access that eventually no one is able to get enough votes leaving the resource unused.

6.4 ELECTION ALGORITHMS

- Distributed system (DS) need a coordinator or initiator within the system.

- If all processes are exactly the same, with no distinguishing characteristics, there is no way to select one of them to be special. Consequently, we will assume that each process P has a unique identifier *id*(P).

- In general, election algorithms attempt to locate the process with the highest identifier and designate it as coordinator. It is assumed that every process knows the identifier of every other process.

- The goal of an election algorithm is to ensure that when an election starts, it concludes with all processes agreeing on who the new coordinator is to be.

The bully algorithm

Consider N processes $\{P_{0,} \ldots\ldots P_{N-1}\}$ and let $id(P_k) = k$.

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

1. P_k sends an ELECTION message to all processes with higher identifiers: $P_{k+1}, P_{k+2} \ldots\ldots\ldots P_{N-1}$

2. If no one responds, P_k wins the election and becomes coordinator.

3. If one of the higher-ups answers, it takes over and P_k's job is done.

- At any moment, a process can get an ELECTION message from one of its lower-numbered colleagues. When such a message arrives, the receiver sends an OK message back to the sender to indicate that he is alive and will take over.
- The receiver then holds an election, unless it is already holding one.
- Eventually, all processes give up but one, and that one is the new coordinator.
- It announces its victory by sending all processes a message telling them that starting immediately it is the new coordinator.
- If a process that was previously down comes back up, it holds an election. If it happens to be the highest-numbered process currently running, it will win the election and take over the coordinator’s job. Thus the biggest guy in town always wins, hence the name “bully algorithm.”

Working of the bully algorithm

- Figure 6.20 depicts eight processes with identifiers from 0 to 7.
- Previously process P7 was the coordinator, but it has just crashed.
- P4 is the first one to notice this, so it sends ELECTION messages to all the processes higher than it, namely P5, P6, and P7, as shown in Figure 6.20(a).
- Processes P5 and P6 both respond with OK, as shown in Figure 6.20(b). Upon getting the first of these responses, P4 knows that its job is over, knowing that either one of P5 or P6 will take over and become coordinator.
- Process P4 just sits back and waits to see who the winner will be.

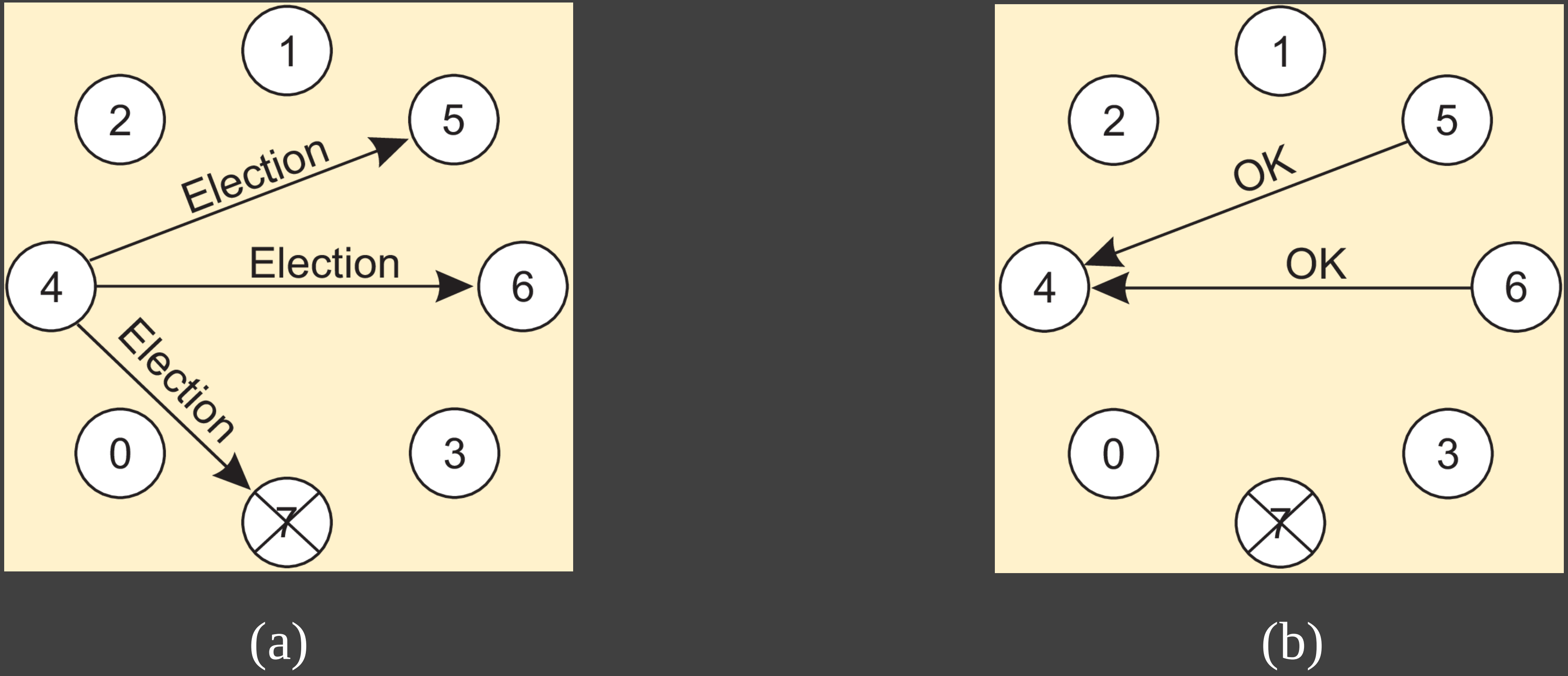


Figure 6.20: The bully election algorithm.
(a) Process 4 holds an election.
(b) Processes 5 and 6 respond, telling 4 to stop.

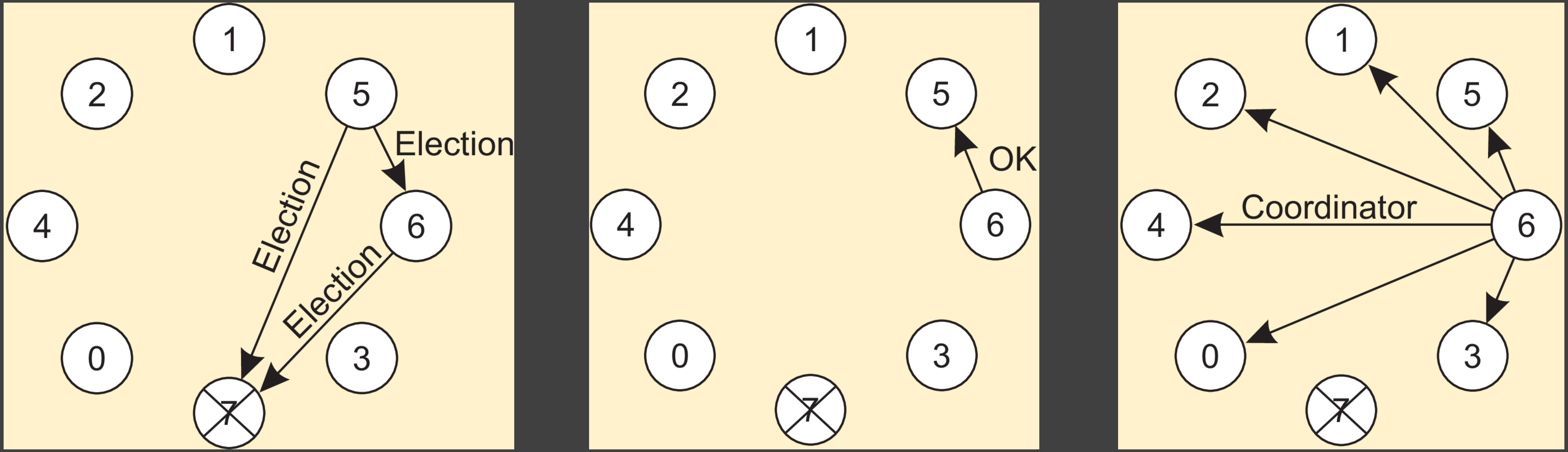
- In Figure 6.20(c) both P5 and P6 hold elections, each one sending messages only to those processes with identifiers higher than itself.

- In Figure 6.20(d), P6 tells P5 that it will take over. At this point P6 knows that P7 is dead and that it (P6) is the winner.

- If there is state information to be collected from disk or elsewhere to pick up where the old coordinator left off, P6 must now do what is needed. When it is ready to take over, it announces the takeover by sending a COORDINATOR message to all running processes. (Figure 6.20(e)).

- When P4 gets this message, it can now continue with the operation it was trying to do when it discovered that P7 was dead, but using P6 as the coordinator this time. In this way the failure of P7 is handled and the work can continue.

- If process P7 is ever restarted, it will send all the others a COORDINATOR message and bully them into submission.



(c) (d) (e)

Figure 6.20: The bully election algorithm.
(c) Now 5 and 6 each hold an election.
(d) Process 6 tells 5 to stop.
(e) Process 6 wins and tells everyone.

A ring algorithm

- The algorithm makes use of a logical ring and it assumes that each process knows its successor.
- When any process notices that the coordinator is not functioning, it builds an ELECTION message containing its own process identifier and sends the message to its successor.
- If the successor is down, the sender skips over the successor and goes to the next member along the ring, or the one after that, until a running process is located.
- At each step along the way, the sender adds its own identifier to the list in the message effectively making itself a candidate to be elected as coordinator.

- Eventually, the message gets back to the process that started it all. That process recognizes this event when it receives an incoming message with its own identifier.

- At that point, the message type is changed to COORDINATOR and circulated once again, this time to inform everyone else who the coordinator is and who the members of the new ring are. The list member with the highest identifier is elected as coordinator. When this message has circulated once, it is removed and everyone goes back to work.

- In Figure 6.21 we see what happens if two processes, P3 and P6, discover simultaneously that the previous coordinator, process P7, has crashed.

- Each of these builds an ELECTION message and each of them starts circulating its message, independent of the other one.

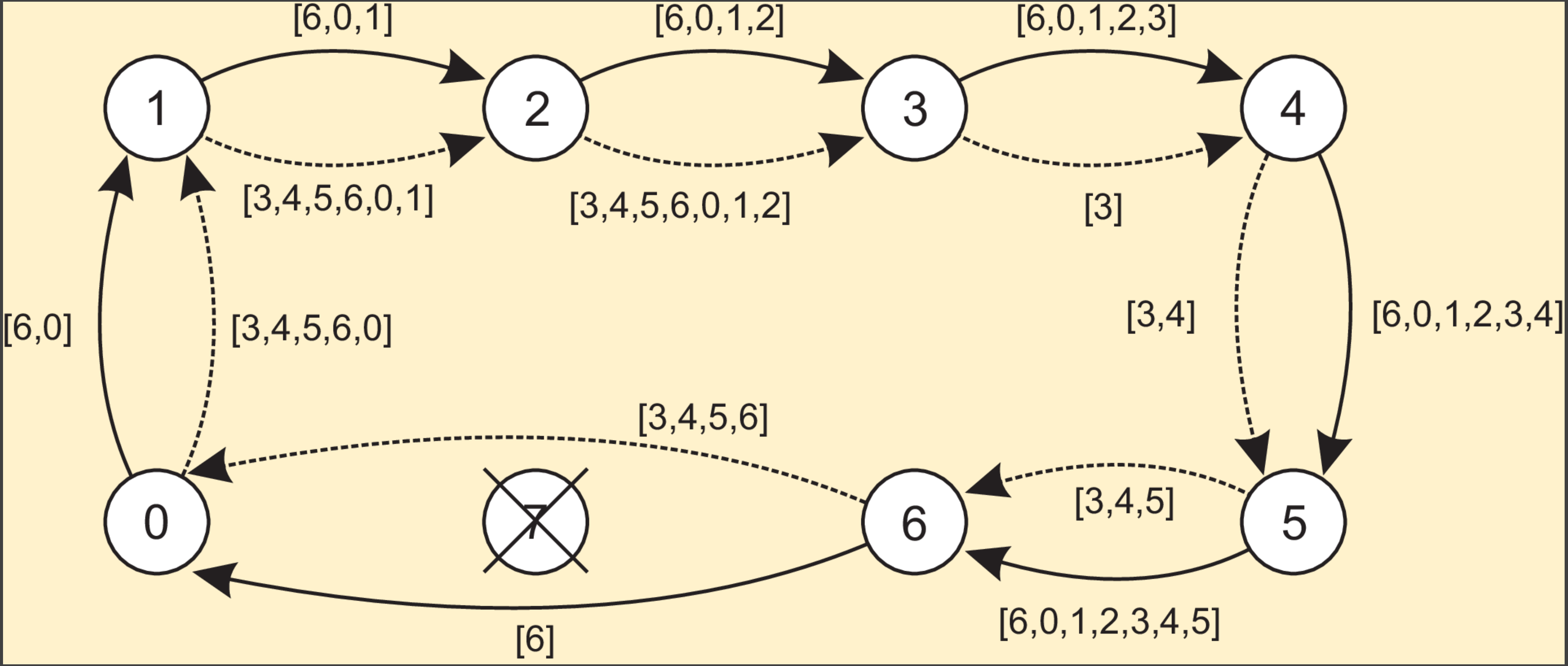


Figure 6.21: Election algorithm using a ring. The solid line shows the election messages initiated by P6; the dashed one those by P3.

- Eventually, both messages will go all the way around, and both P3 and P6 will convert them into COORDINATOR messages, with exactly the same members and in the same order.

- When both have gone around again, both will be removed. It does no harm to have extra messages circulating; at worst it consumes a little bandwidth, but this is not considered wasteful.

Elections in wireless environments

- The assumptions made by traditional election algorithms (such as message passing is reliable and topology is unchanged) are not realistic in wireless environments.
- Vasudevan et al. have proposed a solution in which the ***best*** leader can be elected rather than just a random one.
- The discussion is concentrated only on ad hoc networks and mobility of nodes is ignored.
- To elect a leader, any node in the network, called the source, can initiate an election by sending an ELECTION message to its immediate neighbors (i.e., the nodes in its range).

- When a node receives an ELECTION for the first time, it designates the sender as its parent, and subsequently sends out an ELECTION message to all its immediate neighbors, except for the parent.

- When a node receives an ELECTION message from a node other than its parent, it merely acknowledges the receipt.

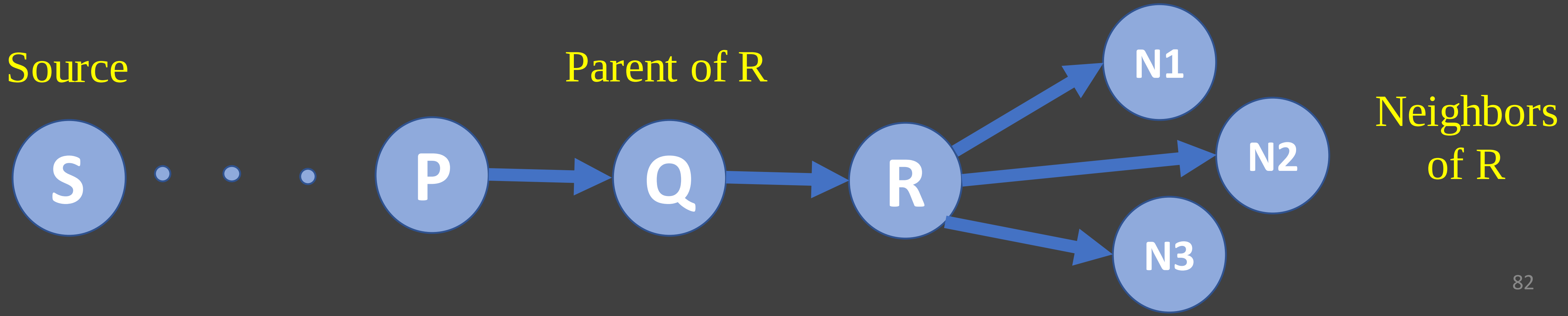
- When node R has designated node Q as its parent, it forwards the ELECTION message to its immediate neighbors (excluding Q) and waits for acknowledgments to come in before acknowledging the ELECTION message from Q.

- This waiting has an important consequence. First, note that neighbors that have already selected a parent will immediately respond to R. More specifically, if all neighbors already have a parent, R is a leaf node and will be able to report back to Q quickly. In doing so, it will also report information such as its battery lifetime and other resource capacities.

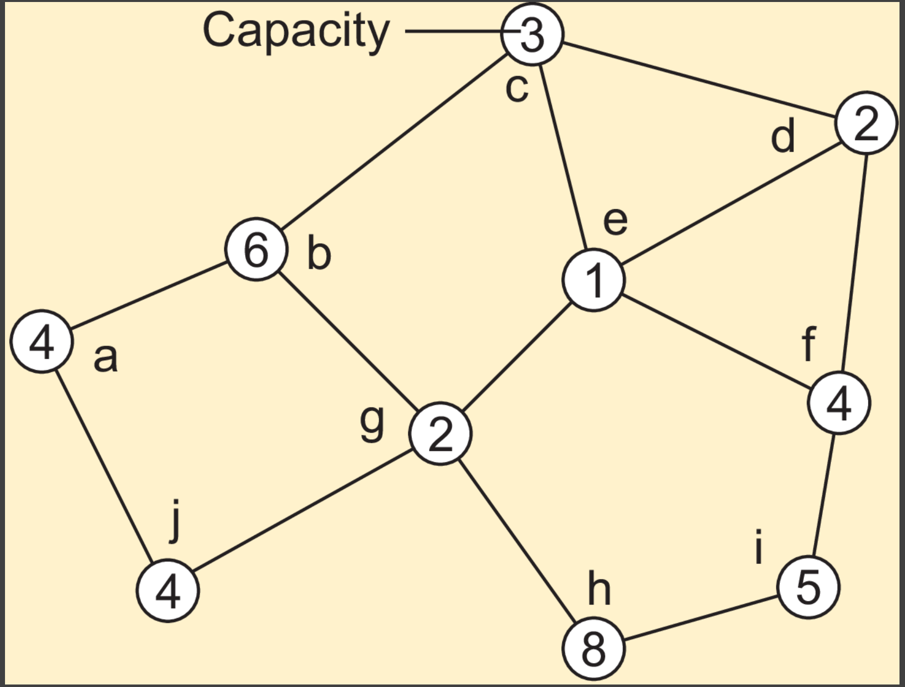
- This information will later allow Q to compare R's capacities to that of other downstream nodes, and select the best eligible node for leadership.

- Of course, Q had sent an ELECTION message only because its own parent P had done so as well.

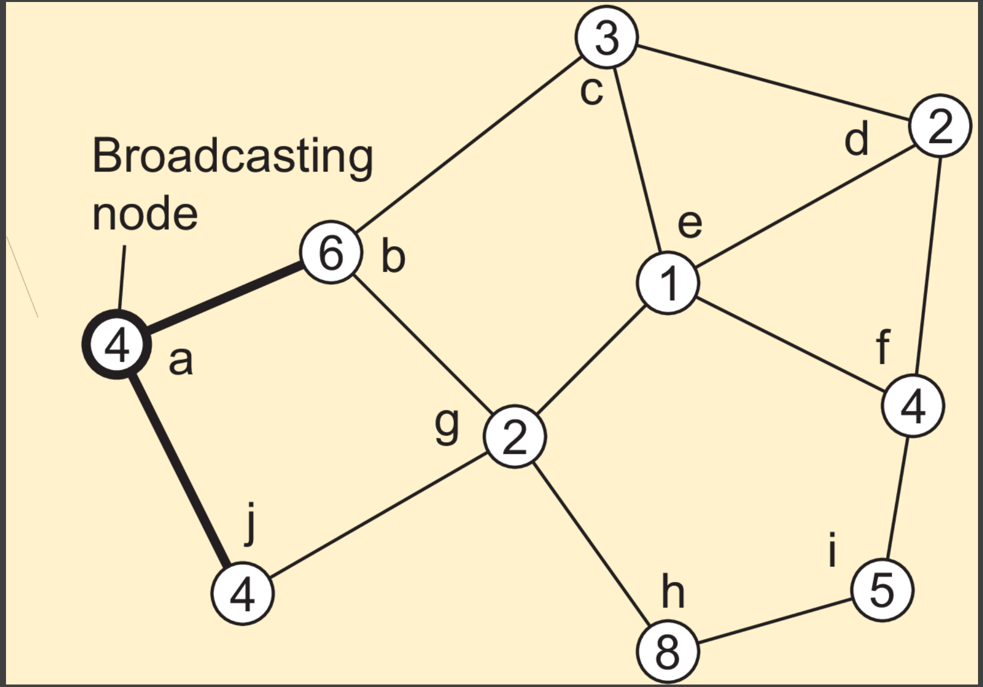
- In turn, when Q eventually acknowledges the ELECTION message previously sent by P, it will pass the most eligible node to P as well. In this way, the source will eventually get to know which node is best to be selected as leader, after which it will broadcast this information to all other nodes.



- This process is illustrated in Figure 6.22. Nodes have been labeled a to j, along with their capacity. Node *a* initiates an election by broadcasting an ELECTION message to nodes *b* and *j*, as shown in Figure 6.22(b)

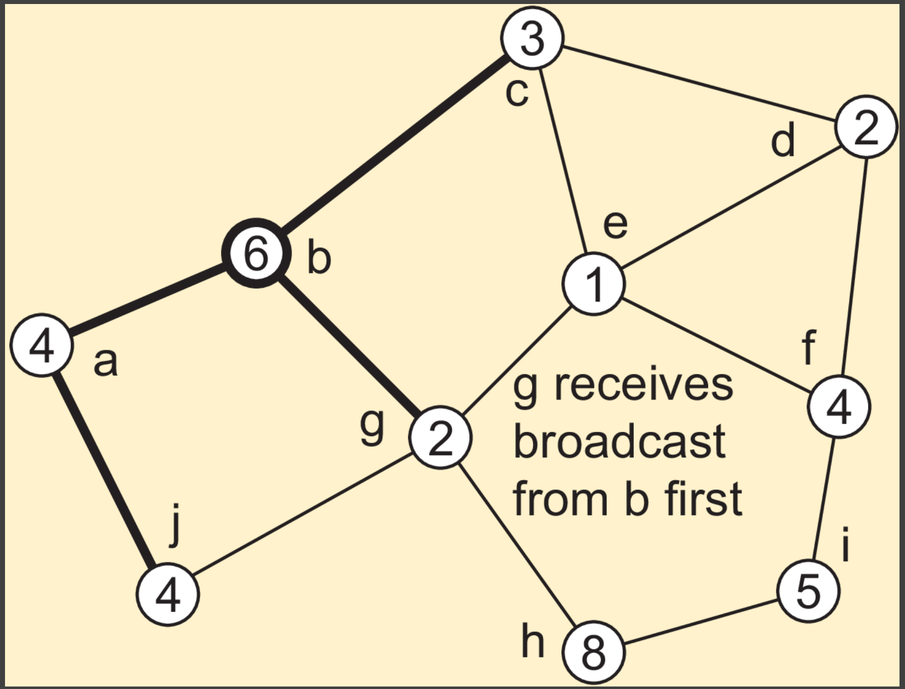


(a)

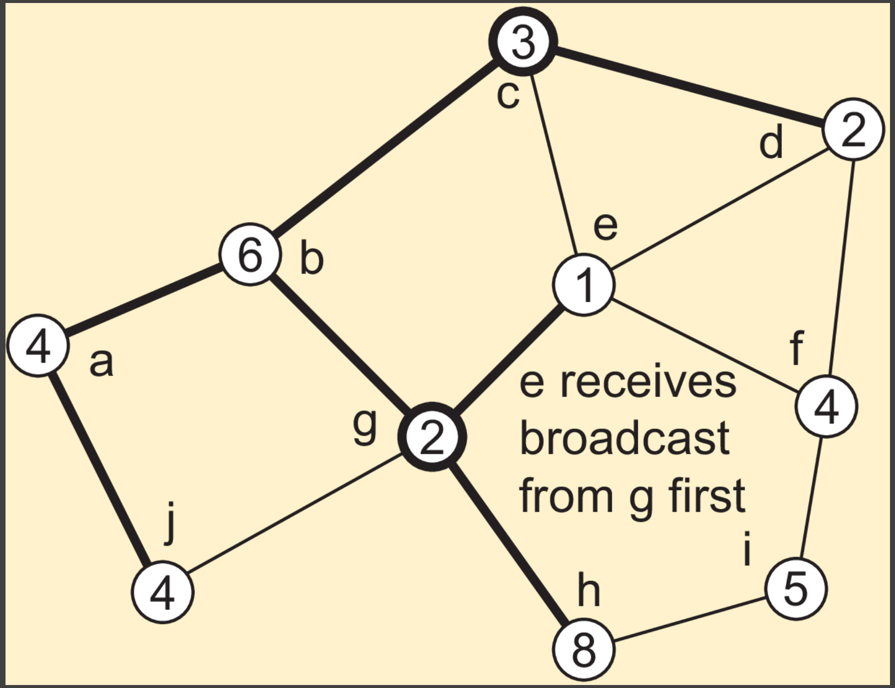


(b)

Figure 6.22: Election algorithm in a wireless network, with node a as the source.
(a) Initial network. (b)–(e) The build-tree phase

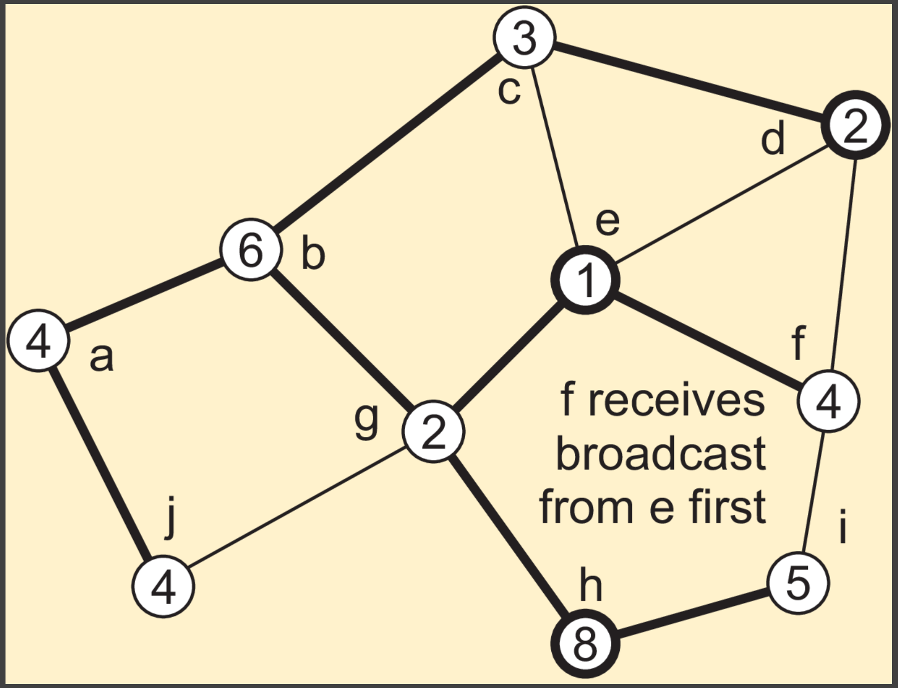


(c)

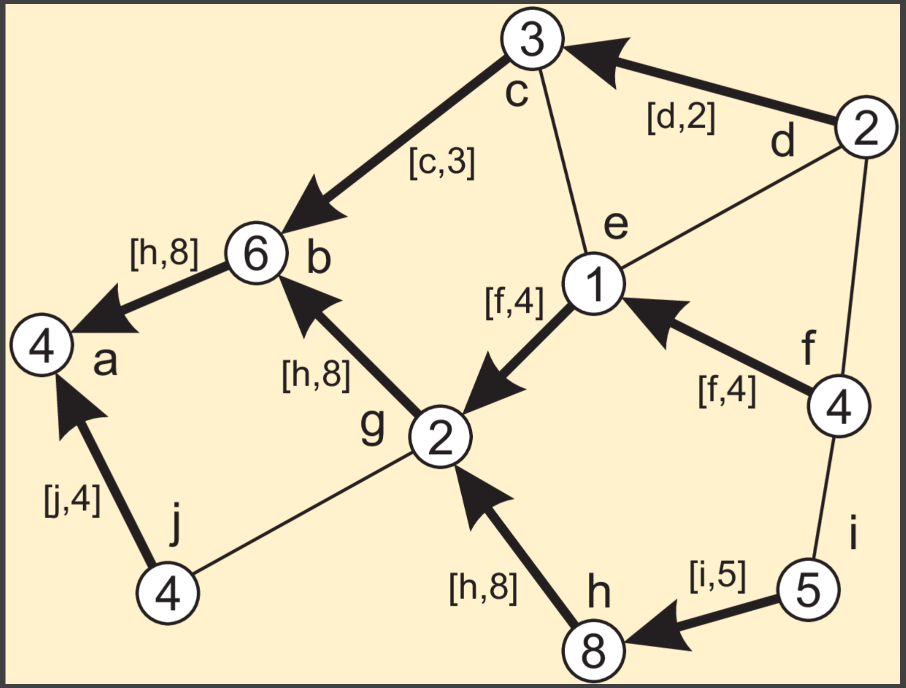


(d)

Figure 6.22: Election algorithm in a wireless network, with node a as the source. (b)–(e) The build-tree phase.



(e)



(f)

Figure 6.22: Election algorithm in a wireless network, with node a as the source. (b)–(e) The build-tree phase (last broadcast step by nodes f and i not shown) (f) Reporting of best node to source.

- ELECTION messages are propagated to all nodes, ending with the situation shown in Figure 6.22(e), where we have omitted the last broadcast by nodes f and i.

- From there on, each node reports to its parent the node with the best capacity, as shown in Figure 6.22(f). For example, when node *g* receives the acknowledgments from its children *e* and *h*, it will notice that h is the best node, propagating *[h, 8]* to its own parent, *node b*.

- In the end, the source will note that h is the best leader and will broadcast this information to all other nodes.

- When multiple elections are initiated, each node will decide to join only one election, the election with the highest identifier.

Elections in large-scale systems

- Many leader-election algorithms apply to only small DS and concentrate on selection of a single node.
- Some situations require several nodes to be selected as in case of super peers in peer-to-peer networks.

Selection of super peers: Following requirements need to be met,

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

- These requirements are easily met using structured Distributed Hash Table (DHT) based systems.

- The basic idea is to reserve a fraction of the identifier space for super peers.

- Each node receives a random and uniformly assigned ***m-bit*** identifier. Now suppose we reserve the first (i.e., leftmost) ***k*** bits to identify super peers.

- **Example:** Assume we have a (small) Chord system with m = 8 and k = 3. When looking up the node responsible for a specific key K, we can first decide to route the lookup request to the node responsible for the pattern K^11100000 which is then treated as the superpeer.

Note: Binary operator '^' denotes a bitwise *and*.

- Another approach is based on positioning nodes in an m-dimensional geometric space. Assume we need to place N super peers evenly throughout the overlay.

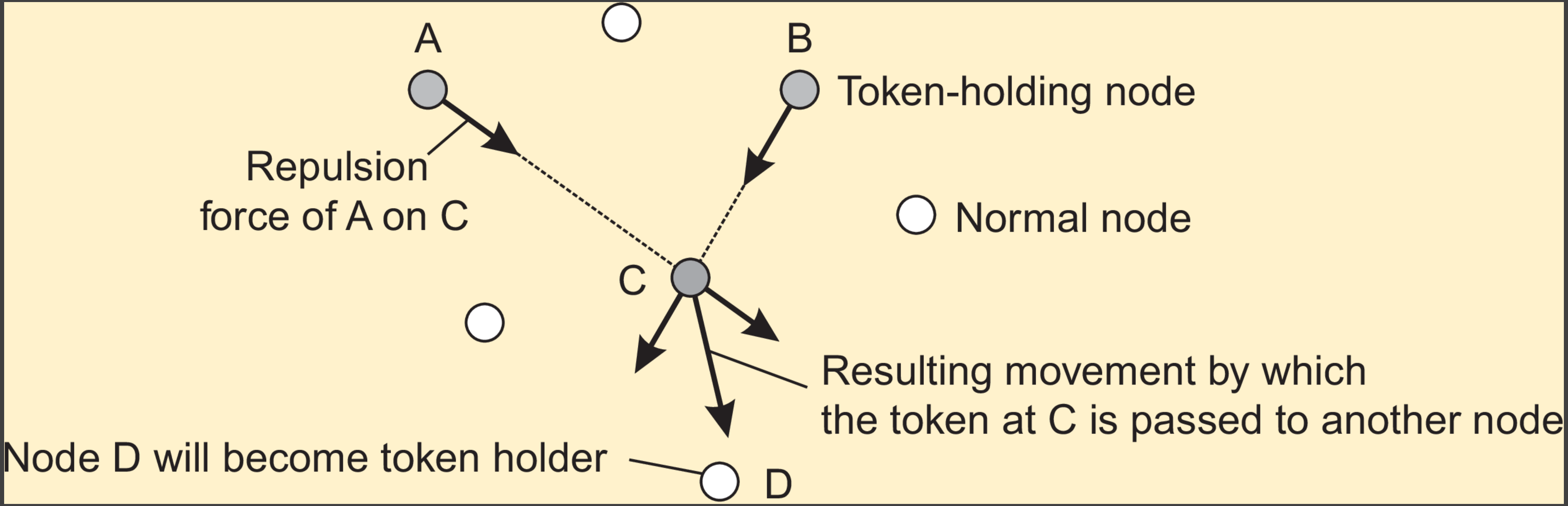
- The basic idea is simple: a total of N tokens are spread across N randomly chosen nodes. No node can hold more than one token.

- Each token represents a repelling force by which another token is inclined to move away. The net effect is that if all tokens exert the same repulsion force, they will move away from each other and spread themselves evenly in the geometric space.

- This approach requires that nodes holding a token learn about other tokens. To this end, we can use a gossiping protocol by which a token’s force is disseminated throughout the network.

- If a node discovers that the total forces that are acting on it exceed a threshold, it will move the token in the direction of the combined forces, as shown in Figure 6.23.

- When a token is held by a node for a given amount of time, that node will promote itself to superpeer.



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.

END