



# Systemy rozproszone

Informatyka, sem. 6  
część II



# Synchronization

na podstawie:

A.S. Tanenbaum, M. van Steen

Distributed Systems Principles and Paradigms

z modyfikacjami P. Kaczmarek



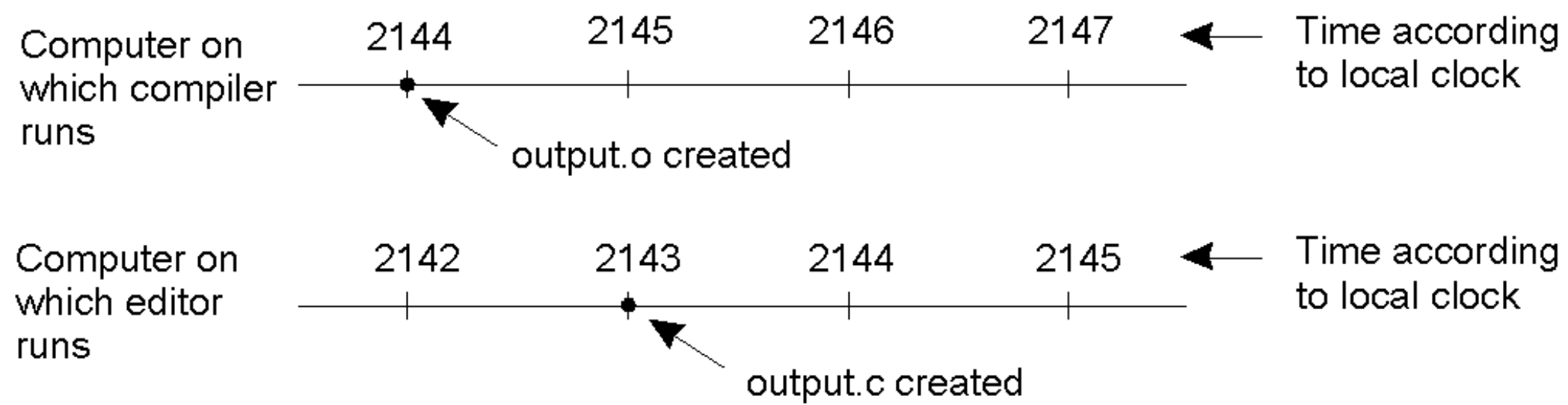
# Synchronization in distributed systems



- The need for synchronization
  - Communication
  - Ordering of events
- Centralized vs distributed clocks
- Group communication and elections



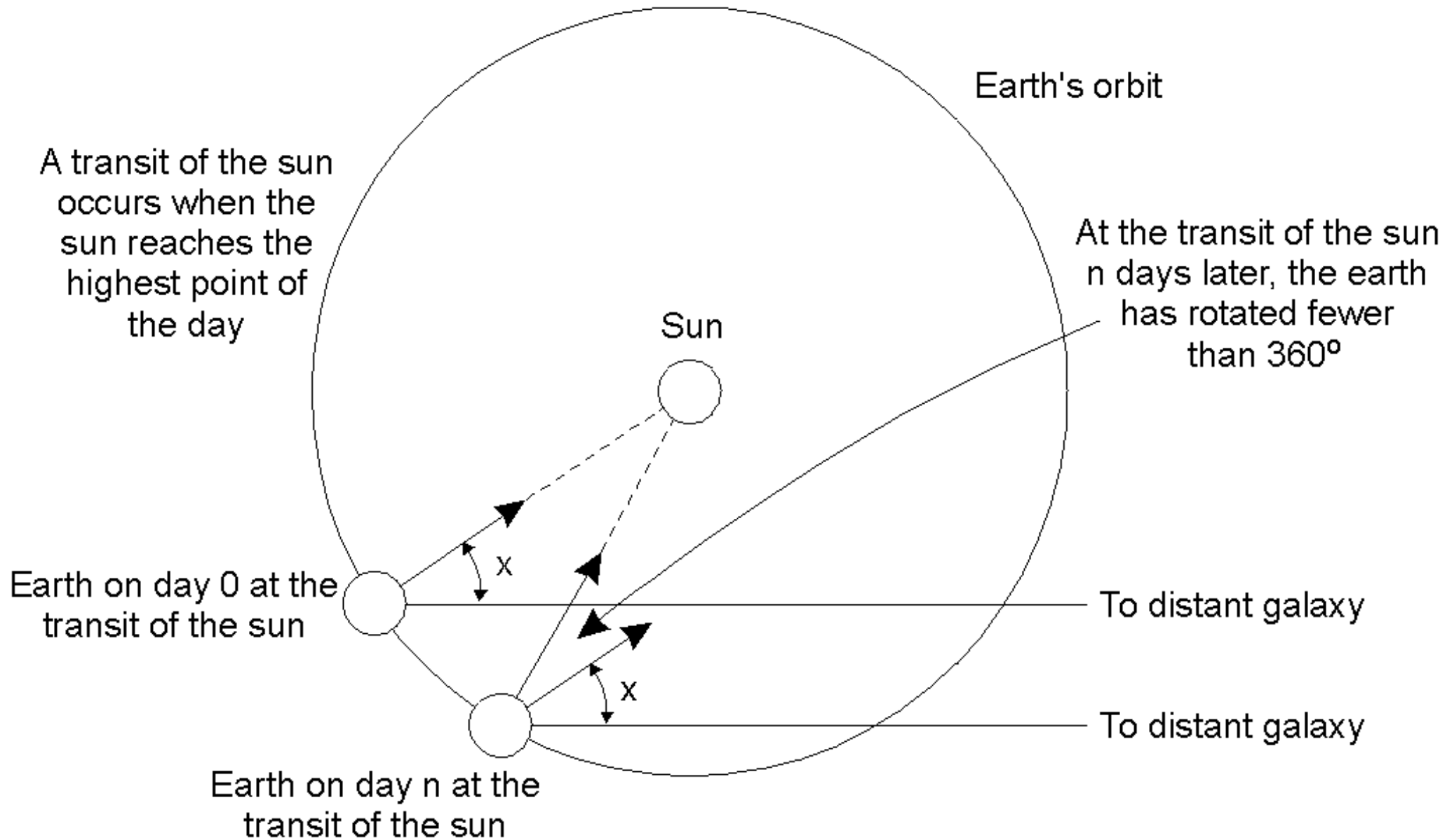
# Clock Synchronization



- Example: edit and compile code on different machines
- When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.



# Physical Clocks (1)



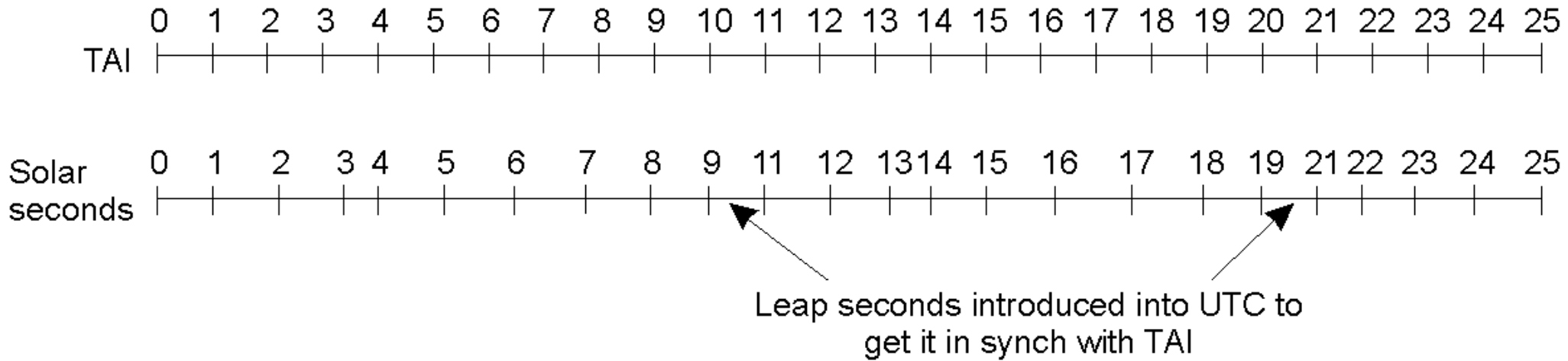


# Physical Clocks (2)

- Computation of the mean solar day.
- Physical clocks
  - timer (quartz crystal) > counter > holding register > clock tick
  - solar day, solar second (1/86400 day), mean solar second
- International Atomic Time (TAI)
  - ticks of cesium 133
  - leap seconds - solar days become longer
- Universal Coordinated Time



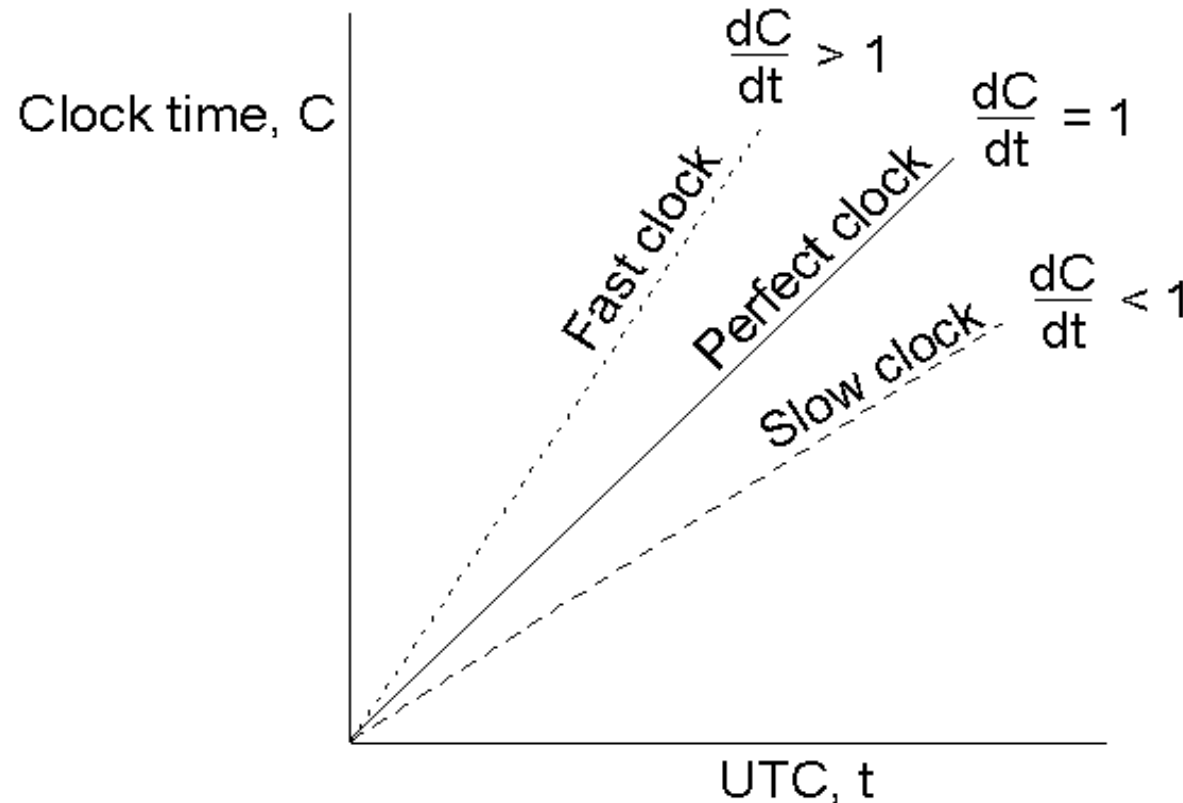
# Physical Clocks (3)



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



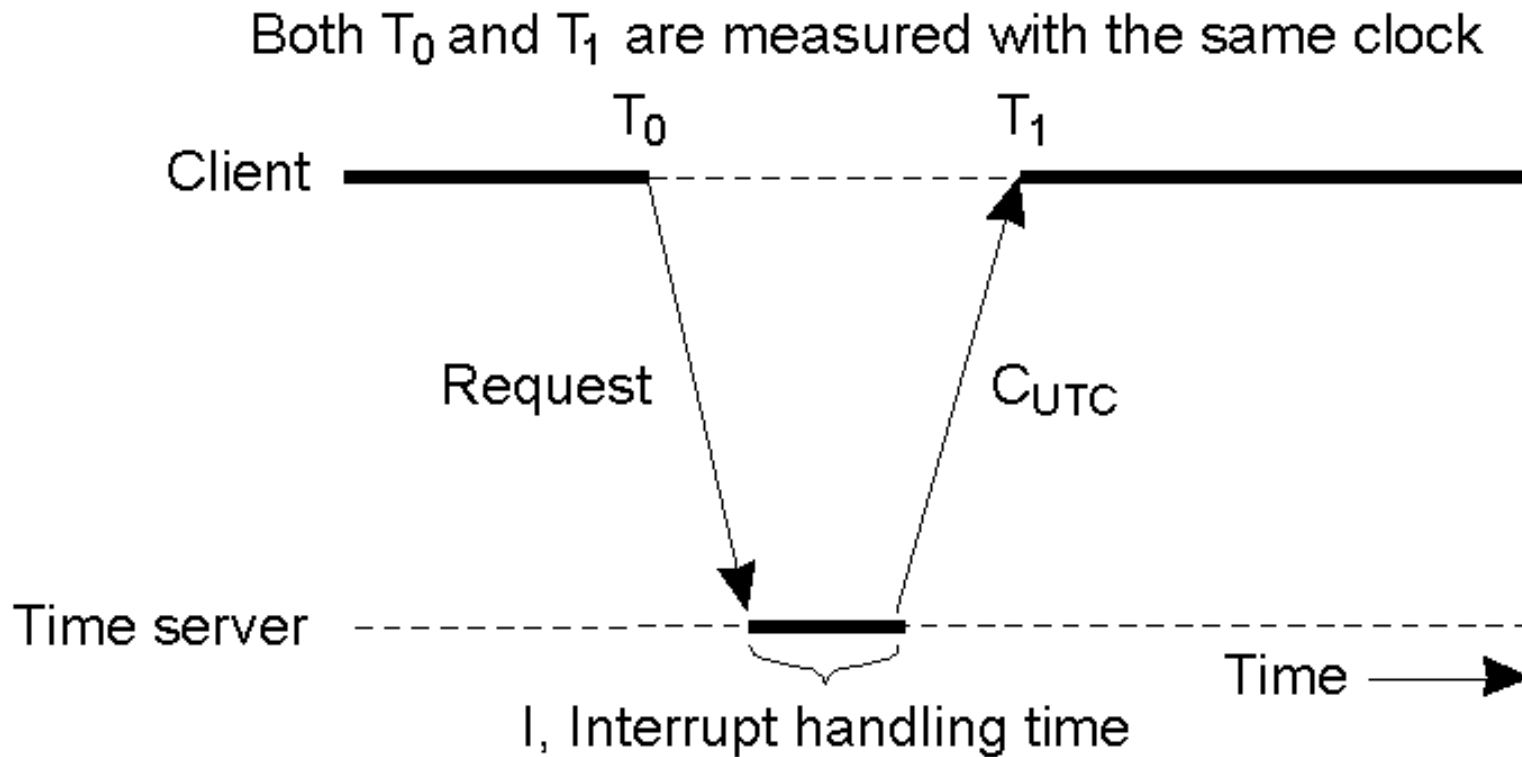
# Clock Synchronization Algorithms



- The relation between clock time and UTC when clocks tick at different rates. (maximum drift rate)



# Cristian's Algorithm (1)



- A time server exists and has a WWV receiver

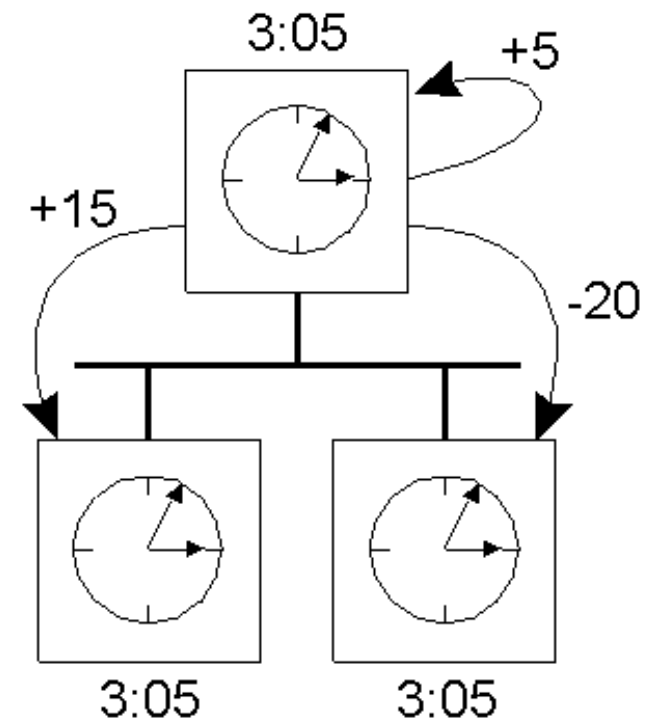
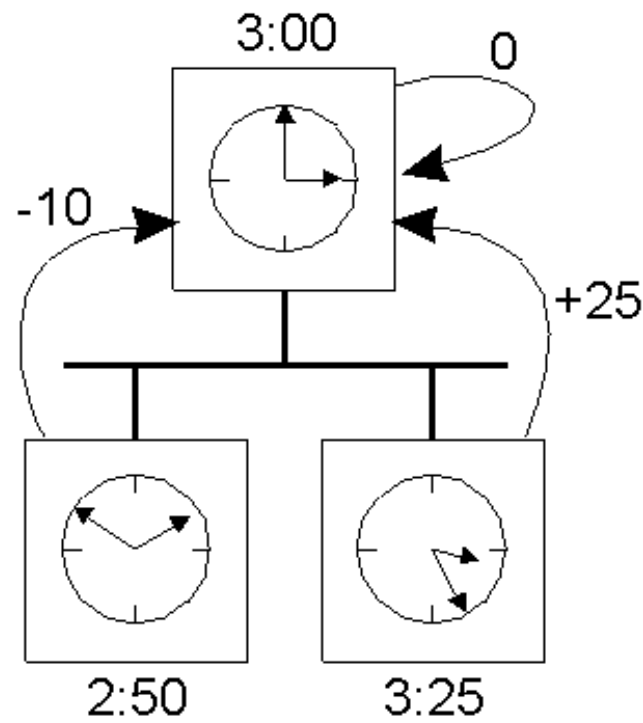
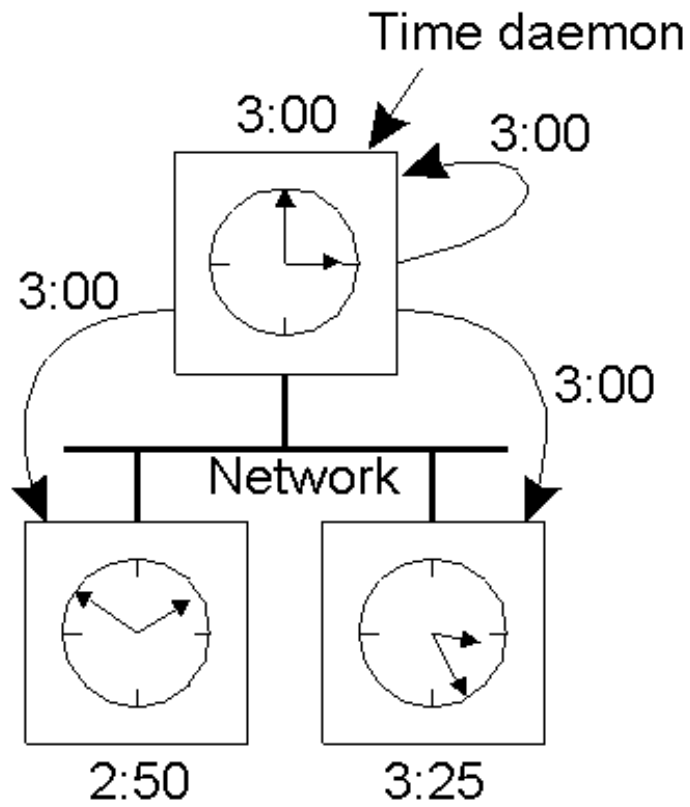


# Cristian's Algorithm (2)

- Each client machine sends a message to the time server asking for the current time
  - The server responds
- Two problems (and solutions)
  - server time might be lower, time must never run backward
    - change must be introduced gradually
  - communication takes a nonzero amount of time
    - a simple estimate  $(T1 - T0) / 2$  or
    - improve by calculating interrupt handling time  $I$   
message propagation time is calculated as  $T1 - T0 - I$



# The Berkeley Algorithm (1)





# The Berkeley Algorithm (2)



- The time daemon asks all the other machines for their clock values (server is active)
- The machines answer
- The time daemon tells everyone how to adjust their clock



# Decentralized Algorithms

- Averaging algorithm
  - Divide time into fixed-length resynchronization intervals
    - $T_0+iR$ ,  $T_0+(i+1)R$ ,  $T_0$  - a moment in past,  $R$  - system parameter
  - Every machine broadcasts its time
  - Every machine collects all other broadcasts
    - when all broadcasts arrive, compute a new time
    - take average, discard  $m$  highest and  $m$  lowest values
    - estimate propagation time (from network topology)



# Logical clocks (1)

- Physical time is not always necessary
- Happens-before relationship (Lamport)
  - $a, b$  are events in the same process,  $a$  occurs before  $b$ ,  $a \rightarrow b$ ,  $C(a) < C(b)$  ( $C == \text{Clock}$ )
  - $a$  - message sending,  $b$  - message receiving,  $a \rightarrow b$   
 $C(a) < C(b)$
- Concurrent events
  - neither  $x \rightarrow y$  nor  $y \rightarrow x$ ,  $C(x) \neq C(y)$
  - the clock time must always go forward



# Logical clocks (2)

- Messages are assigned a timestamp
  - receiver fast forwards its clock if it is prior to message send time
- For example
  - three processes, each with its own clock. The clocks run at different rates.
  - Lamport's algorithm corrects the clocks.



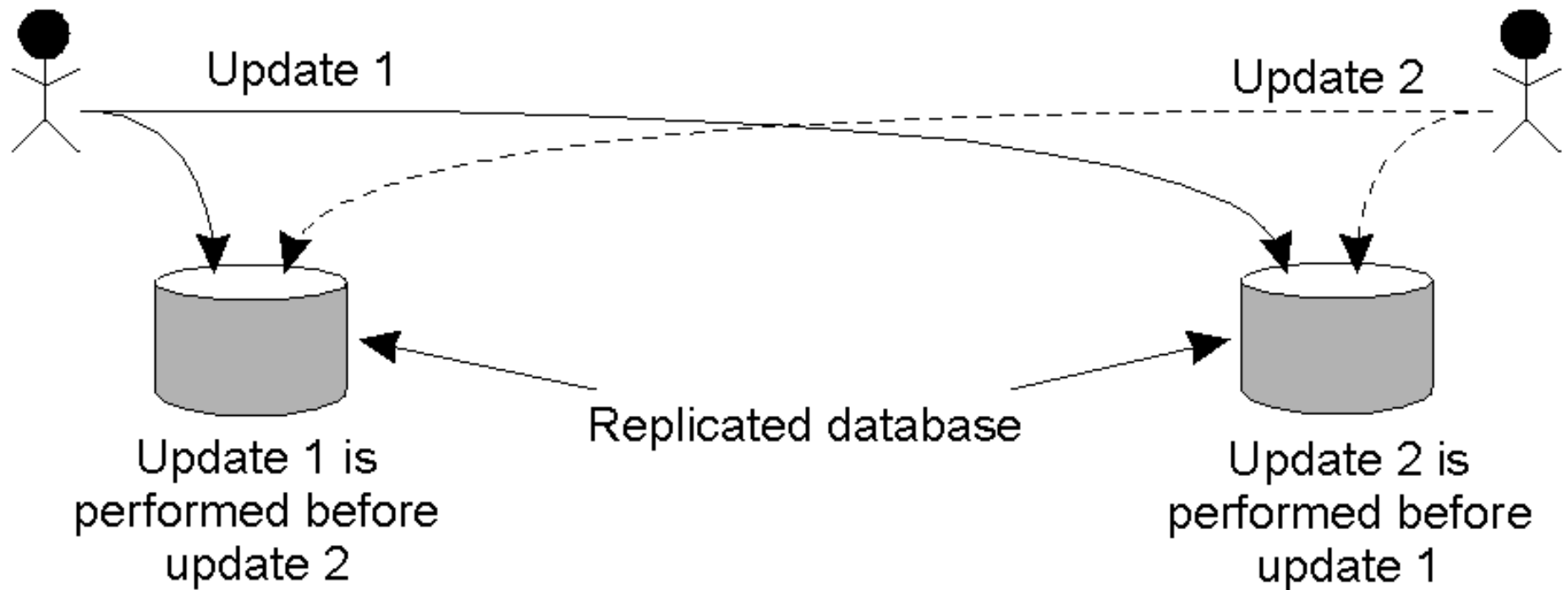
# Logical clocks (3)



1	1	1
2	3	4
3	5	7
4	7	10
5	9	13
6	11 / 14	16
7	16	19
8 / 17	18	22
18	20	25



# Example: Totally-Ordered Multicasting





# Totally-Ordered Multicasting (2)

- Updating a replicated database and leaving it in an inconsistent state.
  - (for example: bank update -100\$, +1% interest rates)
- Two updates should be performed in the same order at each copy
- Totally ordered multicast
  - all messages are delivered in the same order to each receiver
  - use Lamport timestamps - distributed fashion



# Totally-Ordered Multicasting (3)

- Assumptions:
- Each message is timestamped with the current (logical) time of its sender
- A message is conceptually also send to the sender
- Messages from the same sender are received in the order they were send
- No message is lost



# Totally-Ordered Multicasting (4)

- When a process receives a message it is put into a local queue, ordered according to its timestamp
- The receiver multicasts an acknowledgment
  - acknowledgment timestamp is higher than message
- Eventually, all processes will have the same copy of the local queue
  - each message is multicasted (including acknowledgments)
  - processes put messages in local queues according to timestamps (Lamport timestamps ensure consistent global ordering of events)
- A process delivers a queued message to the application if
  - the message is at the head of the queue and
  - it has been acknowledged by each other process



# Vector timestamps (1)

- Lamport timestamps are insufficient in some cases
  - $C(a) < C(b)$  does not necessarily imply that  $a$  happened before  $b$
  - for example network news: delivering message  $B$  after message  $A$  does not imply that  $B$  is a reaction to  $A$
  - causality (przyczynowość)
  - the receipt of an article always causally precedes a reaction



# Vector timestamps (2)

- A vector timestamp  $VT(a)$  assigned to an event  $a$  has the property that if  $VT(a) < VT(b)$  for some event  $b$ , then  $a$  is known to causally precede event  $b$ 
  - each process  $P_i$  maintains a vector  $V_i$ 
    - $V_i[j]$  - the number of events that have occurred so far at  $P_i$
    - if  $V_i[j]=k$  then  $P_i$  knows that  $k$  events have occurred at  $P_j$
  - the first property is maintained by incrementing  $V_i[i]$  at the occurrence of each new event at  $P_i$
  - the second property is maintained by piggy-back vectors along with messages, timestamp  $vt$  is send with messages



# Vector timestamps (3)

- A receiver receives  $v_t$ 
  - it is informed about the number of events that have occurred at  $P_i$
  - informed about how many events at other processes have taken place before  $P_i$  sent message  $m$
  - potentially causally dependent
- When  $P_j$  receives  $m$ , it adjusts its vector
  - $V_j[k] = \max \{V_j[k], v_t[k]\}$

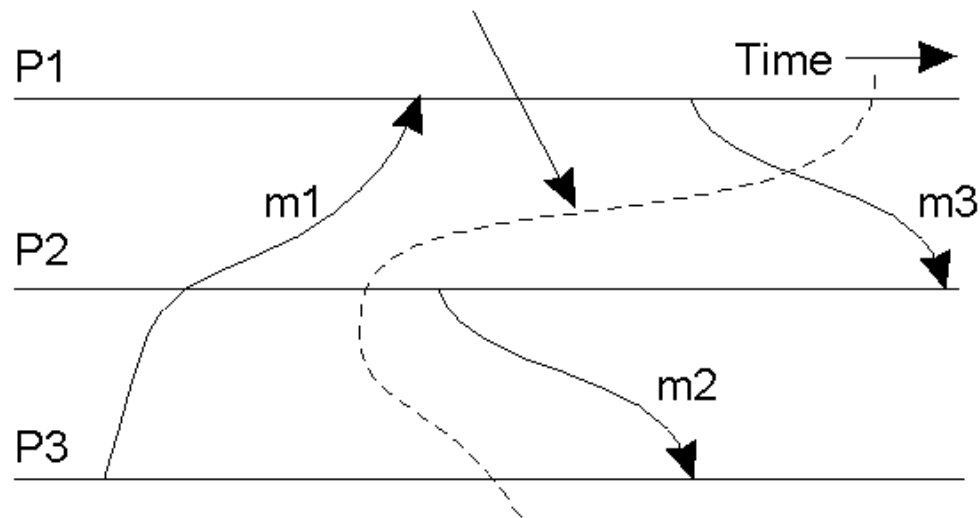


# Global state

- Global state consists of the local state of each process and the messages that are currently in transit
  - detect deadlock or end of computation
- Distributed snapshot
  - reflects a state in which the distributed system might have been
  - reflects a consistent global state
    - if we have recorded that P has received a message from Q, we should have recorded that Q had sent it
    - the reverse condition is allowed (Q has sent, P has not received)

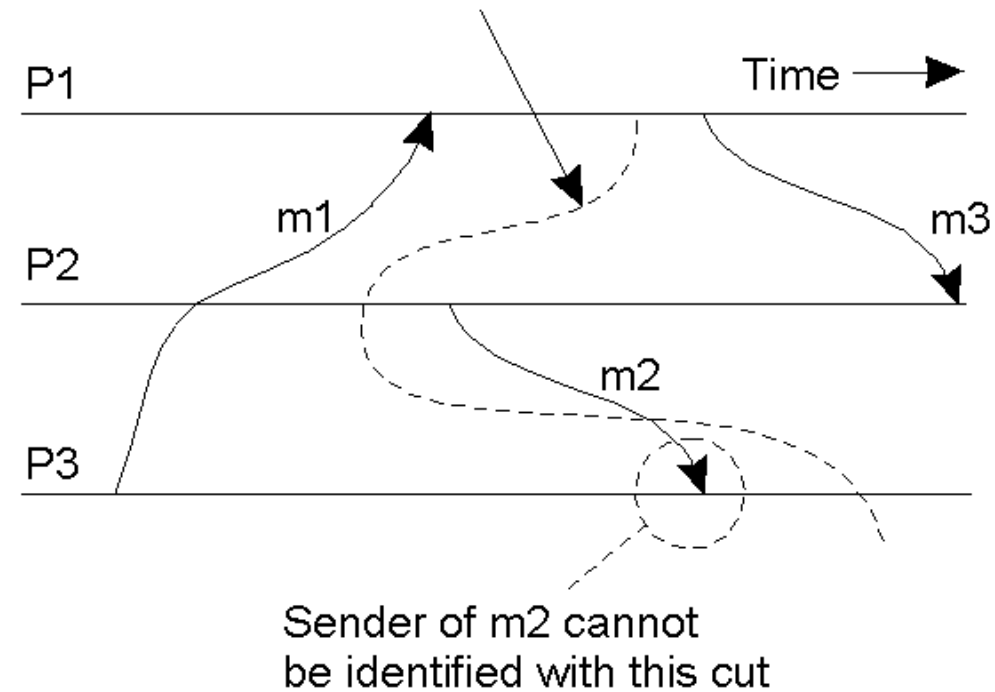
# Global State (2)

Consistent cut



(a)

Inconsistent cut

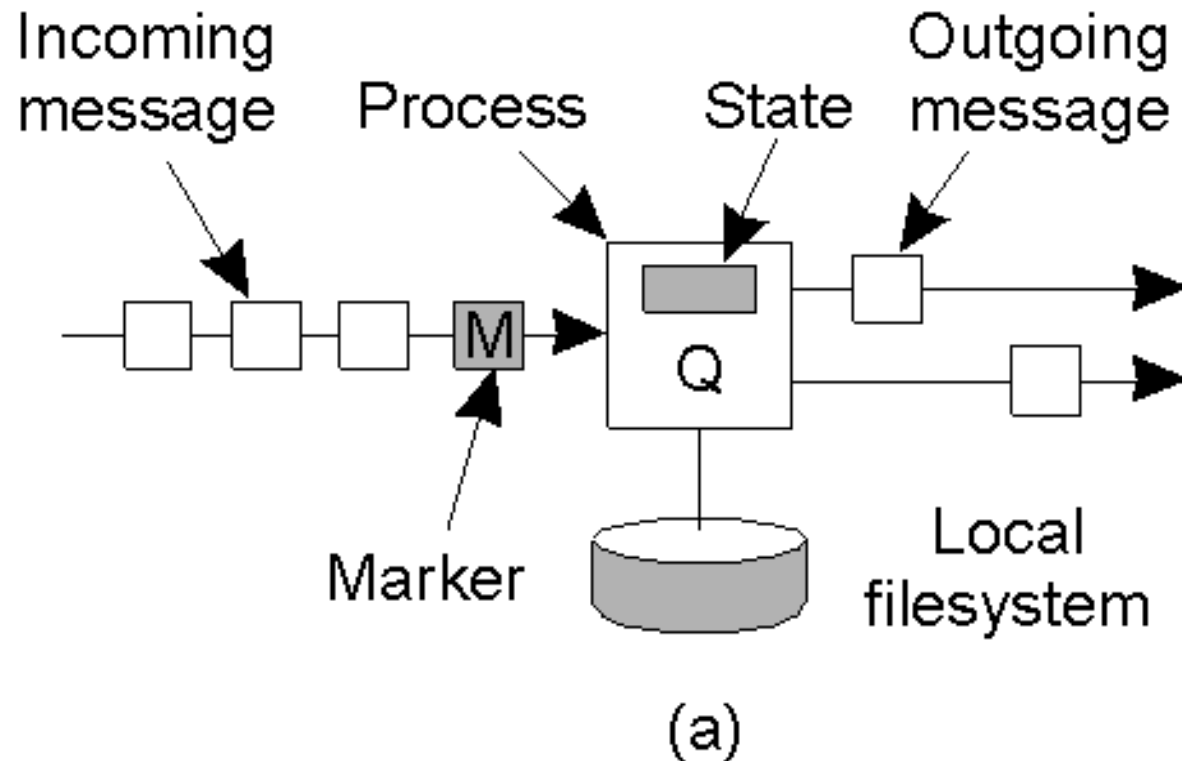


(b)

- (a) A consistent cut
- (b) An inconsistent cut



# Distributed snapshot



- Organization of a process and channels for a distributed snapshot, assumption:
  - a distributed system - a collection of processes connected by unidirectional point-to-point communication channels

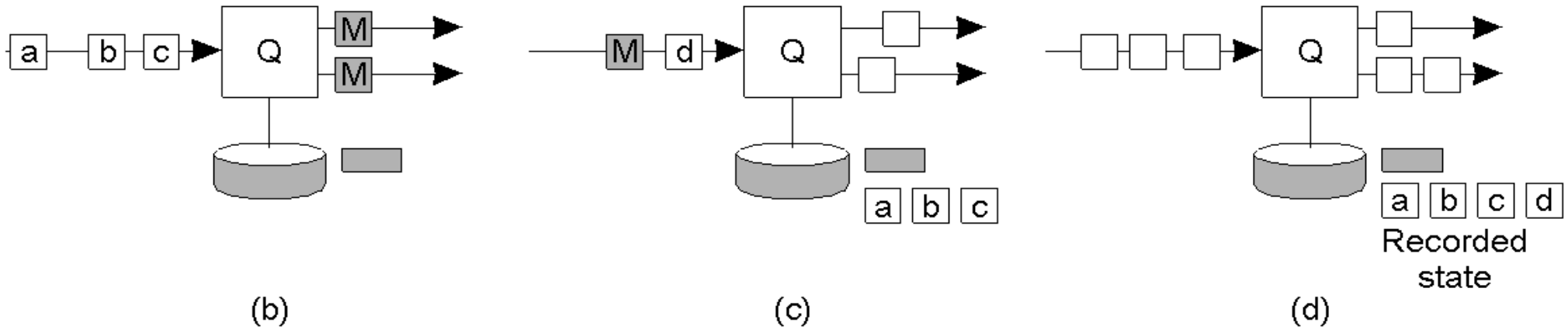


# Distributed snapshot (2)

- Any process can initialize (P)
- P records its state and sends a marker along each of its outgoing channels
- When Q receives a marker
  - if Q had not recorded its state, Q records its state and sends a marker along each of its outgoing channels
  - if Q had recorded its state, the marker on channel C indicates Q should record the state of the channel
  - channel state - the sequence of messages
- A process has finished its part when it has received a marker along each of its incoming channels
- The state can be sent, e.g., to the process that initialized the snapshot (different snapshots might be taken simultaneously)



# Distributed snapshot (3)



- Process Q receives a marker for the first time and records its local state
- Q records all incoming message
- Q receives a marker for its incoming channel and finishes recording the state of the incoming channel



# Election algorithms

- Many distributed algorithms require one process to act as coordinator
- Assumptions
  - processes are the same, each process has a unique number
  - every process knows the process number of every other process
  - processes do not know which ones are currently up
- In general, locate the process with the highest number



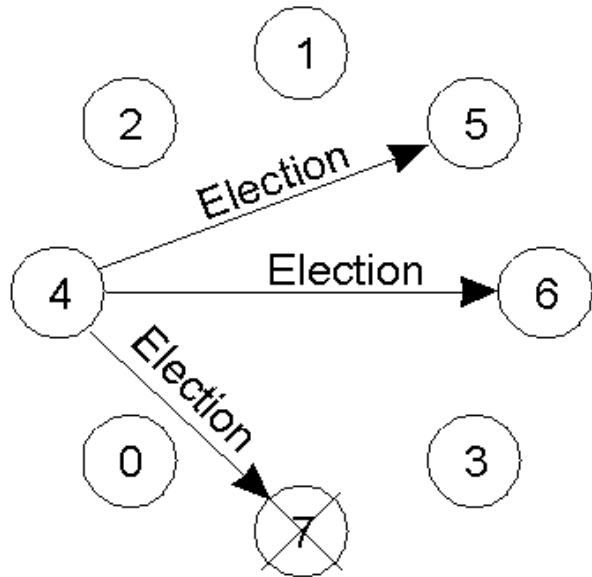
# Election

## The Bully Algorithm (1)

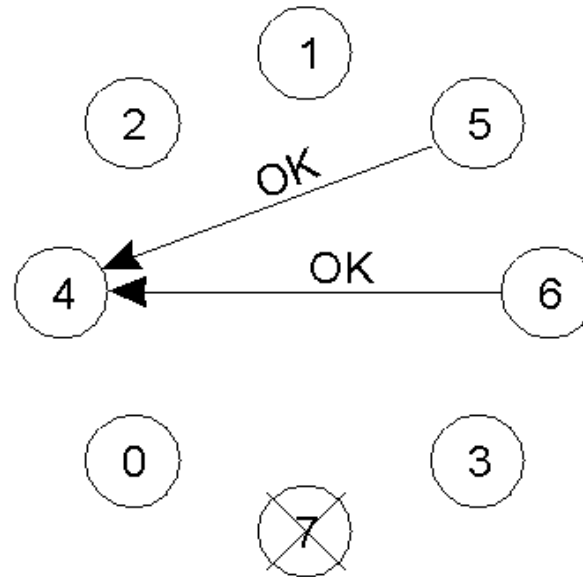
- A process,  $P$ , holds an election as follows:
  - $P$  sends an ELECTION message to all processes with higher number
  - If no one responds,  $P$  wins and becomes coordinator
  - If one of the higher-ups answers, it takes over,  $P$ 's job is done



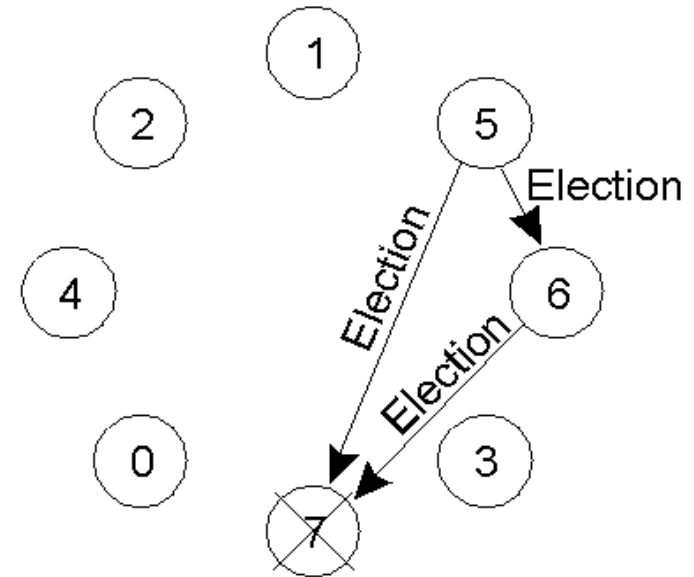
# The Bully Algorithm (2)



(a)



(b)

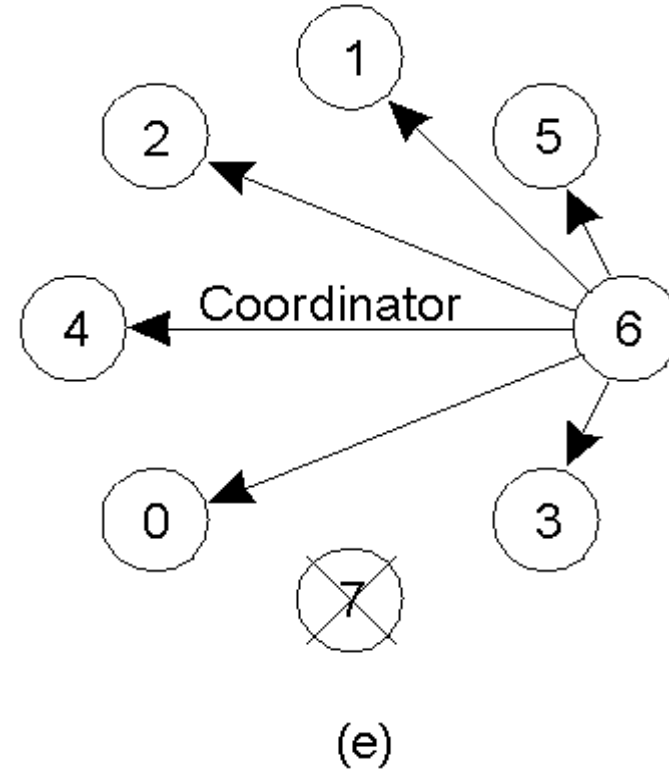
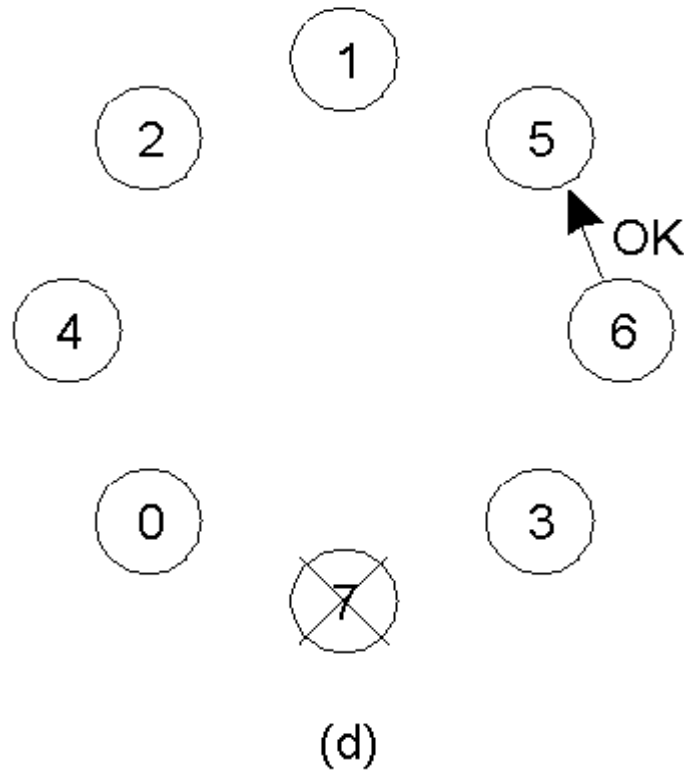


(c)

- The bully election algorithm
  - Process 4 holds an election
  - Process 5 and 6 respond, telling 4 to stop
  - Now 5 and 6 each hold an election



# The Bully Algorithm (3)



- Process 6 tells 5 to stop
- Process 6 wins and tells everyone



# A Ring Algorithm

- Processes are ordered
- Process sends ELECTION message to its successor
  - if the successor is down, the sender skips over it
- At each step, the sender adds its own process number - become a candidate
- Eventually, the message gets back to the process that started it
- The process calculates and sends COORDINATOR (circulated again)



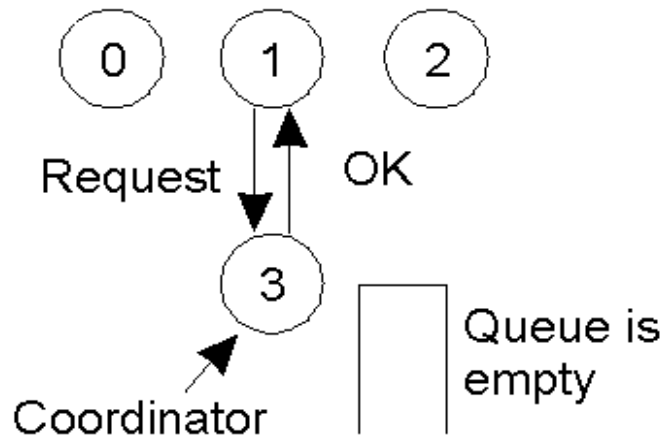


# Mutual Exclusion

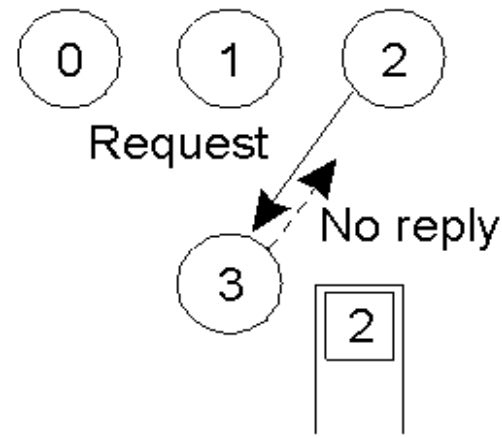
- Sequential systems (semaphores, monitors)
- Distributed systems
  - algorithms: centralized, distributed, token ring



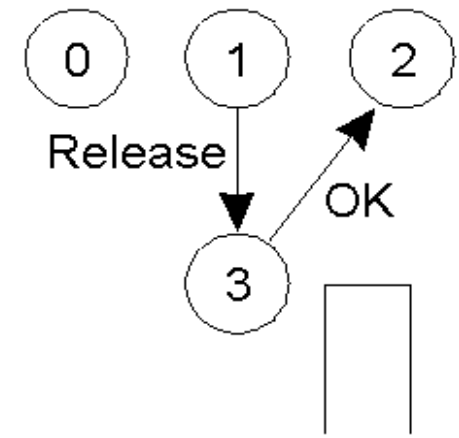
# A Centralized Algorithm



(a)



(b)



(c)

- Process 1 asks the coordinator for permission to enter a critical region. Permission is granted
- Process 2 then asks permission to enter the same critical region. The coordinator does not reply (or denies permission)
- When process 1 exits the critical region, it tells the coordinator, when then replies to 2

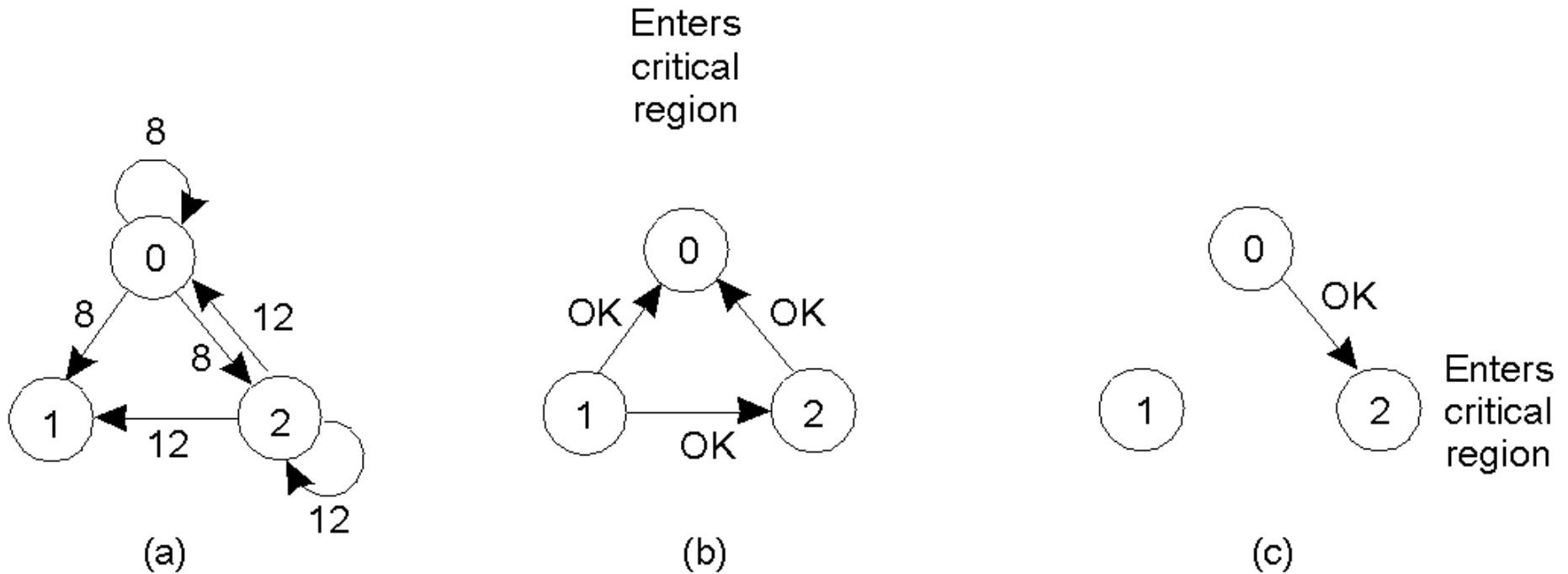


# A Distributed Algorithm

- A process wants to enter a critical region
  - it builds a message (name, process number, time)
  - sends the message to all processes (group communication)
- A process receives a message
  - sends OK
  - if in the critical region, does not reply
    - queues the request
  - if it wants to enter, compares timestamps of the request and its request, the lowest wins
  - queues the request if necessary
- When it exits the region, it sends OK messages to all processes on its queue



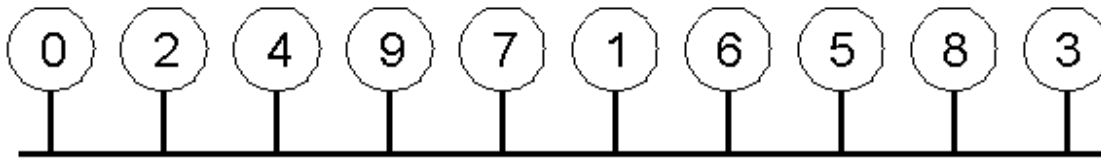
# A Distributed Algorithm



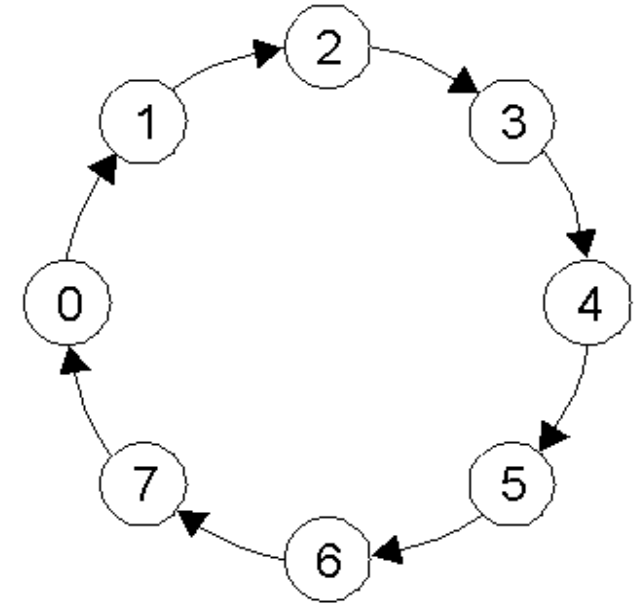
- Two processes want to enter the same critical region at the same moment.
- Process 0 has the lowest timestamp, so it wins.
- When process 0 is done, it sends an OK also, so 2 can now enter the critical region.



# A Token Ring Algorithm



(a)



(b)

a) An unordered group of processes on a network.

b) A logical ring constructed in software



# Comparison

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Distributed	$2(n - 1)$	$2(n - 1)$	Crash of any process
Token ring	1 to $\infty$	0 to $n - 1$	Lost token, process crash

- A comparison of three mutual exclusion algorithms.