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

Chapter 6
Synchronization

*** Modified by Prof. Rivalino Matias, Jr.**

Outline

- Clock synchronization.
- Mutual exclusion.
- Election

Introduction

- 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 process A asks for the time, and then a little later process B asks for the time, the value that B gets will be higher than (or possibly equal to) the value A got. It will certainly not be lower.
- In a DS, agreement on time is not trivial.
  - Each node has a clock device to keep the notion of time.
  - It is not possible to keep different local clocks precisely synchronized, due to physical restrictions.
  - This may cause different problems in distributed applications that are sensitive to time.
  - For example, think how **make** would work in a distributed software development environment.

Introduction

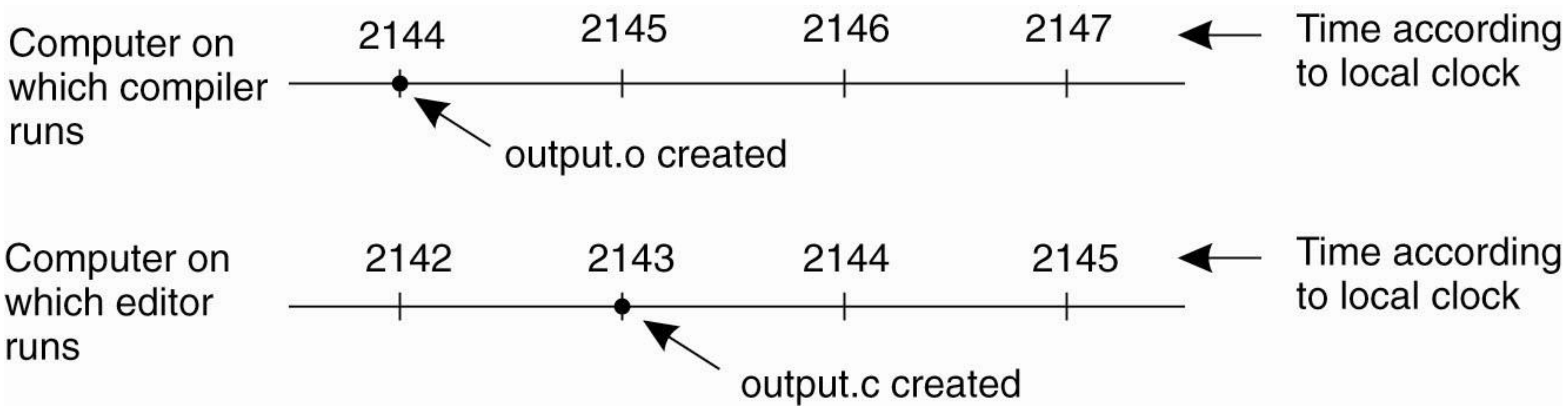

Computer on which compiler runs
2144
2145
2146
2147
Time according to local clock
output.o created
Computer on which editor runs
2142
2143
2144
2145
Time according to local clock
output.c created

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

- All computers have a circuit for keeping track of time.
  - They are not actually clocks in the usual sense. **Timer** is a better word. The timer is a precisely machined quartz crystal.
  - Quartz crystals oscillate at a well-defined frequency that depends on the kind of crystal, how it is cut, and the amount of tension.
  - Associated with each crystal are 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**.

Physical Clocks

- Computers have a battery-backed up CMOS RAM
  - Date and time are kept in this CMOS memory.
  - At every clock tick, the ISR adds one to the time stored in this memory; this way the **(software) clock** is kept up to date.
  - It works well for single computers with single clocks, since all processes in the machine use the same clock.
- In scenarios of multiple nodes, each node has its own clock.
  - It is (physically) **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**.

Clock Synchronization

- DS and clock skew.
  - Because of the *clock skew*, programs that expect the time associated with a file, object, process, or message to be correct and independent of the machine on which it was generated (i.e., which clock it used) can fail, as we saw in the make example above.
- In some DS, the actual clock time is important.
  - For these cases, external physical clocks are needed.
  - For reasons of efficiency and redundancy, multiple physical clocks are generally considered desirable, which yields two problems:

    How do we synchronize them with real-world clocks?

    How do we synchronize the clocks with each other?

Clock Synchronization

- Physical Clock References
  - **International Atomic Time**:
    - Currently, several laboratories around the world have cesium 133 clocks. Periodically, each laboratory tells the Bureau International de l'Heure (BIH) in Paris how many times its clock has ticked.
    - The BIH averages these to produce International Atomic Time, which is abbreviated to TAI.
    - Thus, TAI is just the mean number of ticks of the cesium 133 clocks since midnight on Jan. 1, 1958 (the beginning of time) divided by 9,192,631,770.
    - Although TAI is highly stable and available to anyone who wants to go to buying a cesium clock, it is not highly precise along the time.

Clock Synchronization

- Physical Clock References
  - **Broadcast Stations & Satellites**
    - The BIH (see previous slide) standardizes the time around the globe, which is known as UTC (Coordinated Universal Time).
    - Approx. 40 shortwave radio stations (WWV) around the world broadcast a short pulse at the start of each UTC second.
    - The Geostationary Operational Environment Satellite (GOES) can provide UTC accurately to 0.5 msec, and some other satellites do even better.
    - By combining receptions from several satellites, ground time servers can be built offering an accuracy of 50 nsec.
    - UTC receivers are commercially available, and many computers are equipped with one.

Clock Sync: 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**.
- Many algorithms have been proposed for doing this synchronization.

Clock Sync: Algorithms

- All algorithms have the same underlying model:
  - A **software clock** is derived from computer's hardware clock. The **hardware clock** is assumed to cause an interrupt *f* times per sec.
  - When the timer goes off, the *interrupt handler* adds 1 to a counter that keeps track of the nr. of ticks (interrupts) since some agreed-upon time in the past.
  - This counter acts as a software clock C, resonating at frequency F. When the UTC time is t, denote by $C_p(t)$ the value of the software clock on machine p.
  - The goal of clock synchronization algorithms is to keep the deviation between the machines' clocks, in a DS, within a specified bound, known as the **precision** π:

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

  - When considering an external reference, like UTC, it is known as **accuracy** (α): $\forall t, \forall p\colon\ |C_p(t) - t| \leq \alpha$

Clock Sync: Algorithms

- The whole idea of clock synchronization is that:
  - We keep clocks precise, referred to as **internal synchronization** or accurate, known as **external synchronization**.
  - Hence, being precise does not allow us to conclude anything about the accuracy of clocks.
  - Unfortunately, hardware clocks, and thus also software clocks, are subject to **clock drift**:
    - Because their frequency is not perfect and affected by **external sources** such as temperature, clocks on different machines will gradually start showing different values for time.
    - This is known as the **clock drift rate**, i.e., the difference per unit of time from a perfect reference clock.
    - A typical quartz-based hardware clock has a clock drift rate of some 10^{-6} seconds per second (approx. 31.5 sec / year).

Clock Sync: Algorithms

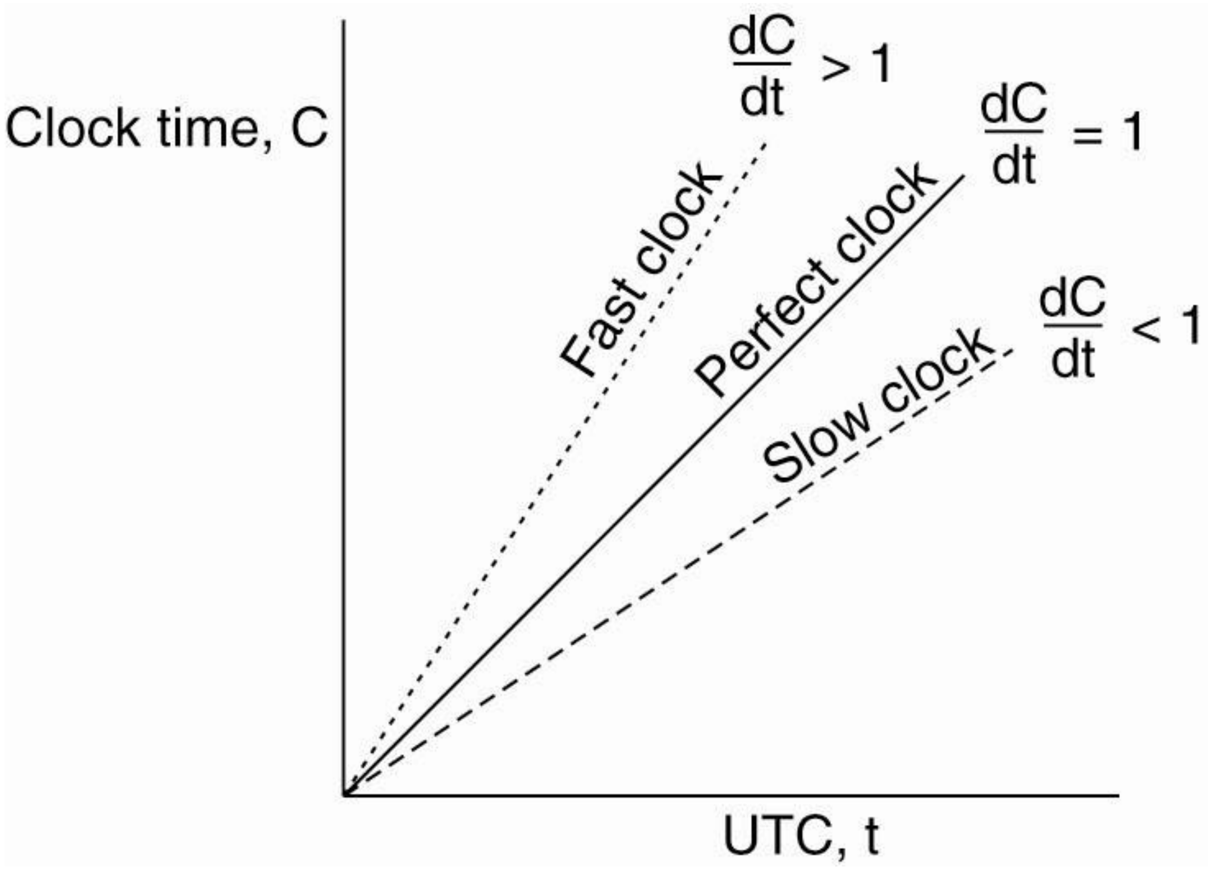


Figure 6-5. The relation between clock time and UTC when clocks tick at different rates.

Clock Sync: Algorithms

- Clock drifting from UTC:
    - If two clocks are drifting from UTC in the opposite direction, at a time Δt after they were synchronized, they may be as much as $2\rho \times \Delta t$ apart; ρ is the **maximal clock drift rate**.
    - If the system designers want to guarantee a precision π, i.e., that no two clocks ever differ by more than π seconds, clock must be resynchronized (in software) at least every $\pi/(2\rho)$ seconds.
    - The various algorithms differ in precisely how this resynchronization is done.

Clock Sync: Algorithms

- Christian’s Algorithm
  - A common approach in many protocols is to let clients contact a **time server** (Cristian, 1989).
  - The time server can be equipped with a UTC receiver to accurately provide the current time.
  - **Problem**: messages delay will have outdated the reported time.
  - **Solution**: to find a good estimation for these delays.

Clock Sync: Algorithms

- Christian's Algorithm - Estimate the offset between two nodes:
  - **A** will send a request to **B** (server), timestamped with value T_1.
  - **B** will record the time of receipt T_2 (taken from its local clock) and returns a response timestamped with value T_3, and piggybacking the recorded value T_2.
  - **A** records the time of the response's arrival, T_4.
  - Assume the propagation delay from **A** to **B** is the same as **B** to **A**, $\delta T_{\mathbf{req}} = T_2 - T_1 \approx T_4 - T_3 = \delta T_{\mathbf{res}}$
  - Thus, **A** can estimate its offset relative to **B** as:

$$\theta = \frac{(T2 - T1) + (T4 - T3)}{2}$$

If **A**'s clock is fast, $\theta < 0$, which means that **A** should set its clock backward; not promptly, but gradually.

Clock Sync: Algorithms

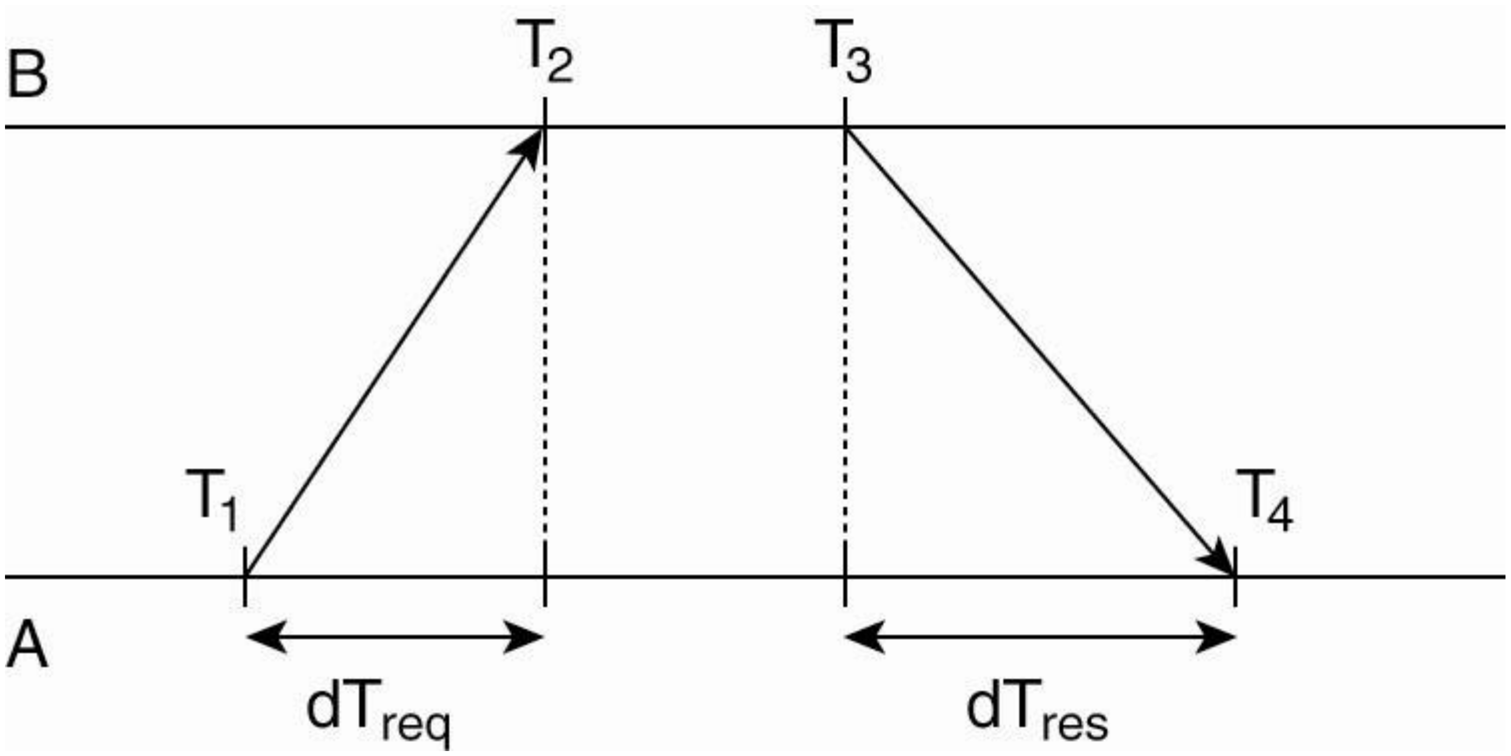


Figure 6-6. Getting the current time from a time server.

Clock Sync: Algorithms

- Network Time Protocol (1)
  - First, this protocol is set up pairwise between servers.
  - **A** and **B** are servers and will probe each other.
  - The offset (θ) between servers is computed as given before.
  - Along computing θ, it is necessary to estimate the delay δ:

$$\delta = \frac{(T4 - T1) - (T3 - T2)}{2}$$

  - Eight pairs of (θ, δ) are buffered, taking the minimal value found for δ as the best estimation for the delay between two servers, and the associated value θ as the most reliable estimation of the offset.
  - Applying NTP symmetrically should, in principle, also let **B** adjust its clock to that of **A**.
    - if **B**'s clock is known to be more accurate, then such an adjustment would be foolish.

Clock Sync: Algorithms

- Network Time Protocol (2)
  - First, this protocol is set up pairwise between servers.
  - **A** and **B** are servers and will probe each other.
  - The offset (θ) between servers is computed as given before.
  - Along computing θ, it is necessary to estimate the delay δ:

$$\delta = \frac{(T4 - T1) - (T3 - T2)}{2}$$

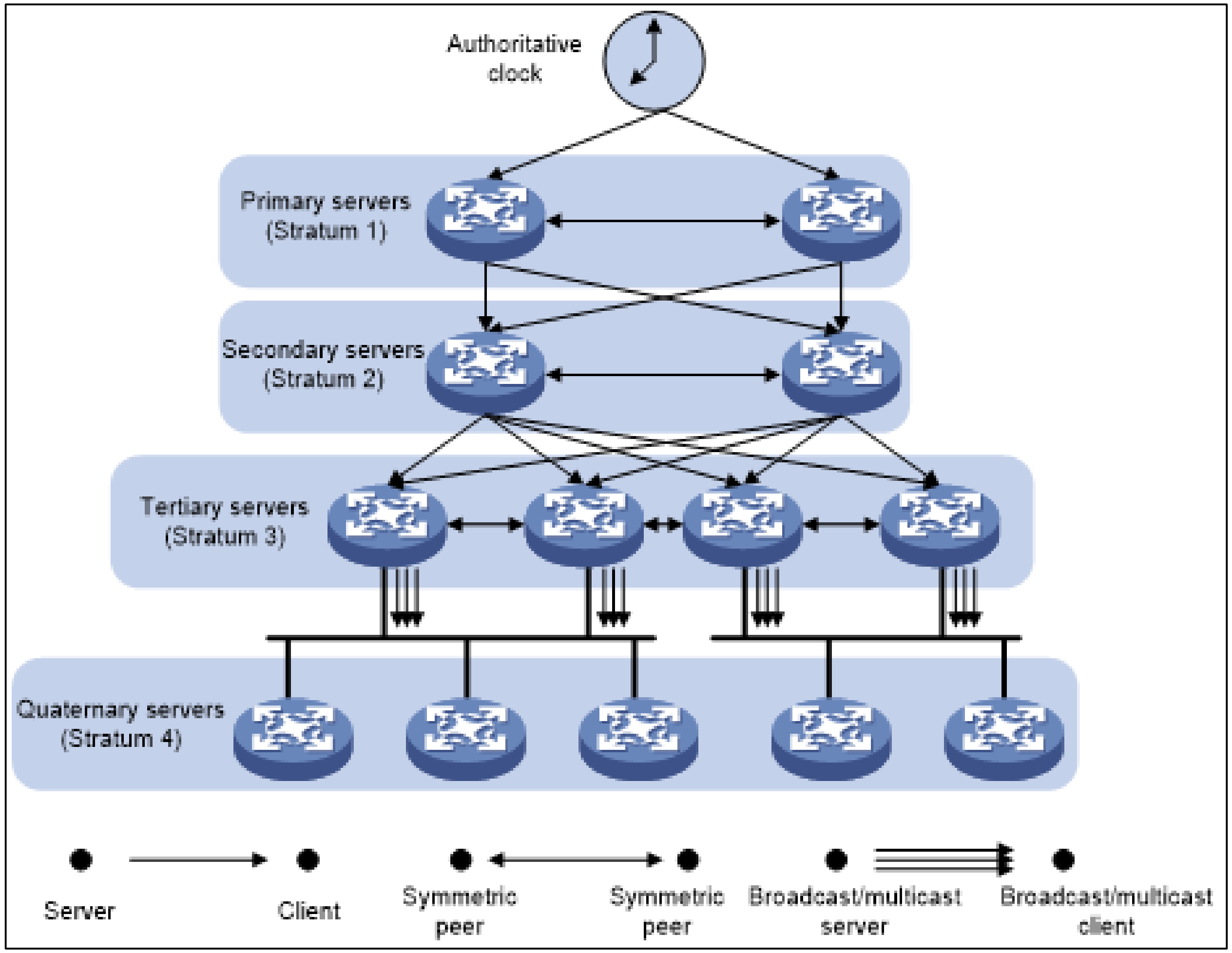
  - Eight pairs of (θ, δ) are buffered, taking the minimal value found for δ as the best estimation for the delay between two servers, and the associated value θ as the most reliable estimation of the offset.
  - Applying NTP symmetrically should, in principle, also let **B** adjust its clock to that of **A**.
    - if **B**'s clock is known to be more accurate, then such an adjustment would be foolish.

Clock Sync: Algorithms

- Network Time Protocol (3)
  - To solve this inaccuracy synchronization problem, NTP divides servers into strata.
  - A server with a reference clock such as a UTC receiver or an atomic clock is known to be a **stratum-1 server**.
  - When **A** contacts **B**, it will adjust only its time if its own stratum level is higher than that of **B**.
  - Due to the symmetry of NTP, if **A**'s stratum level was lower than that of **B**, **B** will adjust itself to **A**.
  - NTP is known to achieve (worldwide) accuracy in the range of 1–50 msec.

Clock Sync: Algorithms

Hierarchy of NTP Stratum Servers


Authoritative clock
Primary servers (Stratum 1)
Secondary servers (Stratum 2)
Tertiary servers (Stratum 3)
Quaternary servers (Stratum 4)
Server
Client
Symmetric peer
Symmetric peer
Broadcast/multicast server
Broadcast/multicast client

techhub.hpe.com

Clock Sync: Algorithms

- Berkeley's algorithm
  - In many clock synchronization algorithms, the time server is passive. Other machines periodically ask it for the time.
    - All it does is respond to their queries.
  - In Berkeley Unix is the opposite:
    - 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.

The Berkeley Algorithm (1)

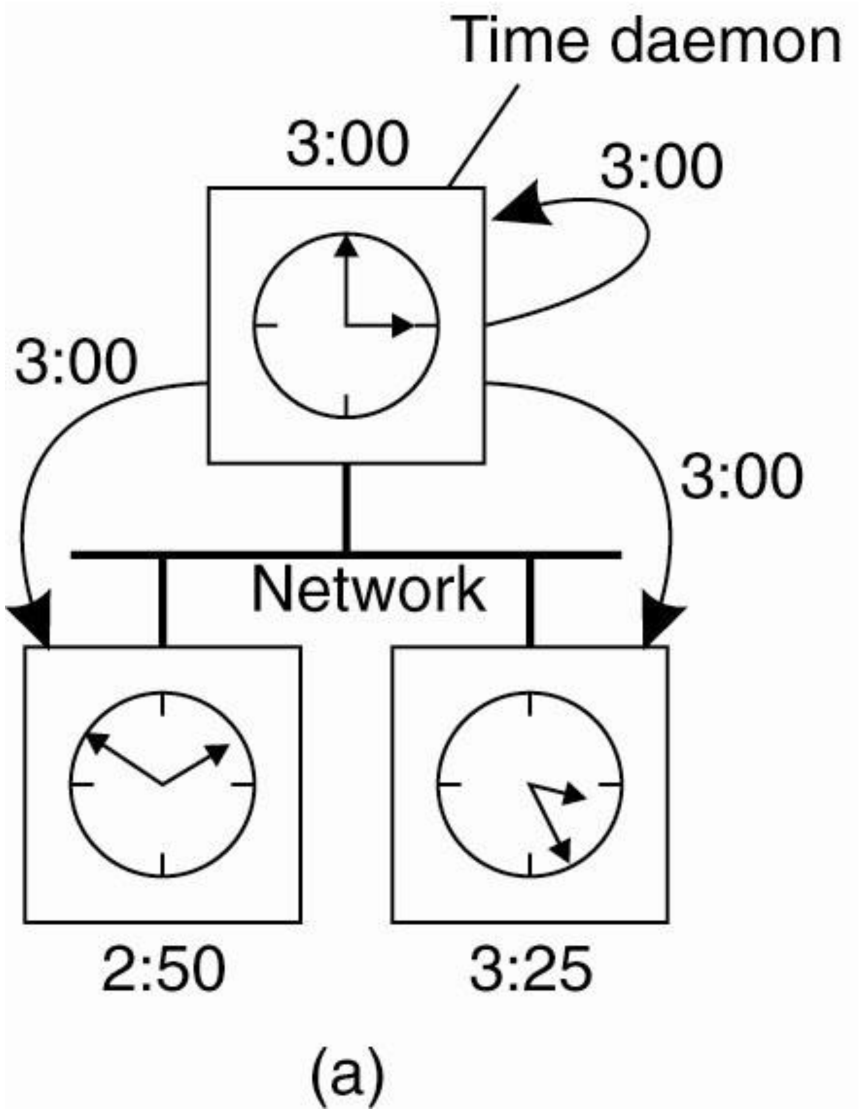


Figure 6-7. (a) The time daemon asks all the other machines for their clock values.

The Berkeley Algorithm (2)

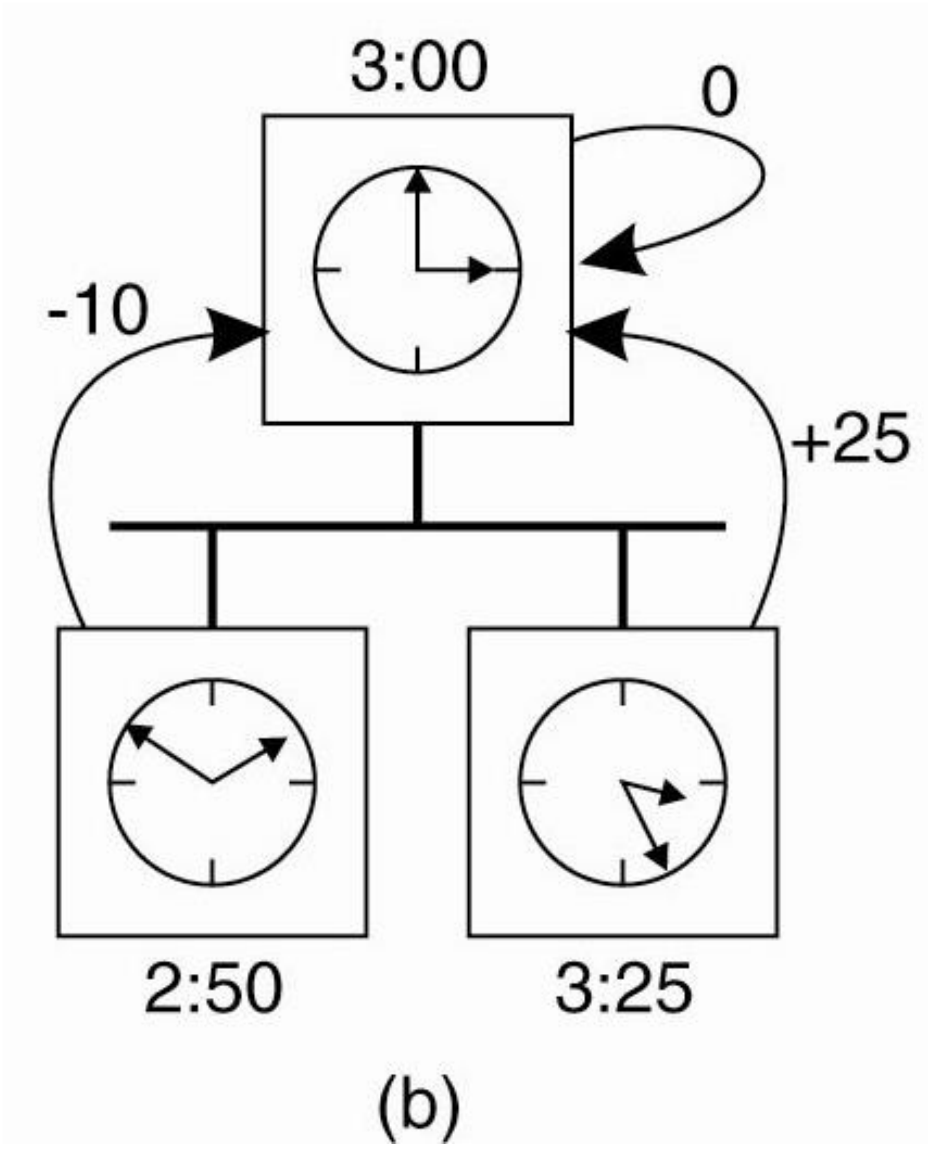


Figure 6-7.

(b) The machines answer.

The Berkeley Algorithm (3)

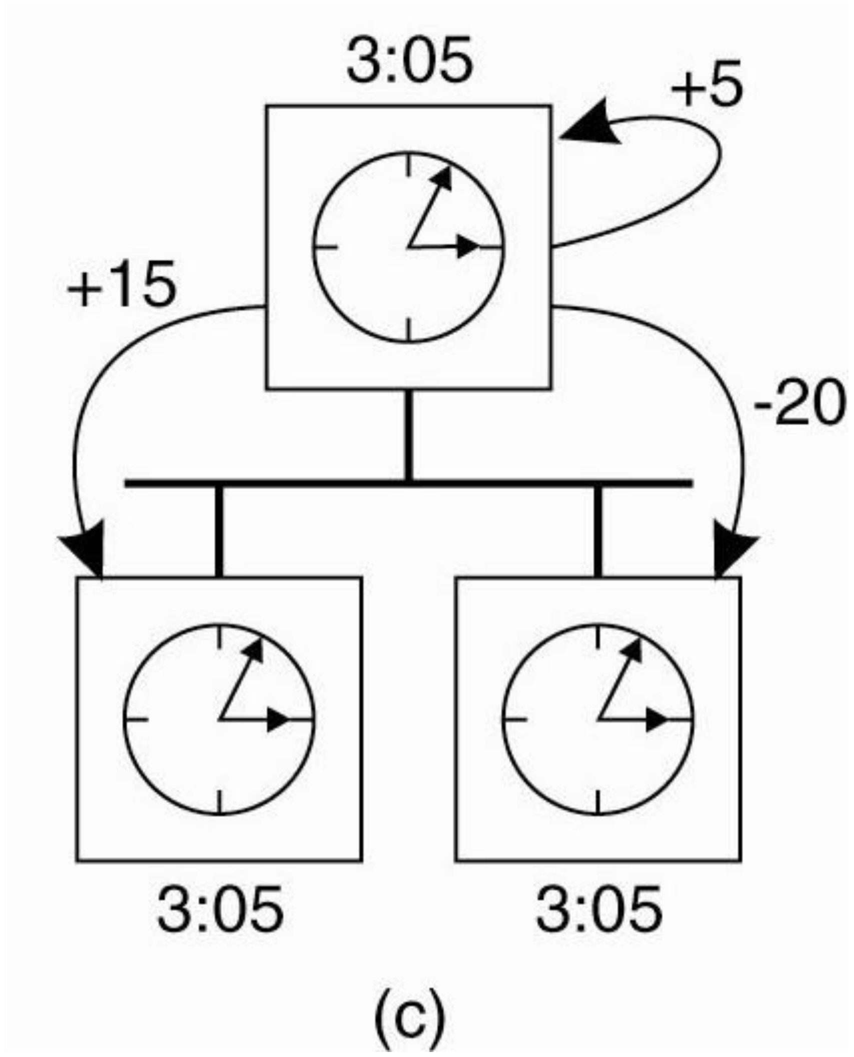


Figure 6-7. (c) The time daemon tells everyone how to adjust their clock.

Logical Clocks

- Clock synchronization is related to time, although it may not be necessary to have an accurate account of the real time.
  - It may be sufficient that every node in a DS agrees on a current time.
  - For running **make** it is adequate that two nodes agree that *input.o* is outdated by a new version of *input.c*, for example.
  - In this case, keeping track of each other's events (such as a producing a new version of *input.c*) is what matters.
  - For these algorithms, it is conventional to speak of the clocks as logical clocks.

Logical Clocks

- Lamport’s logical clocks.
  - In a seminal paper, Lamport [1978] showed that although clock synchronization is possible, it need not be absolute.
  - If two processes do not interact, it is not necessary that their clocks be synchronized because the lack of synchronization would not be observable and thus could not cause problems.
  - Furthermore, he pointed out that what usually matters is not that all processes agree on exactly what time it is, but rather that they agree on the order in which events occur.
  - In the **make** example, what counts is whether *input.c* is older or newer than *input.o*, not their respective absolute creation times.

Logical Clocks

- Lamport's logical clocks.
  - To synchronize logical clocks, Lamport defined a relation called **happens-before**.
  - The expression **a** → **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:
    - 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** 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.

Logical Clocks

- Lamport’s logical clocks.
  - **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 no 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.

Logical Clocks

- Lamport's logical clocks.
  - How do we maintain a global view on the system's behavior that is consistent with the happens-before relation?
    - 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 a property that if **a** $\rightarrow$ **b**, then 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.

Logical Clocks

- Let us look at the algorithm Lamport proposed:
  - Consider three processes (P1,P2, P3).
    - The processes run on different machines, each with its own clock.
    - Assume that a clock is implemented as a software counter, which 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 P2, and 10 units in P3, respectively.
    - Thus, the Lamport clocks are, in fact, **event counters**.
    - Picture on the next two slides illustrate the above-mentioned scenario.

Lamport's Logical Clocks (1)

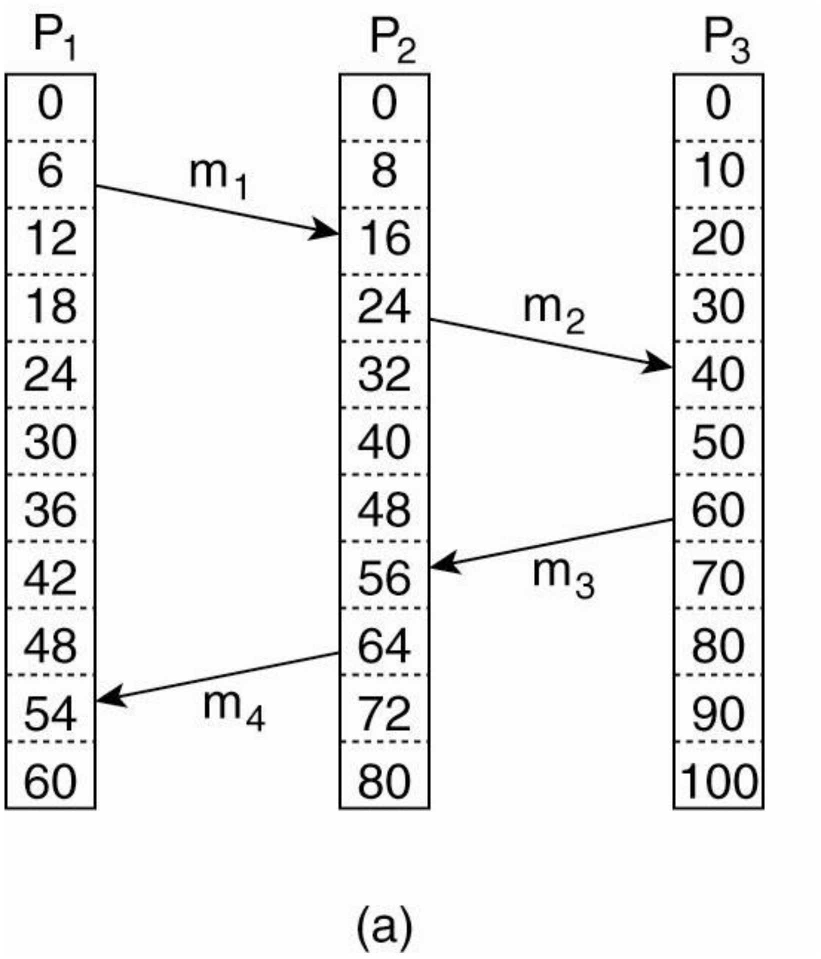


Figure 6-9. (a) Three processes, each with its own clock. The clocks run at different rates.

Lamport's Logical Clocks (2)

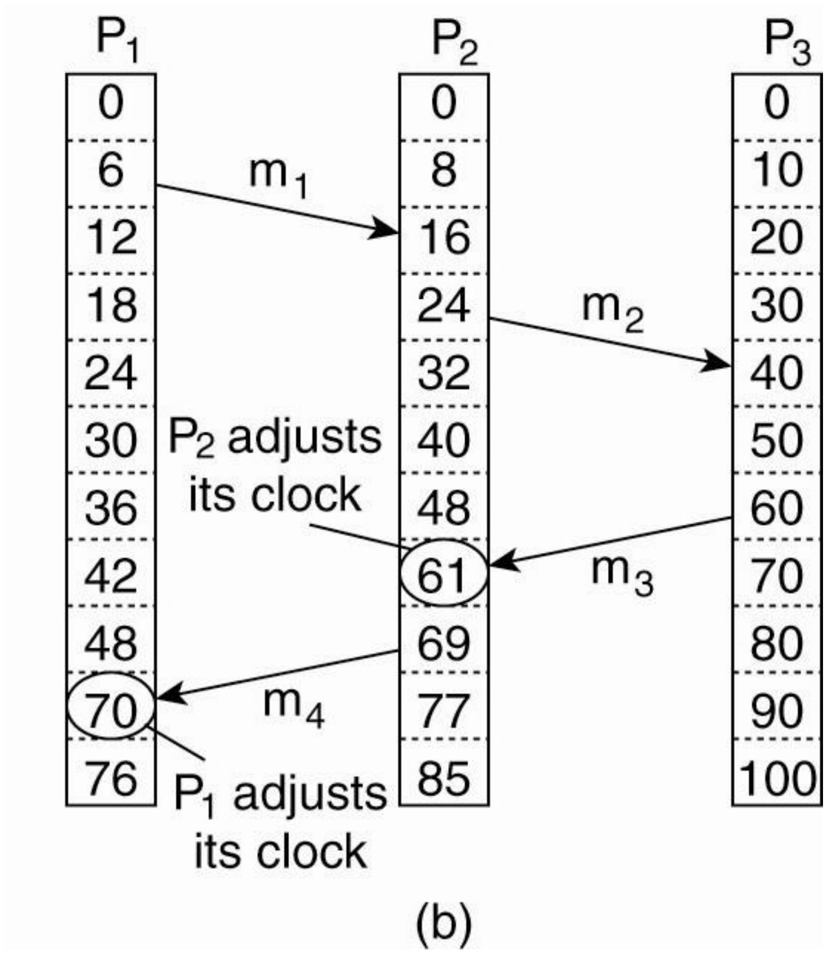


Figure 6-9. (b) Lamport's algorithm corrects the clocks.

Logical Clocks

- Let us look at the algorithm Lamport proposed:
  - 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 6.9, we see that *m3* now arrives at 61. Similarly, *m4* arrives at 70.
  - Let us formulate this procedure more precisely (see next slide).

Logical Clocks

- Let us look at the algorithm Lamport proposed:
  - To implement Lamport's logical clocks, each process Pi maintains a local counter Ci.
  - These counters are updated according to the following steps:
    - Before executing an event, Pi increments Ci: Ci $\leftarrow$ Ci + 1.
    - 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.
    - Upon the receipt of a message *m*, process Pj adjusts its own local counter as Cj $\leftarrow$ max{Cj, ts(m)} after which it then executes the first step and delivers the message to the application.

Lamport's Logical Clocks

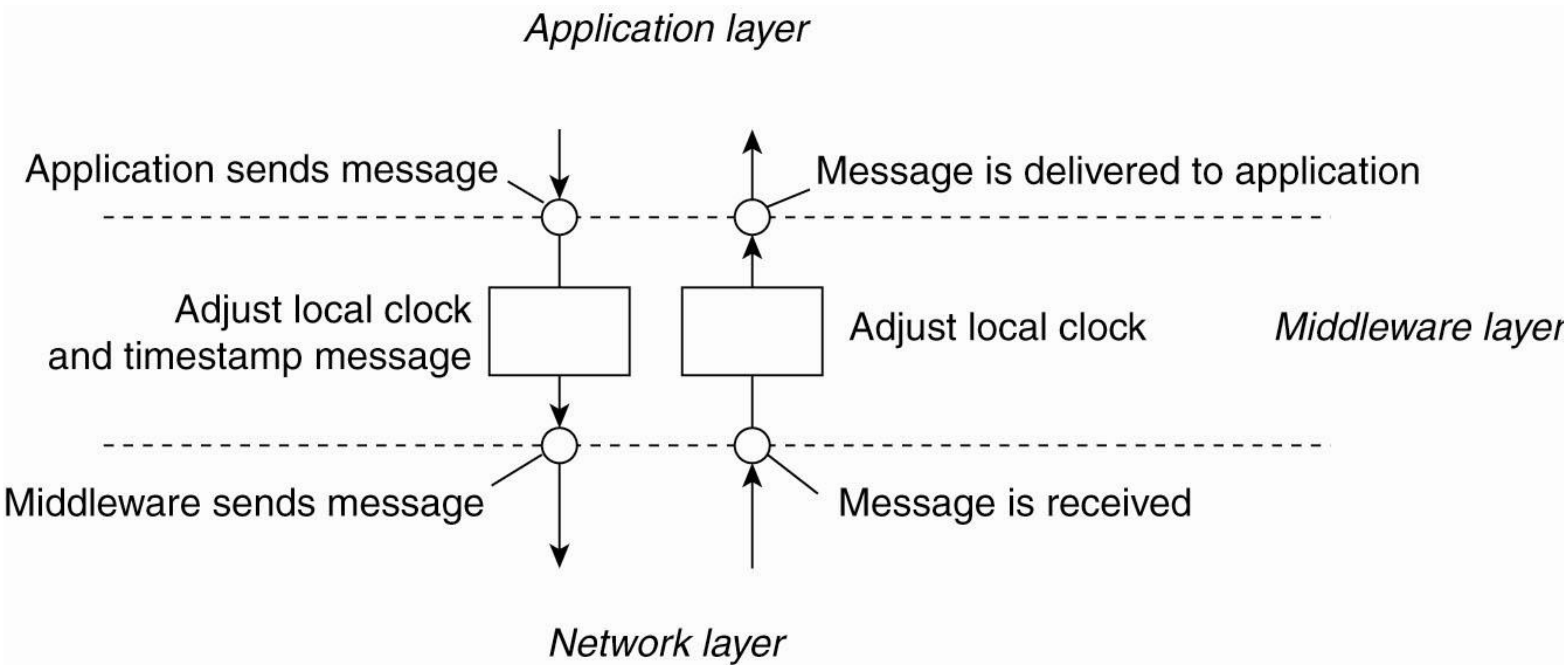


Figure 6-10. The positioning of Lamport's logical clocks in distributed systems.

Logical Clocks

- Let us look at the algorithm Lamport proposed:
  - In some situations, an additional requirement is desirable: no two events ever occur at the same time.
    - To achieve this goal, we also use the unique process identifier to break ties and use tuples instead of only the counter's values.
    - For example: an event at time 40 at process Pi will be timestamped as $\langle 40, i \rangle$. If we also have an event $\langle 40, j \rangle$ and $i < j$, then $\langle 40, i \rangle < \langle 40, j \rangle$.
    - Note that by assigning the event time C(**a**) ← Ci(**a**) if **a** happened at Process Pi at time Ci(**a**), we have a distributed implementation of the global time value; therefore, it is constructed a **logical clock**.

Logical Clocks

- 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**).
  - In other words, if C(**a**) < C(**b**), it does not necessarily imply that **a** **causally preceded** **b**.
  - Important: Precedence vs. Dependency
    - We say that **a** causally precedes **b**, if **a** $\rightarrow$ **b**
    - **b** may causally depend on **a**, as there may be information from **a** that is propagated into **b**.
  - To explain, consider the messages as sent by three processes shown on next slide.

Vector Clocks

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**.

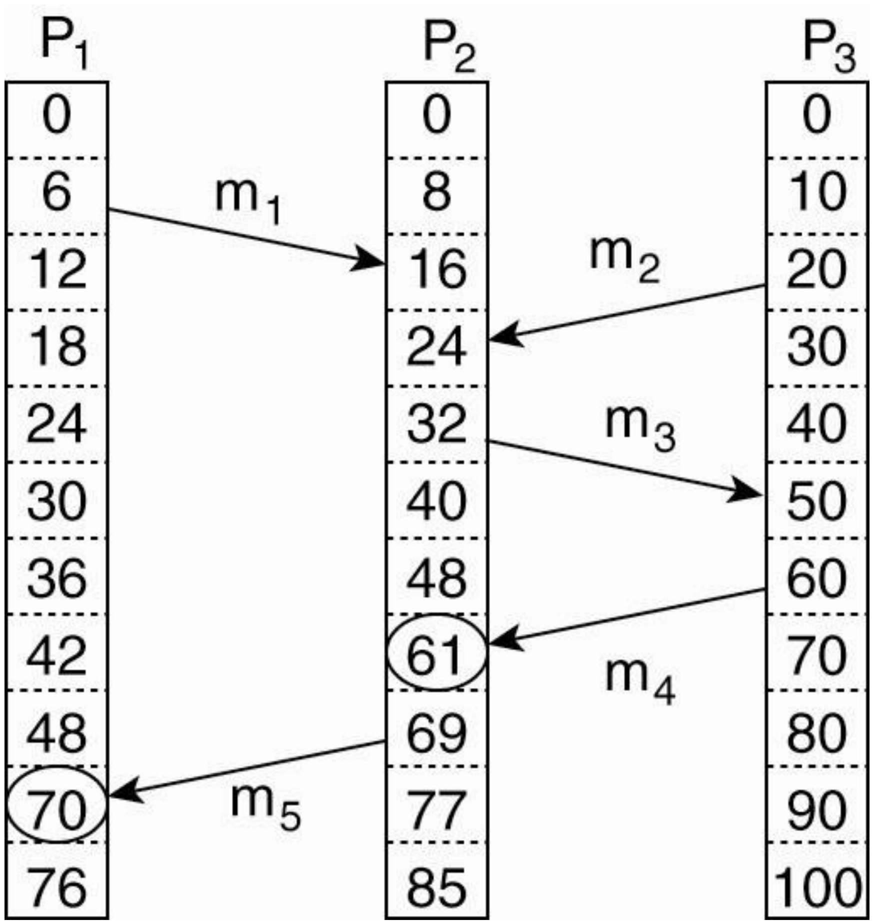
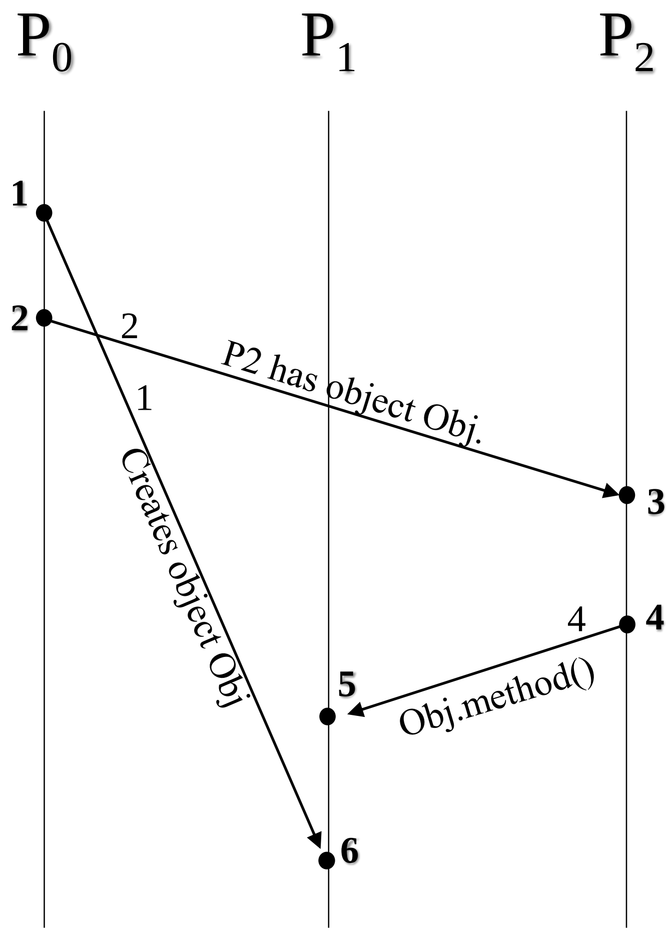


Figure 6-12. Concurrent message transmission using logical clocks.

Vector Clocks

- Message sent (#1) in P_0 is delayed, provoking a *causality violation* in P_1.
- Lamport's logical clock fails to detect this situation.



source: cseweb.ucsd.edu

Example of *causality violation*.

Vector Clocks

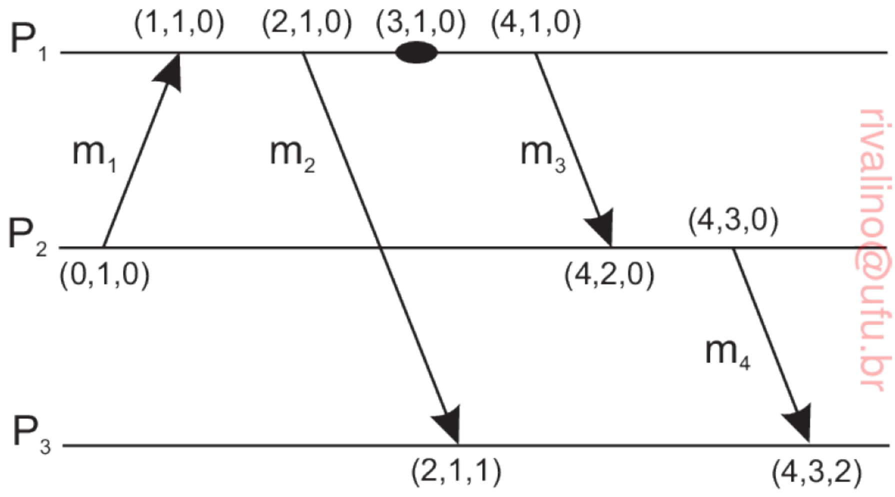
- Vector clocks are constructed by letting each process P_i maintain a vector VC_i with the following two properties:
    1. $VC_i [i]$ is the number of events that have occurred so far at P_i. In other words, $VC_i [i]$ is the local logical clock at process P_i .
    2. If $VC_i [j] = k$ then P_i knows that k events have occurred at P_j. It is thus P_i's knowledge of the local time at P_j .
- The 1st property is maintained by incrementing $VC_i[i]$ at the occurrence of each new event that happens at process P_i.
- The 2nd property is maintained by piggybacking vectors along with messages that are sent (see next slide).

Vector Clocks

- 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)* equals 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

1. P_2 sends a message m_1 at logical time $VC_2 = (0,1,0)$ to P_2.
2. Message m_1 thus receives timestamp $ts(m_1) = (0,1,0)$.
3. Upon its receipt, P_1 adjusts its logical time to $VC_1 \leftarrow (1,1,0)$ and delivers it.
4. Message m_2 is sent by P_1 to P_3 with $ts(m_2) = (2,1,0)$.
5. Before P_1 sends another message, m_3, an event happens at P_1, leading to timestamping m_3 with value (4,1,0).
6. After receiving m_3, process P_2 sends message m_4 to P_3, with timestamp $ts(m_4) = (4,3,0)$.

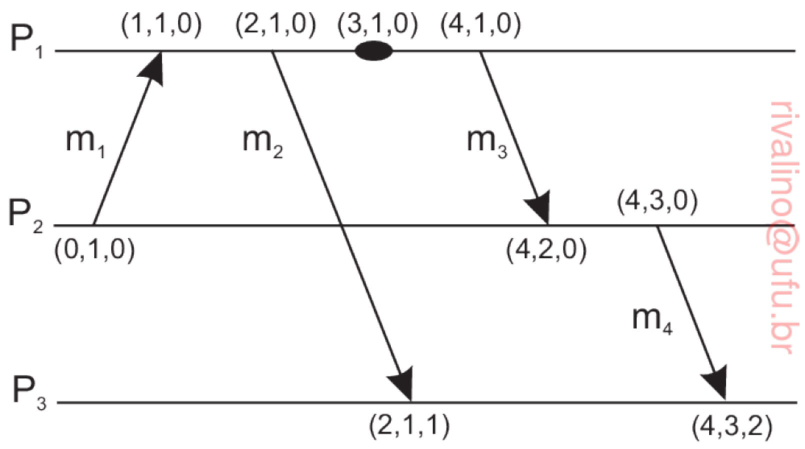


(a)

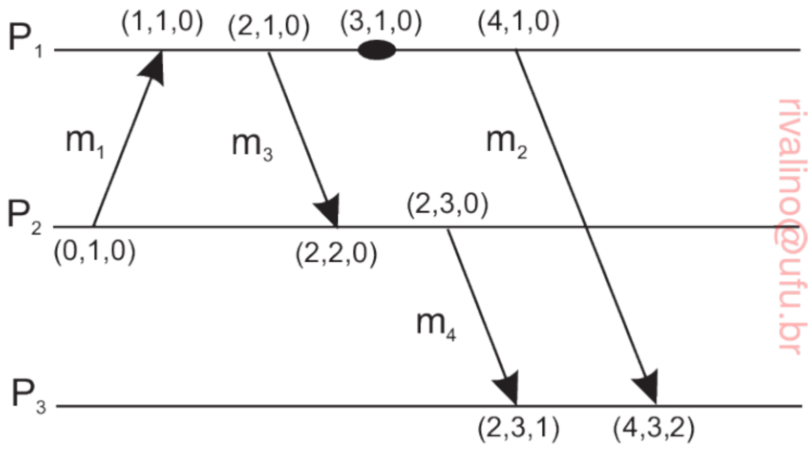
Vector Clocks

- Here, we have delayed sending message m_2 until message m_3 has been sent, after the event had taken place.
- One can see that ts(m_2) = (4, 1, 0), while ts(m_4) = (2, 3, 0).
- Based on the vectors, P_3 can identify, easily, a potential conflict.

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



(a)



(b)

Enforcing Causal Communication

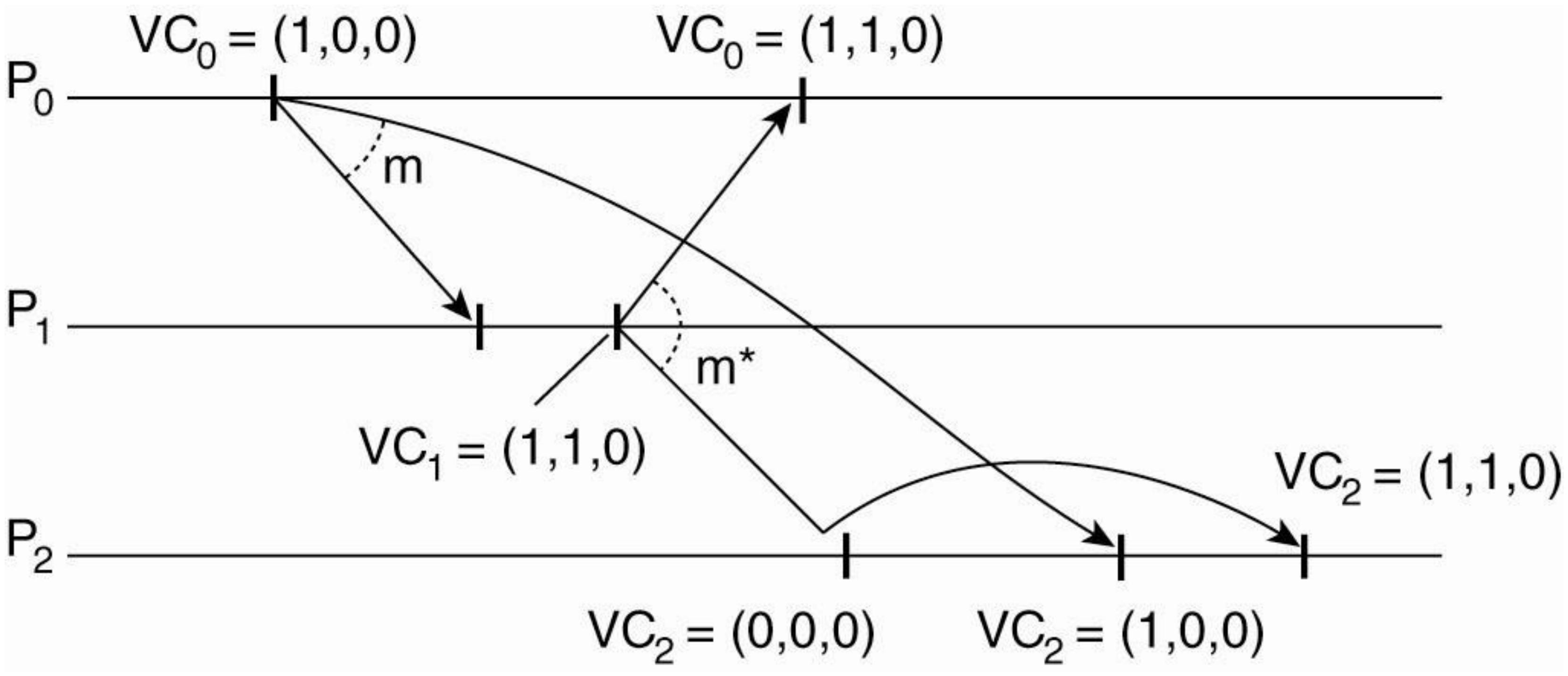


Figure 6-13. Enforcing causal communication.

Mutual Exclusion

- Fundamental to DS is the concurrency and collaboration among multiple processes.
- In many case, processes will need to concurrently access the same resources.
  - Not different than we studied in OS for non-distributed systems.
- To prevent concurrent accesses corrupting the resources, or make them inconsistent, distributed algorithms for mutual exclusion are needed.
  - **Permission-based solutions**: A process wanting to enter its critical section, or access a resource, needs permission from other processes.
  - **Token-based solutions**: A token is passed between processes. The one who has the token may proceed in its critical section or pass it on when not necessary.

Mutual Exclusion

- Permission-based, centralized
  - 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.
  - When the reply arrives, the requester can go ahead.

Mutual Exclusion A Centralized Algorithm (1)

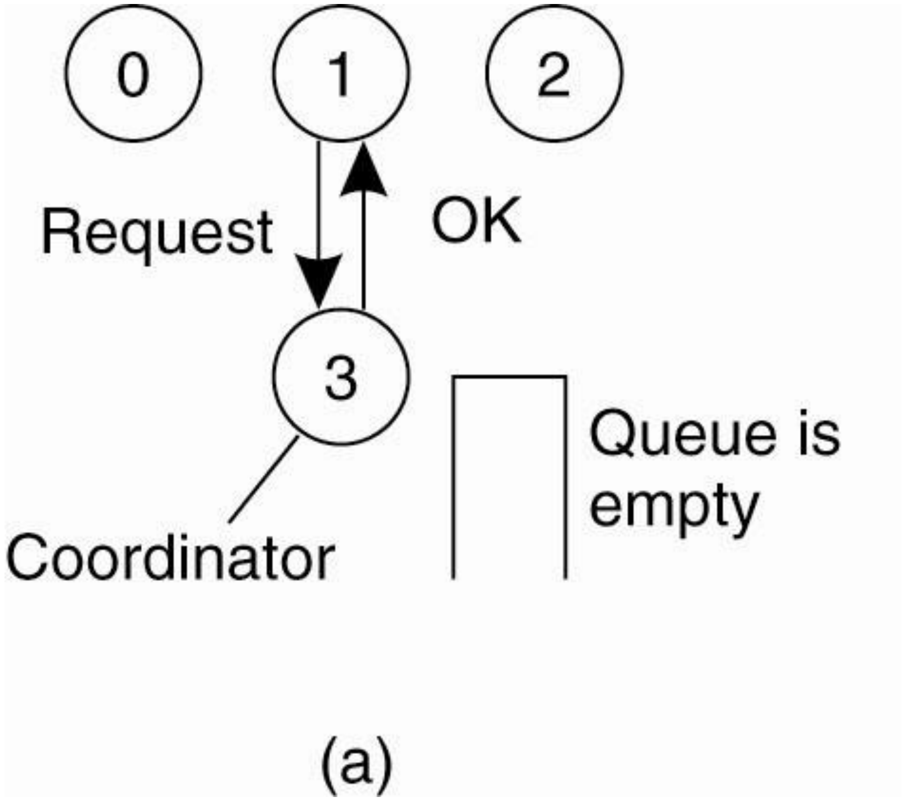


Figure 6-14. (a) Process 1 asks the coordinator for permission to access a shared resource. Permission is granted.

Mutual Exclusion
A Centralized Algorithm (2)

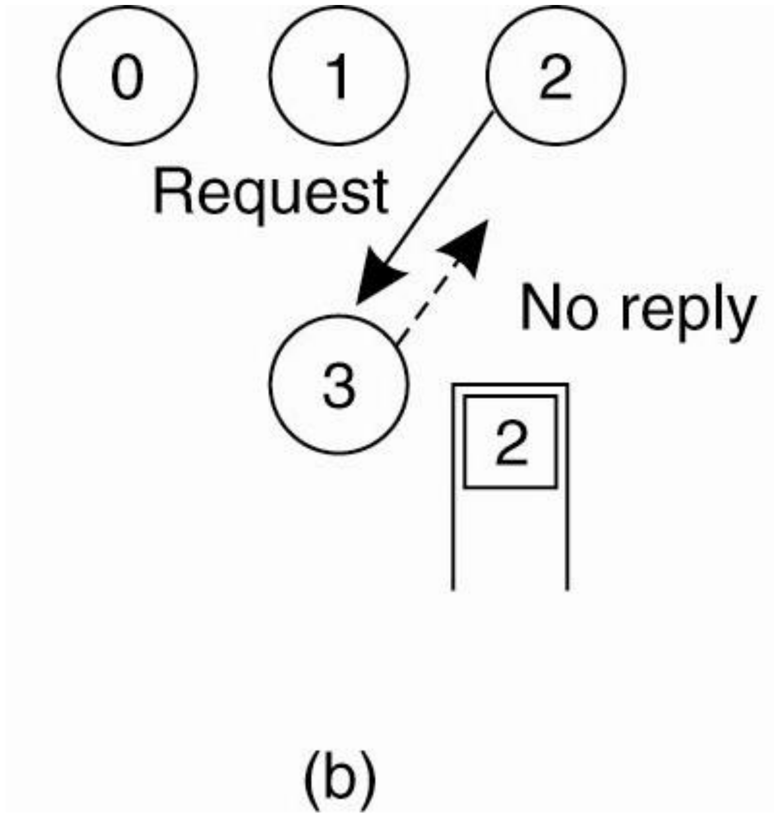


Figure 6-14. (b) Process 2 then asks permission to access the same resource. The coordinator does not reply.

Mutual Exclusion
A Centralized Algorithm (3)

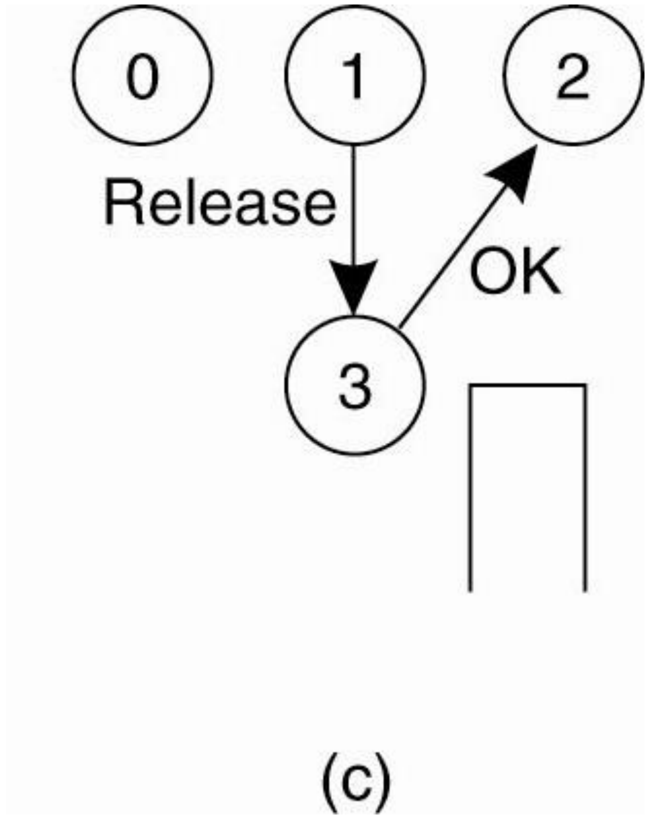


Figure 6-14. (c) When process 1 releases the resource, it tells the coordinator, which then replies to 2.

Mutual Exclusion

- Permission-based, centralized
  - It is easy to see that the algorithm guarantees mutual exclusion: the coordinator lets only one process at a time access the resource.
  - It is also fair, since requests are granted in the order in which they are received. No process ever waits forever (no starvation).
  - The scheme is easy to implement, too, and requires only three messages per use of resource (*request*, *grant*, *release*).
- This approach has shortcomings:
  - See next slide.

Mutual Exclusion

- Permission-based, shortcomings:
  - The coordinator is a single point of failure (SPOF), 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 a large system, a single coordinator can become a performance bottleneck.
  - Nevertheless, the benefits coming from its simplicity outweigh in many cases the potential drawbacks.
  - Moreover, distributed solutions are not necessarily better, as we illustrate next.

Mutual Exclusion

- Ricart & Agrawala algorithm, distributed:
  - Use Lamport's logical clock.
  - The solution requires a total ordering of all events in the system.
    - For any pair of events, such as messages, it must be unambiquous which one actually happened first.
- Algorithm overview: (1)
  - When a process wants to access a shared resource, it builds a message containing the name of the resource, its process ID, and the current (logical) time.
  - It sends the message to all other processes, including itself.
  - The communication is assumed to be reliable; 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 (see next slide).

Mutual Exclusion

- Ricart & Agrawala algorithm: Overview (2)
  - Three different cases must be clearly distinguished:
    - If the receiver is not accessing the resource and does not want to access it, it sends back an OK message to the sender.
    - If the receiver already has access to the resource, it simply does not reply. Instead, it queues the request.
    - 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.

Mutual Exclusion

- Ricart & Agrawala algorithm: Overview (3)
  - 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 the next slide.

A Distributed Algorithm

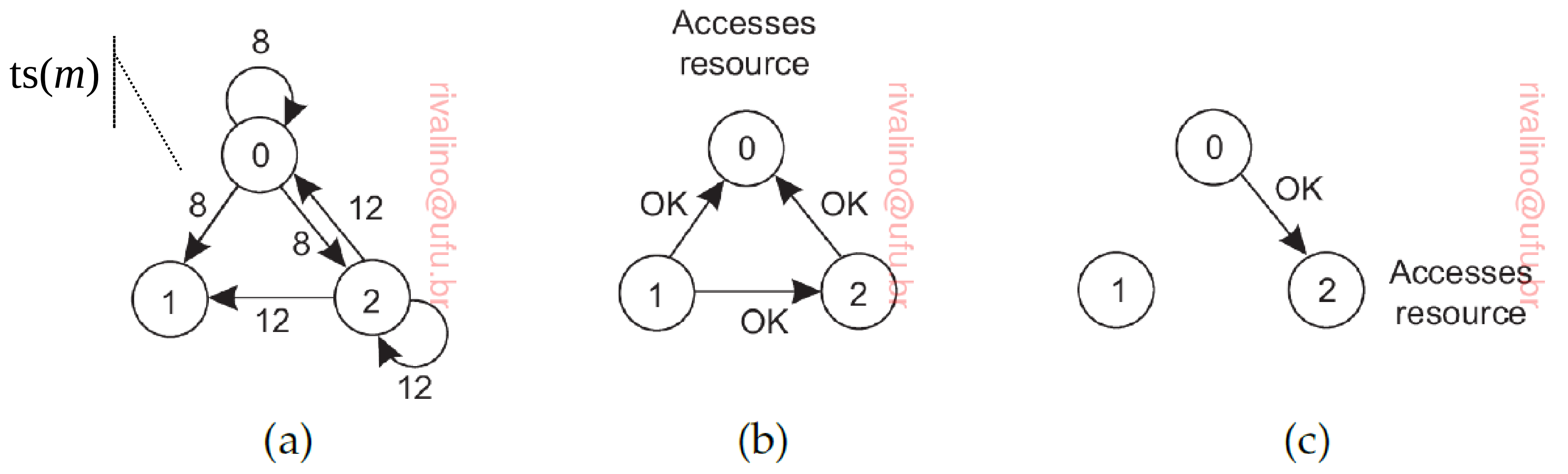


Figure 6.16: (a) Two processes want to access a shared resource at the same moment. (b) P_0 has the lowest timestamp, so it wins. (c) When process P_0 is done, it sends an OK also, so P_2 can now go ahead.

Mutual Exclusion

- Ricart & Agrawala algorithm: Overview (4)
  - With this algorithm, 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(*N*-1), i.e., *N*-1 requests and subsequently *N*-1 OK messages, one from each other process.
- This approach has shortcomings:
  - It has *N* points of failures (SPOF). If any process crashes, it will fail respond requests.
    - This silence will be interpreted (incorrectly) as denial of permission, blocking all processes to enter any of their respective critical regions.
    - Timeout can be used to mitigate this problem.
  - Works better for small groups of processes.

Mutual Exclusion

- Token-ring algorithm
  - Approach to deterministically achieving mutual exclusion in a DS.
  - An overlay network in the for of a ring is created.
  - Each process is assigned a position in the ring.
  - The **token** circulates around the ring, and the process that takes the token has exclusive access to a shared resource. After it finishes, it releases the token.
  - If the process is handed the token by its neighbor and is not interested in the resource, it just passes the token along.
- This approach has shortcomings:
  - If the token is lost (e.g., node crash), it must be regenerated.
  - It is difficult detecting a lost token. How long it takes to detect a missed token?

A Token Ring Algorithm

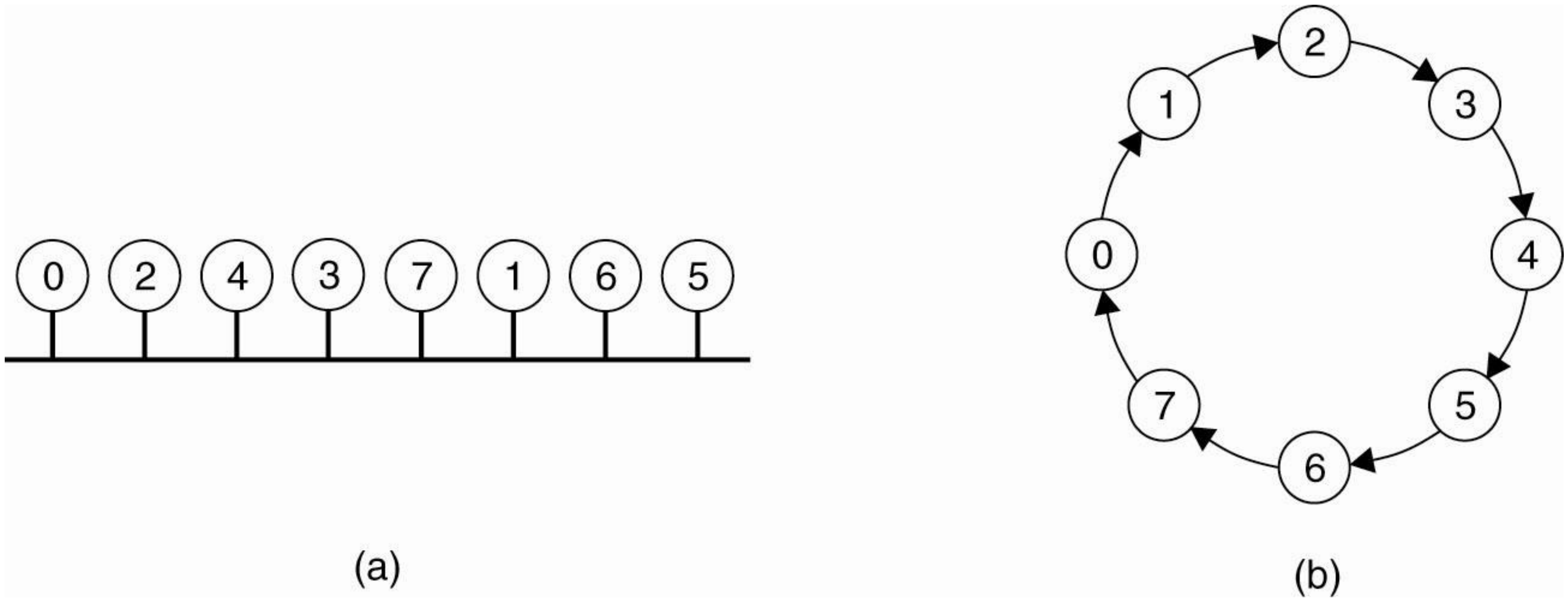


Figure 6-16. (a) An unordered group of processes on a network. (b) A logical ring constructed in software.

Mutual Exclusion

- Fully decentralized mutual exclusion
  - Approach proposed by Lin et al. [2004], use a voting algorithm.
  - Each shared resource is assumed to be replicated *N* times.
  - Every replica has its own mutual exclusion coordinator.
  - Whenever a process wants to access the resource, it will simply need to get a majority vote from $m > N/2$ coordinators.
  - Assume that:
    - When a coordinator does not give permission to access a resource, it will tell the requester.
    - When a coordinator crashes, it recovers quickly but will have forgotten any vote it gave before it crashed.
- This approach has shortcomings:
  - If *f* coordinators reset, the correctness of the voting will be violated when there is more than *m* other coordinators think it is okay to allocate the resource.

A Comparison of the Four Algorithms

Algorithm	Messages per entry/exit	Delay before entry (in message times)
Centralized	3	2
Distributed	$2 \cdot (N-1)$	$2 \cdot (N-1)$
Token ring	$1, \ldots, \infty$	$0, \ldots, N-1$
Decentralized	$2 \cdot m \cdot k + m, k = 1, 2, \ldots$	$2 \cdot m \cdot k$

Figure 6.19: A comparison of four mutual exclusion algorithms.

Election Algorithms

- Many distributed algorithms require one process to act as coordinator, initiator, or otherwise perform some special role.
    - For example, mutual exclusion algorithms seen before.
- In many systems the coordinator is chosen by hand (e.g., file servers).
    - This leads to centralized solutions and thus with SPOF.
- In general, it does not matter which process takes on this special responsibility, but one of them must do it.
- Next, we look at algorithms for electing a coordinator.

Election Algorithms

- Overview:
  - 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.
  - The algorithms differ in the way they locate the coordinator.
  - Also, they assume that every process has complete knowledge of the process group in which the coordinator must be selected.
  - What the processes do not know is which ones are currently up and which ones are currently down.
  - The goal of an election algorithm is to ensure that when an election starts, it concludes with all processes agreeing on who will be the next coordinator.

Election Algorithms

- The Bully Algorithm
  - Consider N processes $\{P_0, \ldots, P_{N-1}\}$ 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:

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

The Bully Algorithm (1)

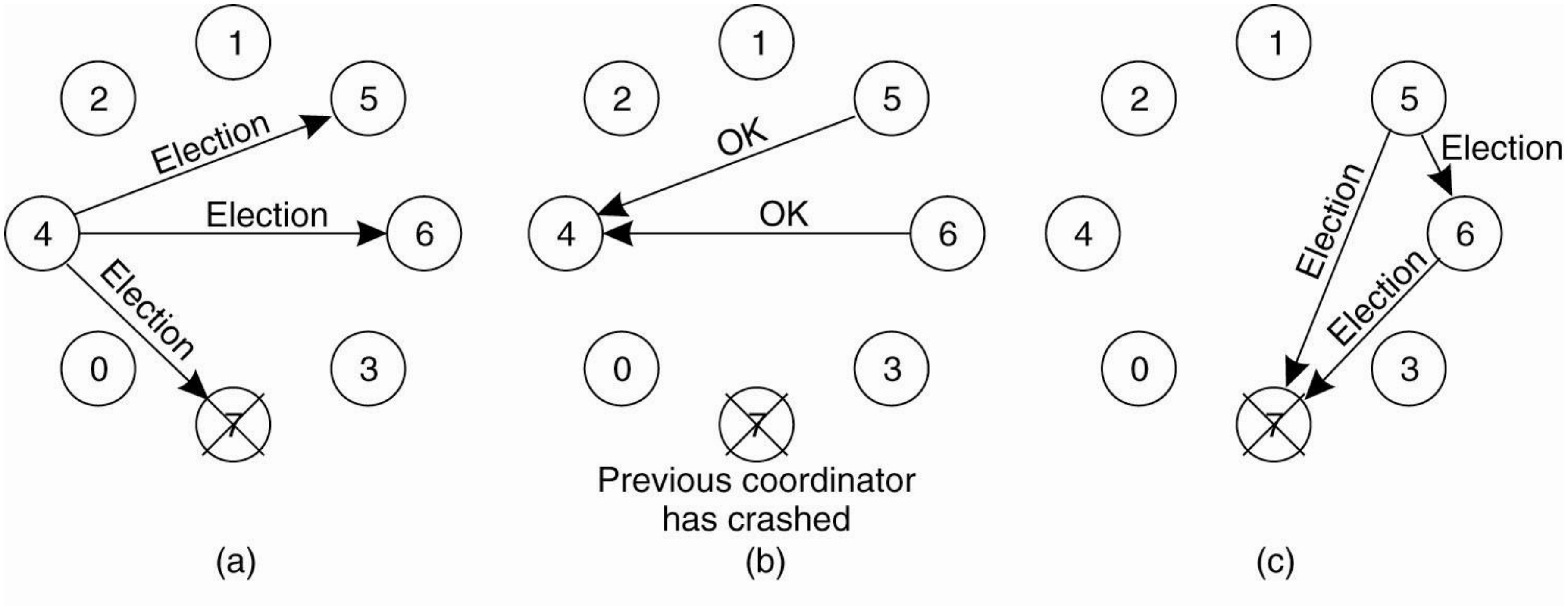


Figure 6-20. The bully election algorithm. (a) Process 4 holds an election. (b) Processes 5 and 6 respond, telling 4 to stop. (c) Now 5 and 6 each hold an election.

The Bully Algorithm (2)

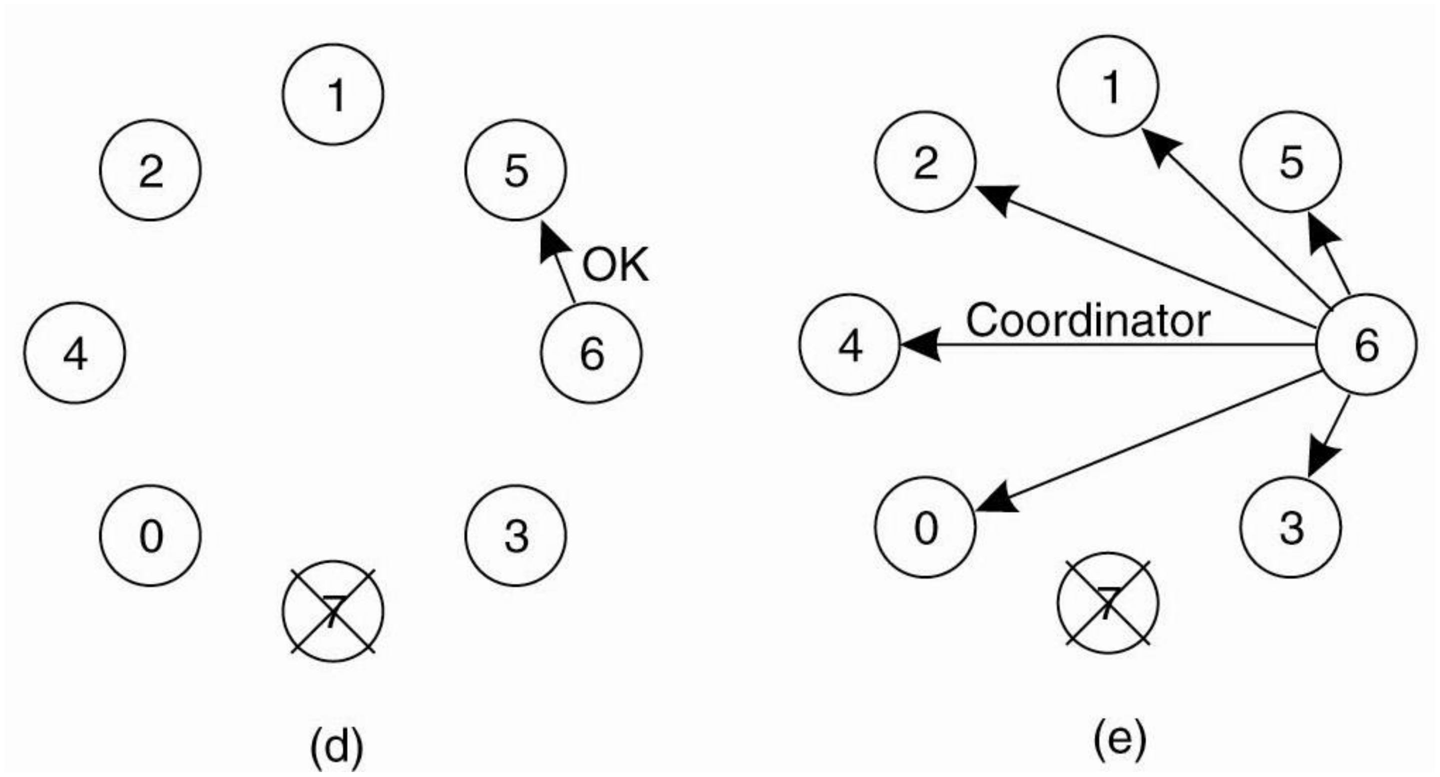


Figure 6-20. The bully election algorithm. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone.

Election Algorithms

- The Ring Algorithm
  - Based on a logical ring (overlay network).
  - 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 running member along the ring.
  - At each step, 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 containing 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 (the list member with the highest identifier) and who the members of the new ring are.
  - When this message has circulated once, it is removed and everyone goes back to work.

A Ring Algorithm

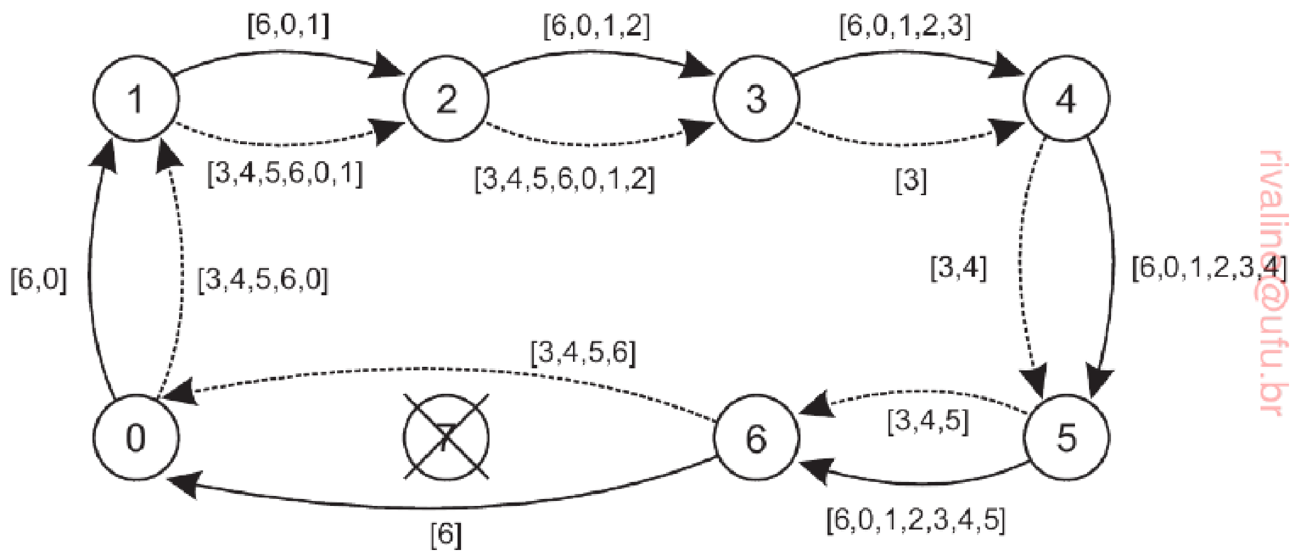


Figure 6.21: Election algorithm using a ring. The solid line shows the election messages initiated by P_6; the dashed one those by P_3.