Distributed System

Clock Synchronization

- Input.c---2151
- Input.o---2150
- Input file is changed so recompilation is needed
- Output.c---2144
- Output.o---2145
- No compilation is needed
- Logical Clocks: Internal consistency of the clocks
- Physical Clocks

Clock Synchronization

- Logical Clocks
  - Lamport algorithm
    - Happens before: a→b [a happens before b] means that all processes agree that first event a occurs then event b occurs
      - 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 events of the message being received by another process then a→b is also true. A message can not be received before it is sent, or even at the same time it is sent, since it takes a finite amount of time to arrive
    - Happens before is a transitive relation: If a→b and b→c then a→c
    - Concurrent event: If two events x and y happen in different processes that do not exchange messages then x→y is not true but neither is y→x. Nothing can be said about when they happened or which is first
    - For every event a, a time value C(a) is assigned on which all processes agree
    - If a→b then C(a)<C(b)
    - C must always go forward, never backward
    - Corrections to time can be made by adding a positive value, never by subtracting one

Clock Synchronization

- Logical Clocks
  - Lamport algorithm
    - 3 processes run on different machines with own clock running at own speed
    - 6 clock ticks in process 0 = 8 clock ticks in process 1 = 10 clock ticks in process 2
    - Each clock runs at different rate
    - At time 6, process 0 sends a message A to process 1
    - When the message arrives, the clock in process 1 reads 16
    - 10 ticks for the transmission
    - 16 ticks for the transmission of message B from process 1 to process 2
    - Message C from process 2 to process 1 leaves at 60 but arrives at 56
    - Message D from process 1 to process 0 leaves at 64 but arrives at 54
    - Impossible situation
    - Solution using the concept of happens before

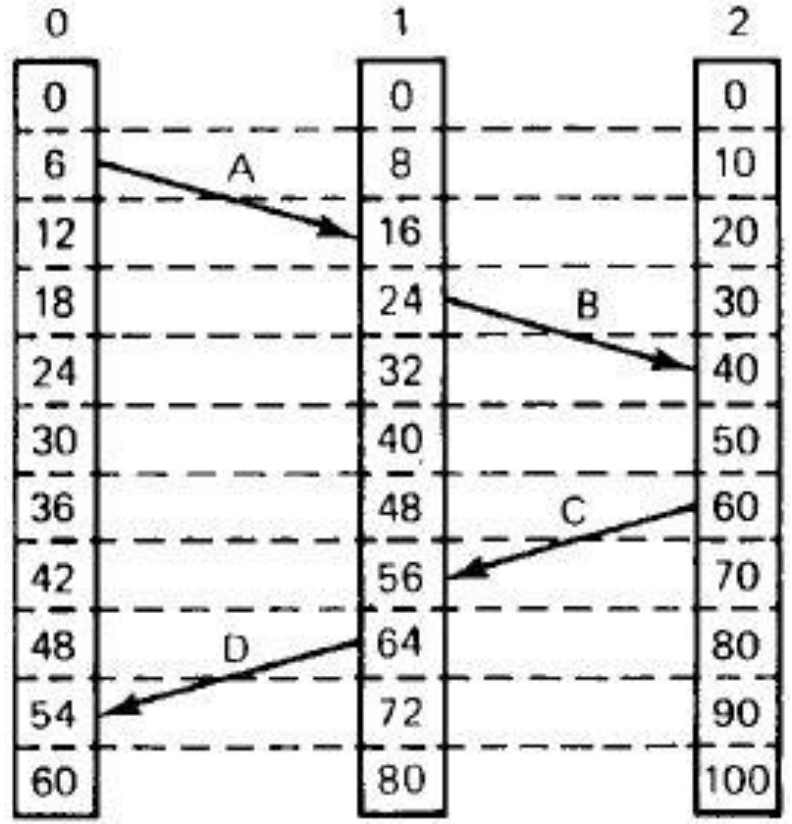

Figure 1: Three processes with own clocks running at different rates

Clock Synchronization

- Logical Clocks
  - Lamport algorithm
    - Message C from process 2 to process 1 leaves at 60, so it must arrive at 61 or later
    - Each message carries sending time according to sender's clock
    - When a message arrives and 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
    - Message C arrives at 61
    - Message D arrives at 70
    - No two events ever occur at exactly the same time (40.1, 40.2)

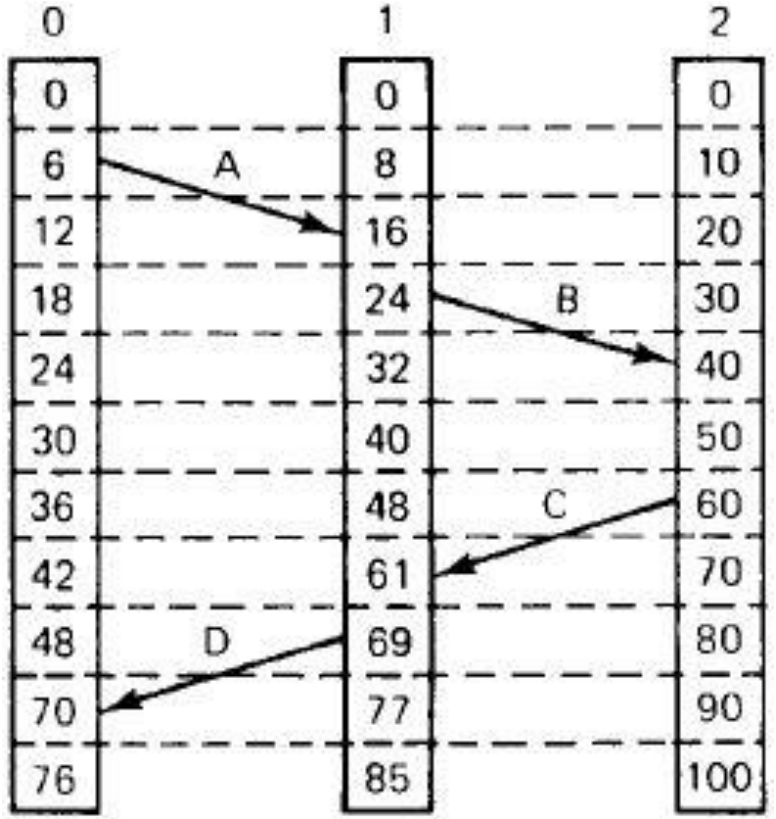

Figure 2: Lamport's algorithm corrects the clock

Clock Synchronization

- Physical Clocks
  - Christian's algorithm
    - › Periodically each machine sends a message to the time server asking current time
    - › The machine responds with a message containing its current time

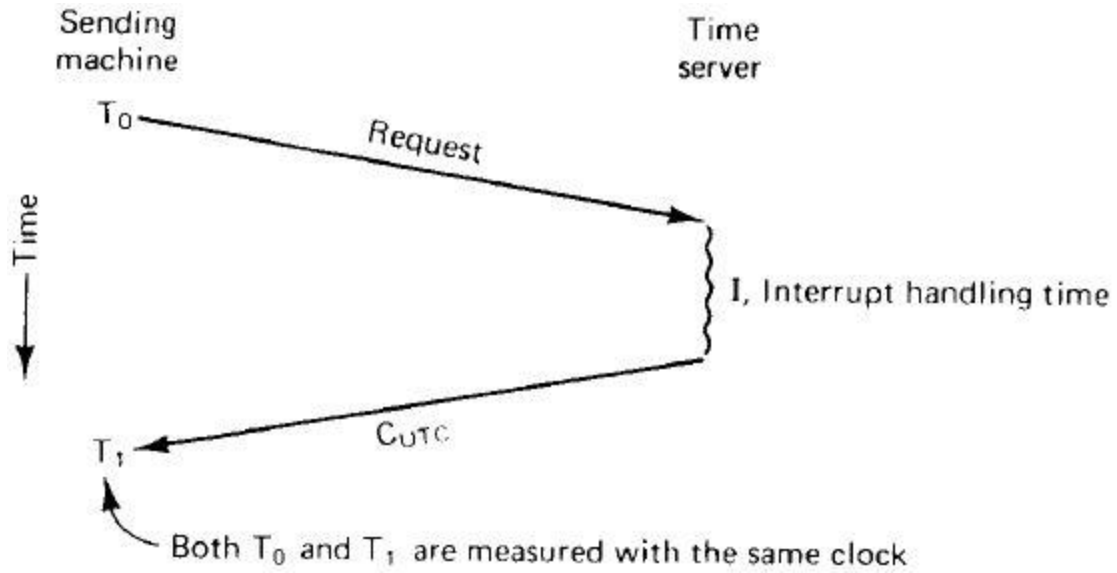

Figure 3: Getting the current time from a time server

Clock Synchronization

- Physical Clocks
  - Christian's algorithm
    - Time must never run backward
    - If sender's clock is fast C_{UTC} could be smaller than sender's current value of C
    - Can not just take over C_{UTC}
    - Gradually slowing down the clock

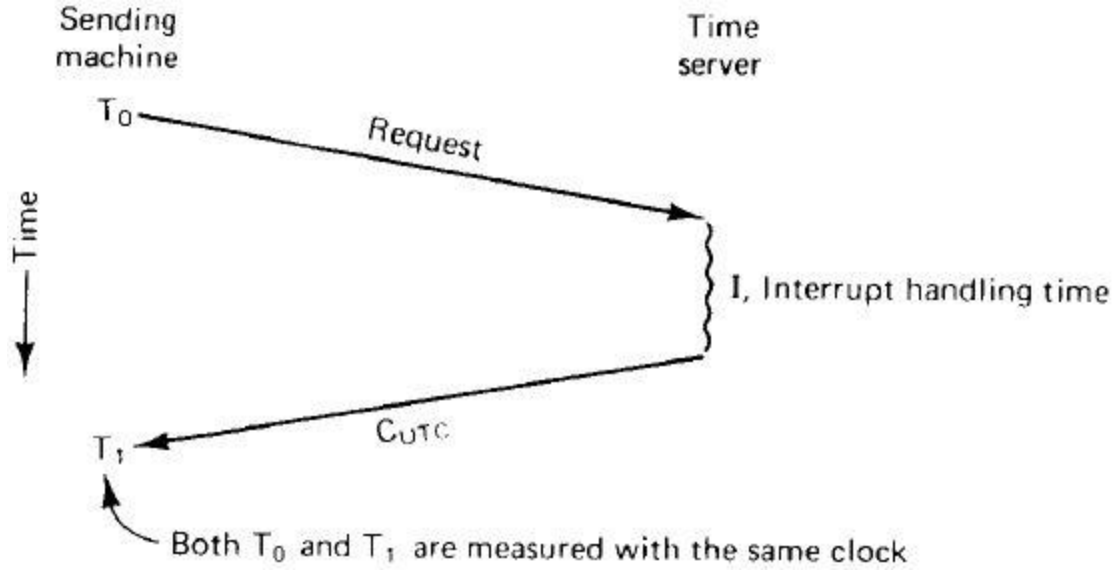

Figure 3: Getting the current time from a time server

Clock Synchronization

- Physical Clocks
  - Christian's algorithm
    - › It takes non zero amount of time for the time server's reply to get back to the sender
    - › Even the delay might be large
    - › Measuring the delay
    - › Increase the value of the message by $\frac{T_1 - T_0}{2}$ to estimate server's current time
    - › Increase the value of the message by $\frac{T_1 - T_0 - I}{2}$ to estimate server's current time
    - › Message from A to B might take a different route than message from B to A
    - › Take several measurements
    - › Average the value
    - › Fastest message can be considered

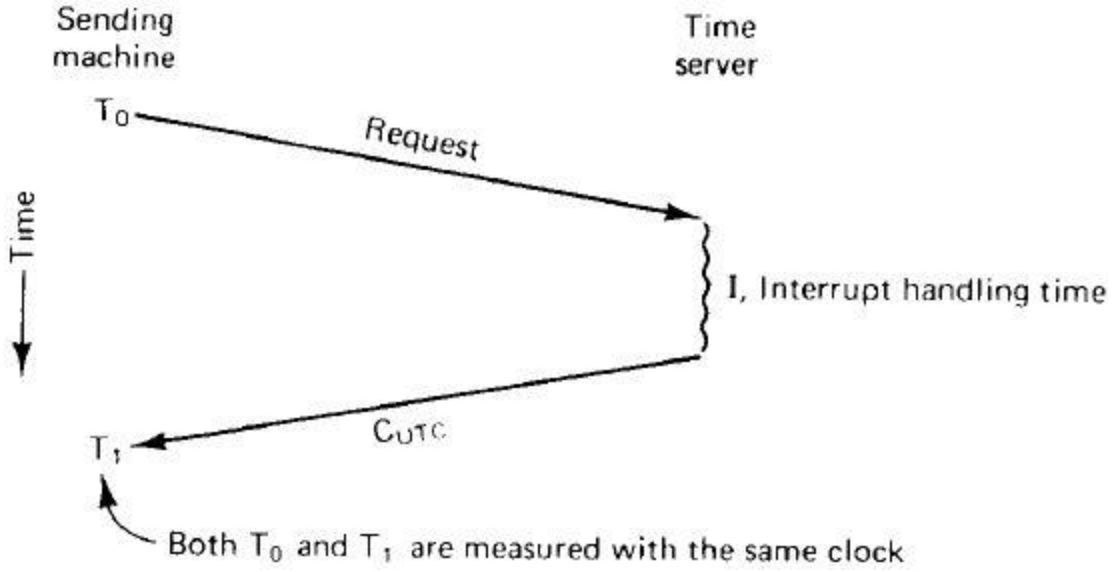

Figure 3: Getting the current time from a time server

Clock Synchronization

- Physical Clocks
  - Berkeley algorithm
    - › Christian's algorithm: Passive time server
    - › Berkeley algorithm: Active time server
    - › Time server periodically asks every machine about their time
    - › Based on answers, it computes an average time and tells other machines to adjust their clocks

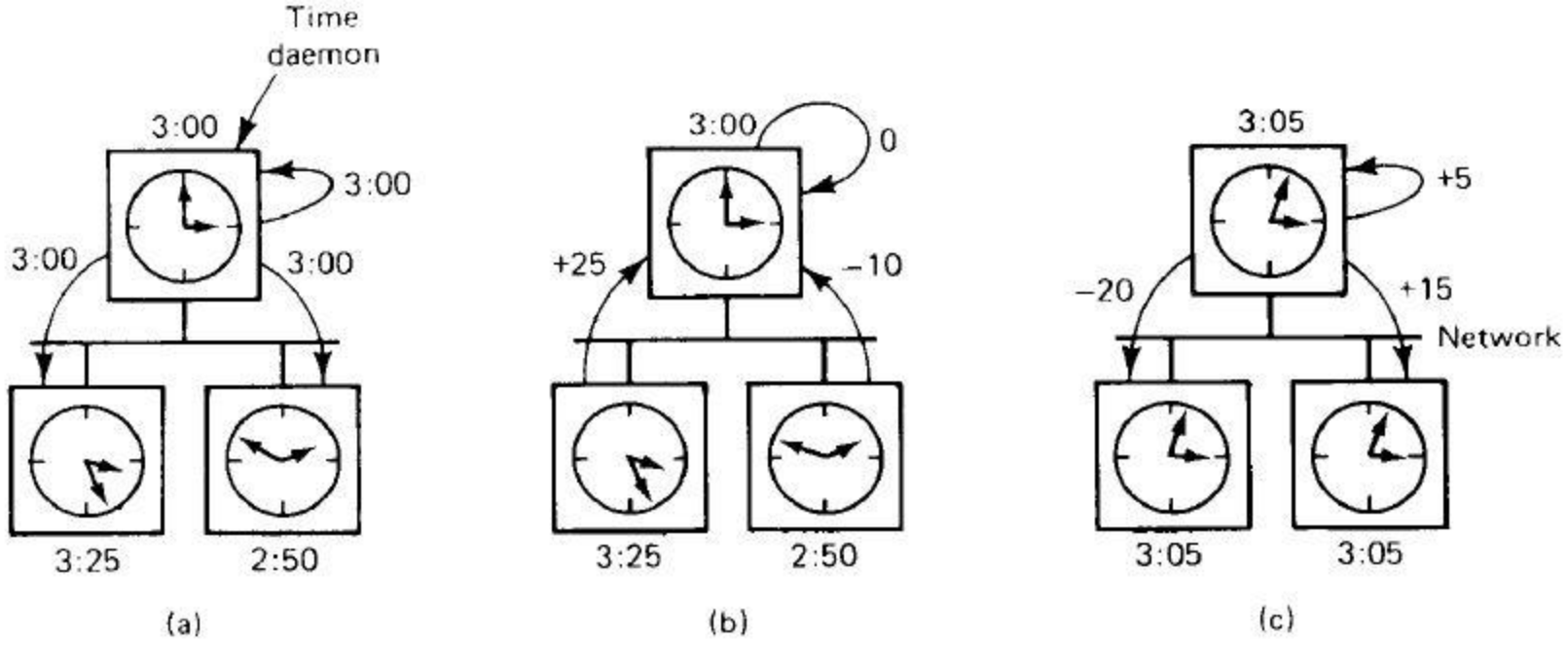

Figure 4: (a) Time daemon asks all other machines for their clock values
(b) The machines answer
(c) Time daemon tells everyone how to adjust their clocks

Election Algorithms

- Procedure to designate a process as coordinator/initiator
- Coordinator has special responsibility
- Initially all processes are same
- No way to distinguish a process as special one
- Each process has a unique number (network address)
- Process having the highest process number is recognized as the coordinator
- Every process knows the process number of every other process
- A process does not know which processes are down and which are up
- Election algorithm aims to select a coordinator on the consent of every process

Election Algorithms

The Bully Algorithm

- Process P notices that coordinator is no longer responding to requests
- Process P holds an election to find new coordinator
  - P sends an ELECTION message to all processes with higher process numbers than itself

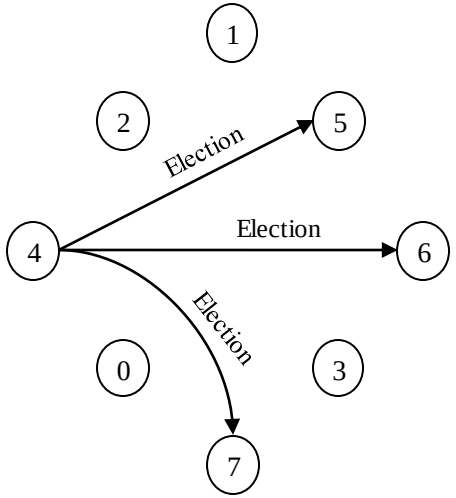

Figure 5: A process sends ELECTION message to all other higher numbered processes

- If no other process responds, P wins the election and becomes coordinator
- If one of the higher-ups answers, it takes over. P's job is done

Election Algorithms

The Bully Algorithm

- Any process can get ELECTION message from any other process with lower process number
- After receiving an ELECTION message, a process sends OK message to the sender to indicate that he is alive and will take over
- Now the receiver holds an election
- This procedure goes on and on until all processes give up except one
- The process which wins all the steps of election is the new coordinator
- New coordinator sends coordinator message to all other processes to inform its victory

Figure 6: Steps of bully election algorithm

Election Algorithms

The Bully Algorithm

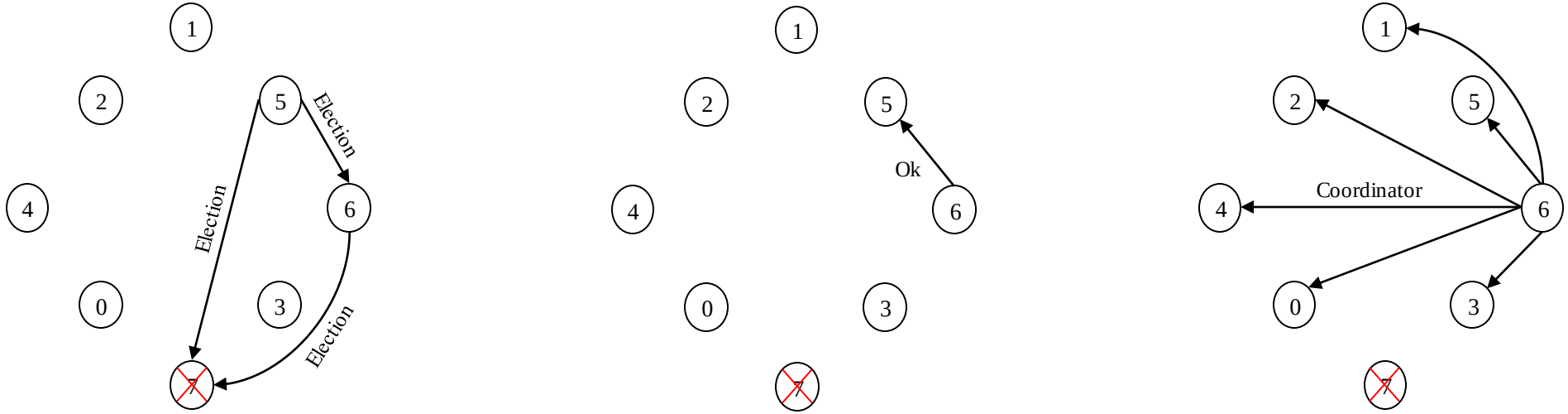

Figure 7: Steps of bully election algorithm

- If process 7 is ever restarted, it will send a COORDINATOR message to all other

Election Algorithms

A Ring Algorithm

- Processes are physically or logically ordered (assumption)
- Each process knows about its successor
- If any process notices that the coordinator is not functioning then it builds an ELECTION message and sends it to its successor
- ELECTION message contains process number of a processor
- If the successor is down, the sender skips over the successor and goes to the next number along the ring, or the one after that, until a running process is located
- At each step, the sender adds its own process number to the list in the message to make itself a candidate for coordinator
- The message gets back to the process that started sending message
- Own process number in the message indicates that it is the message which was sent by that processor
- COORDINATOR message is sent to all processes indicating the new coordinator

Election Algorithms

A Ring Algorithm

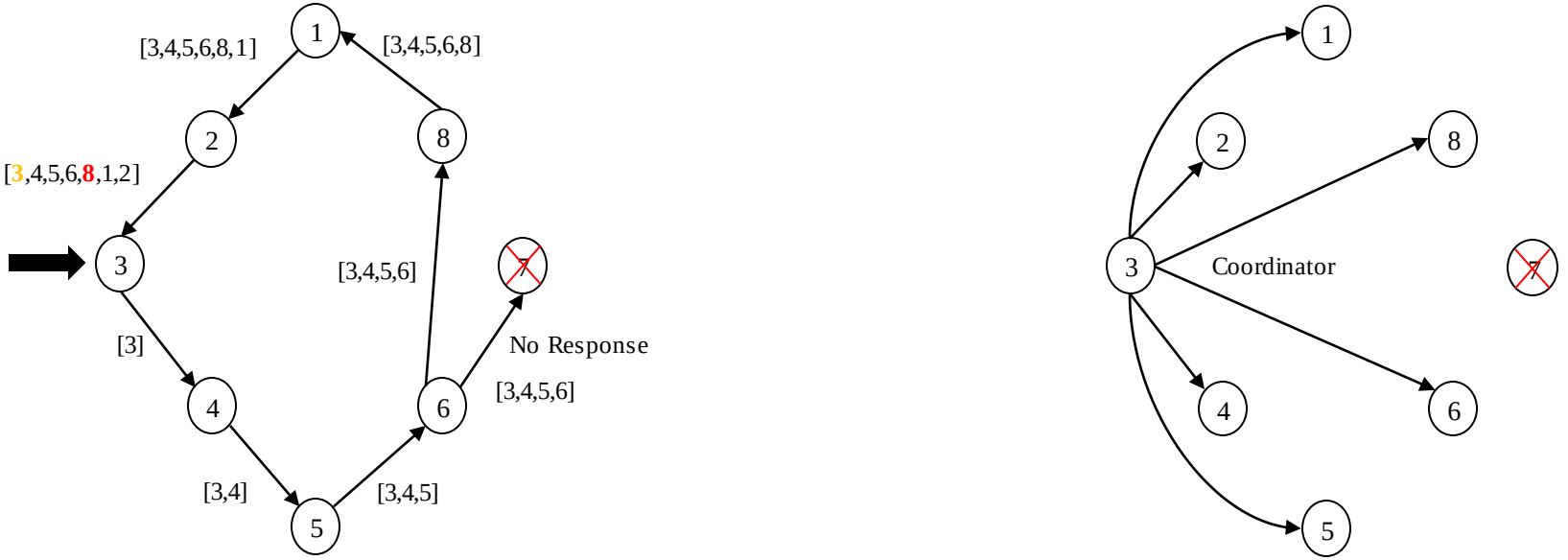

Figure 8: Steps of ring election algorithm

Mutual Exclusion

- Preventing simultaneous access to a shared resource

A centralized algorithm

- One process elected as coordinator
- If any process wants to enter critical region, it sends a request message to the coordinator mentioning the name of the critical region and asks for permission
- If no process is using that critical region then the coordinator sends a request back granting permission
- The requesting process enters the critical region after the reply arrives

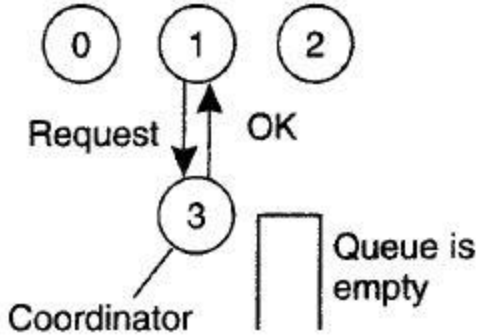

Figure 9: Steps of granting a permission

Mutual Exclusion

A centralized algorithm

- One process (process 1) is in the critical region
- Another process (process 2) asks for permission to enter that critical region
- Coordinator can not grant permission as it knows one process is using it
- The coordinator refrains from replying to process 2
- The request from process 2 is kept in queue
- When process 1 exits the critical region it sends a message to coordinator releasing its exclusive access
- Coordinator takes the 1st item from the queue and sends that process a grant message
- That process (process 2) sees the grant message and enters the critical region

(a) (b)

Figure 10: (a) No reply from coordinator
(b) Request granted

Mutual Exclusion

A centralized algorithm

- Guarantees mutual exclusion
- Fair algorithm
- No starvation
- Easy to implement
- Only 3 messages (request, grant, release) required
- Single point of failure
- Dead coordinator?
- Performance bottleneck

Mutual Exclusion

A distributed algorithm

- When a process wants to enter a critical region, it builds a message
- Message: Name of the critical region, process number, current time
- The process sends the message to all other processes including itself
- Every message is acknowledged
- When a process receives a request message, three different cases have to be distinguished
  - Receiver sends back OK: If the receiver is not in the critical region and does not want to enter
  - No reply: If the receiver is in the critical region
  - If the receiver wants to enter the critical region but has not entered yet then timestamp is considered (lower timestamp wins)
    - Receiver sends back OK: If the incoming message has lower timestamp
    - No reply and queue incoming request: Own message has lower timestamp
- After sending requests asking permission to enter a critical region, a process sits back and waits until everyone else has given permission
- If all permissions are in, it may enter the critical region
- When it exits critical region, it sends OK to all processes on its queue and deletes all

Mutual Exclusion

A distributed algorithm

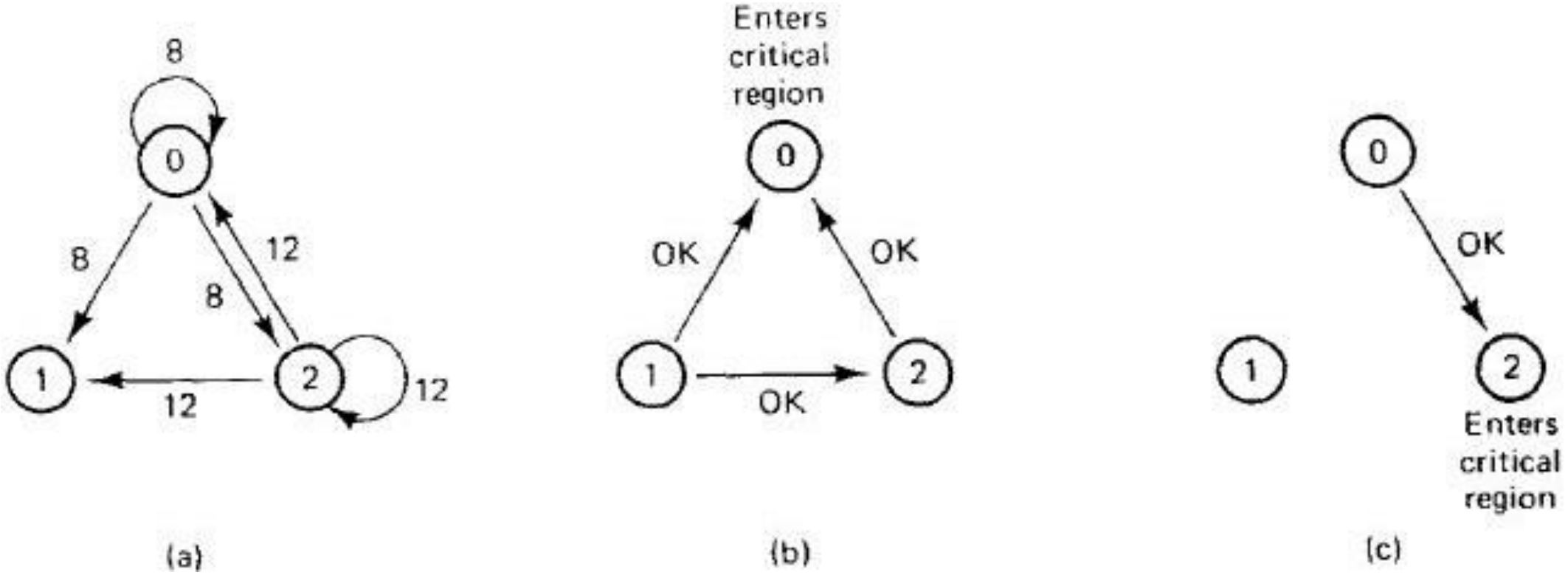

Figure 11: (a) Two processes want to enter the same critical region at the same moment
(b) Process 0 has the lowest timestamp, so it wins
(c) When process 0 is done, it sends an OK also, so 2 can now enter the critical region

Mutual Exclusion

A distributed algorithm

- The number of messages required per entry is 2(n-1), where n is the total number of processes in the system
- No single point of failure
- n points of failure
- If any process crashes, it will fail to respond to requests. This silence will be interpreted as denial of permission, thus blocking all subsequent attempts by all processes to enter all critical regions
- Improvement: A process should be allowed to enter critical region when it succeeds in getting permission from most of the other processes. After a process has granted a permission it can not grant the same permission to another process until the first one has released that permission

Mutual Exclusion

A token ring algorithm

- When the ring is initialized, process 0 is given a token which circulates around the ring
- When a process acquires a token, it may enter a critical region
- After exiting from the region, it passes the token to the neighbor
- It is not permitted to enter a second critical region with the same token

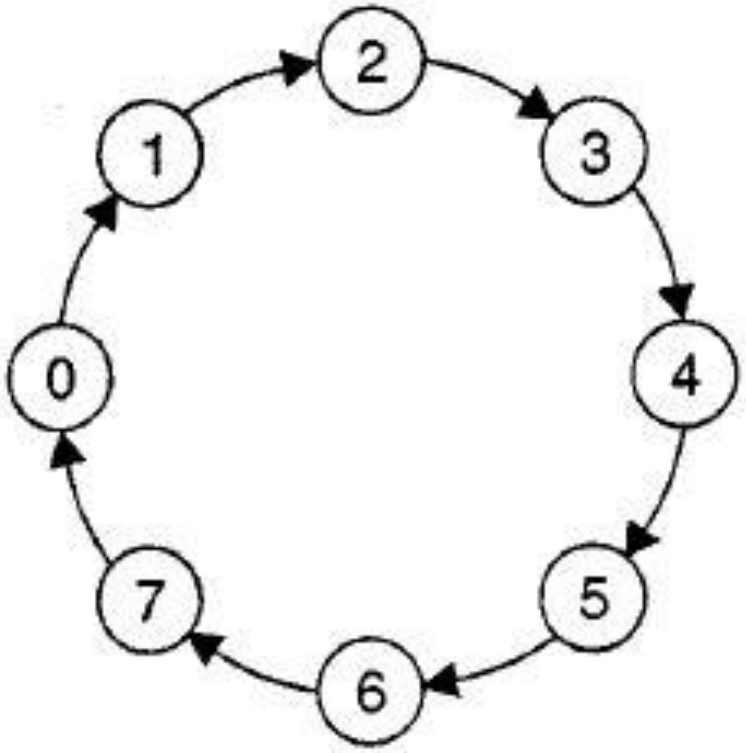

Figure 12: A logical ring constructed

Mutual Exclusion

A token ring algorithm

- If the token is ever lost, it must be regenerated
- How to know whether a token is lost?
- Process crash

Mutual Exclusion

Comparison

Algorithm	Messages per entry/exit	Delays before entry	Problems
Centralized	3	2	Coordinator crash
Distributed	2(n-1)	2(n-1)	Crash of any process
Token ring	1 to ∞	0 to n-1	Lost token, process crash

Deadlocks in Distributed Systems

- Strategies to handle deadlocks
  - The ostrich algorithm (ignore the problem)
  - Detection (detect and recover)
  - Prevention (statically make deadlocks structurally impossible)
  - Avoidance (avoid deadlocks by allocating resources carefully)

Deadlocks in Distributed Systems

Centralized deadlock detection

- Each machine maintains the resource graph for own processes and resources
- Central coordinator maintains the resource graph for the entire system (union of all individual graphs)
- Coordinator kills off one process to break deadlock
- Several possibilities
  - Whenever an arc is deleted or added to the resource graph, a message can be sent to the coordinator providing the update
  - Periodically every process can send a list of arcs added or deleted since the previous update
  - Coordinator can ask for information when it needs

Deadlocks in Distributed Systems

Centralized deadlock detection

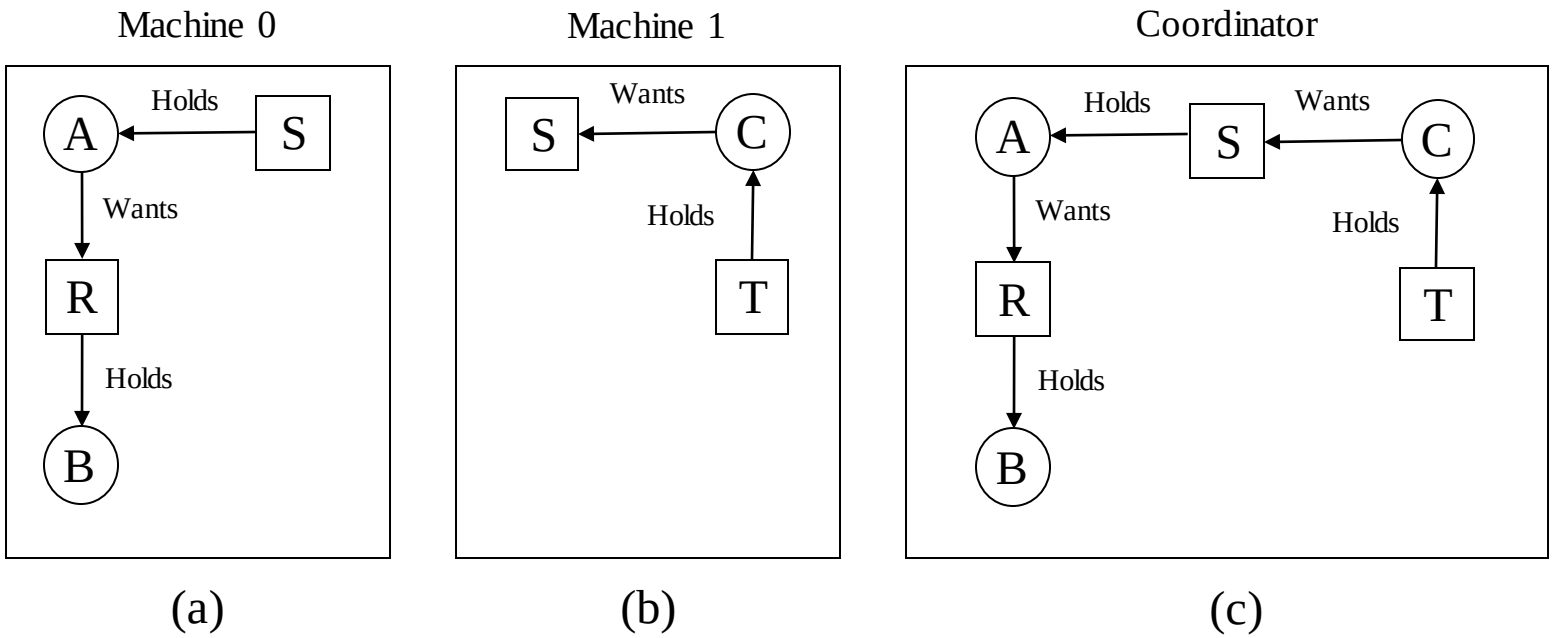

Figure 13: (a) Initial resource graph for machine 0
(b) Initial resource graph for machine 1
(c) The coordinator's view

Deadlocks in Distributed Systems

Centralized deadlock detection

- B releases R, asks for T
- Machine 0 sends a message to coordinator announcing the release of R
- Machine 1 sends a message to coordinator announcing B is waiting for T

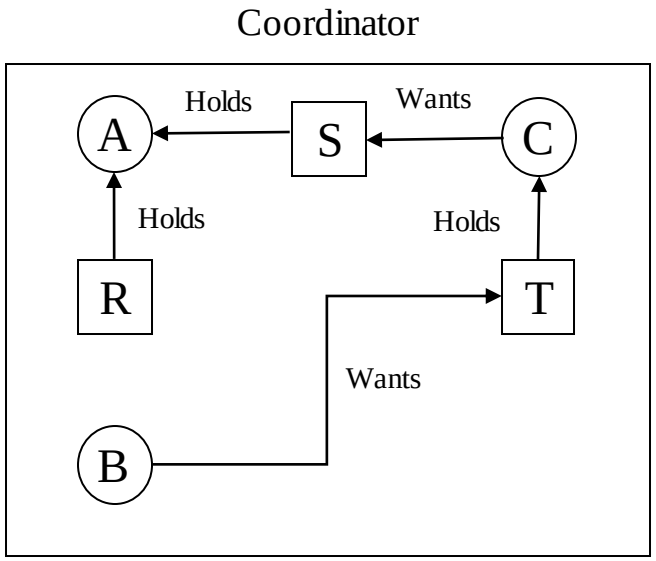

Figure 14: Coordinator's view after the mentioned operations

Deadlocks in Distributed Systems

Centralized deadlock detection

- B releases R, asks for T
- Machine 0 sends a message to coordinator announcing the release of R (arrives later)
- Machine 1 sends a message to coordinator announcing B is waiting for T (arrives first)
- False deadlock

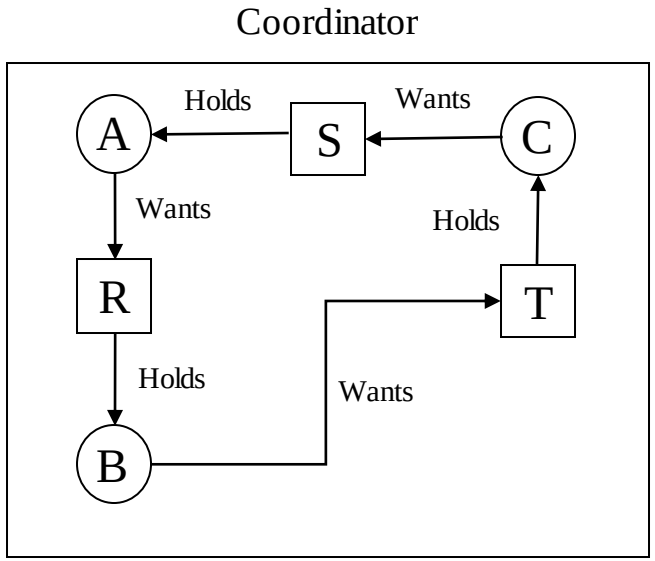

Figure 15: Coordinator's view after the delayed message

Deadlocks in Distributed Systems

Centralized deadlock detection

- Handling false deadlock
  - Using Lamport's algorithm to provide global time
  - Message from machine 1 triggered by request from machine 0
  - Message from machine 1 will have a later timestamp
  - After receiving message from machine 1 which leads to deadlock, it sends a message to every machine asking for messages with earlier/lower timestamp
  - After getting reply from every machine the coordinator sees that there is no deadlock
  - Requires global time
  - Expensive

Deadlocks in Distributed Systems

Distributed deadlock detection

Chandy-Misra-Haas algorithm

- Processes are allowed to request multiple resources at once
- Probe message: The process that just blocked, the process sending the message, the process to whom it is being sent
- When the message arrives, the recipient checks to see if it itself is waiting for any processes
- If so, the message is updated, keeping the first field unchanged
- If a message goes all the way around and comes back to the original sender, a cycle exists and the system is deadlocked

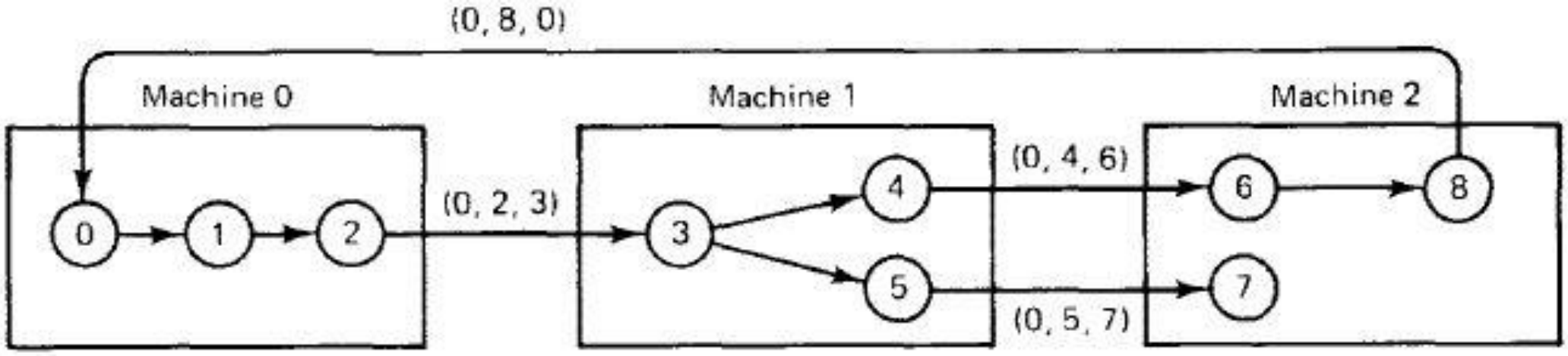

Figure 16: The Chandy-Misra-Haas distributed deadlock detection algorithm

Deadlocks in Distributed Systems

Distributed deadlock detection

Ways to break deadlock

- Process that initiated the probe can commit suicide after detecting deadlock
- Overkill: If several processes invoked the algorithm simultaneously then each of them would kill itself. killing only one process would be enough to break the deadlock

---

- Adding identity to the end of the probe message
- Kill the process with the highest number
- If several processes invoked the algorithm simultaneously then only the process with the highest number would be killed

Deadlocks in Distributed Systems

Distributed deadlock prevention

- Carefully designing the system so that deadlocks are structurally impossible

Wait-die deadlock prevention algorithm

- Wait: An old process wants a resource held by a young process
- Die: A young process wants a resource held by an old process

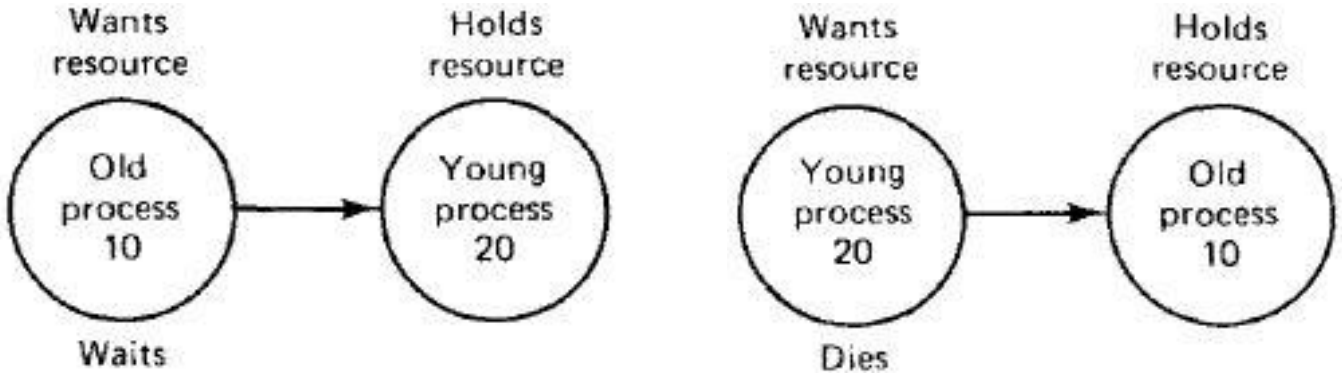

Figure 17: The wait-die deadlock prevention algorithm

Deadlocks in Distributed Systems

Distributed deadlock prevention

Wound-wait deadlock prevention algorithm

- Preempt: An old process wants a resource held by a young process
- Wait: A young process wants a resource held by an old process

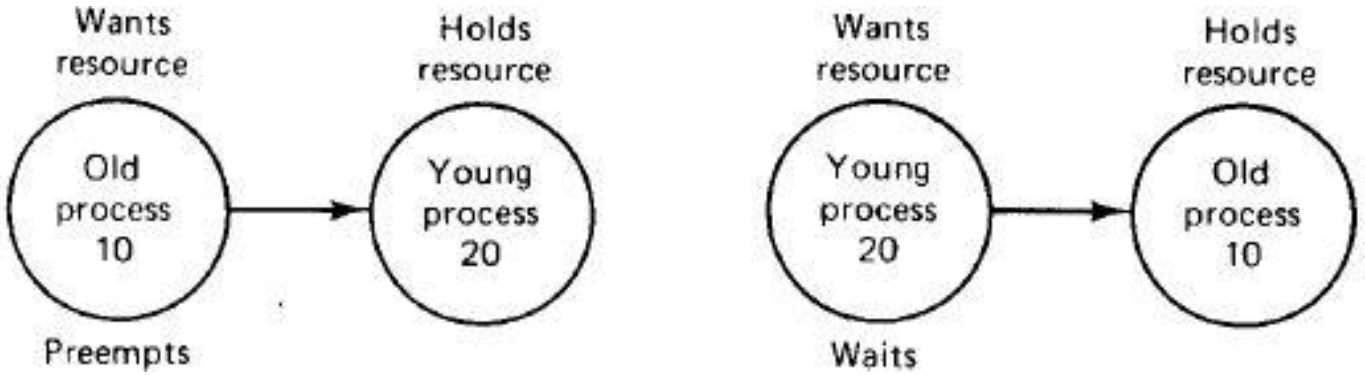

Figure 18: The wound-wait deadlock prevention algorithm