Parallel & Distributed System

Synchronization

Md. Biplob Hosen

Lecturer, IIT-JU

Email: biplob.hosen@juniv.edu

Contents

- Clock Synchronization
- Logical Clocks
- Mutual Exclusion
- Election Algorithms

- Reference Books
  - ✓ Distributed Systems: Principles and Paradigms, 3rd Edition by Andrew S. Tanenbaum & Maarten van Steen, Publisher: Pearson Prentice Hall **[CH-06]**.

Overview

- Why do processes communicate in DS?
    1) To exchange messages.
    2) To synchronize processes.
- Why do processes synchronize in DS?
    1) To coordinate access of shared resources.
    2) To order events.

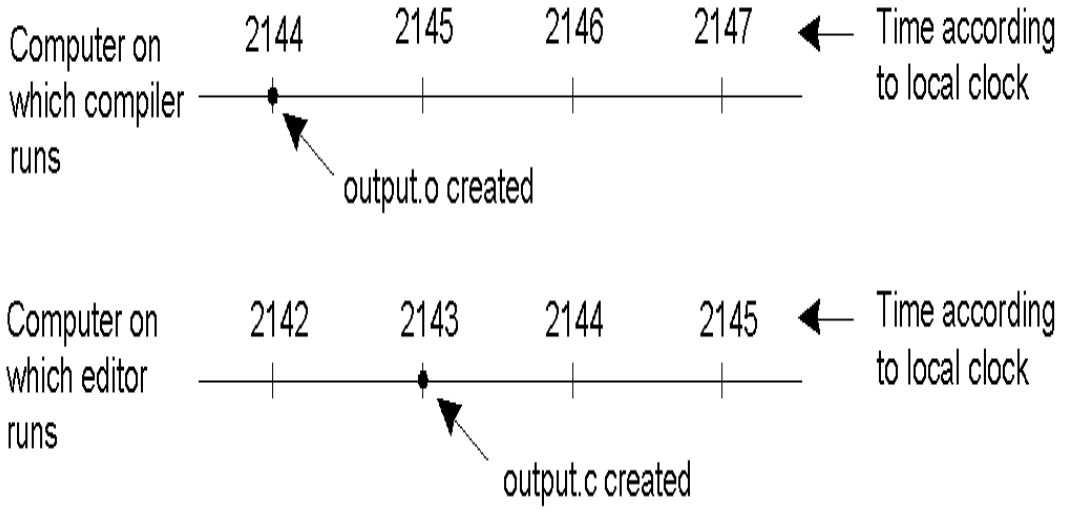
- We mainly concentrate how processes can synchronize.
- For example, it is important that multiple processes do not access a shared resource (such as printer) at the same time.
- Sometimes, multiple processes may need to agree on the ordering of an event, such as, whether message m_1 from process P was sent before or after message m_2 from process Q.
- Synchronization in distributed system is often much more difficult compared to synchronization in uniprocessor or multiprocessor system.
- In many cases, it is important that a group of processes can appoint one process as a **coordinator.**

Clock Synchronization

- In a centralized system, time is unambiguous. When a process wants to know the time, it makes a system call and the kernel tells it.
- 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 the value A got.
- Normally, in UNIX, large programs are split up into multiple source files, so that a change to one source file only requires one file to be recompiled, not all the files. Therefore, if a program has 100 files, not having to recompile everything because one file has been changed greatly increases the speed at which programmers can work.
- When the programmer has finished changing all the source files, he starts a program "**make**", that examine the times at which all the source and object files were last modified.

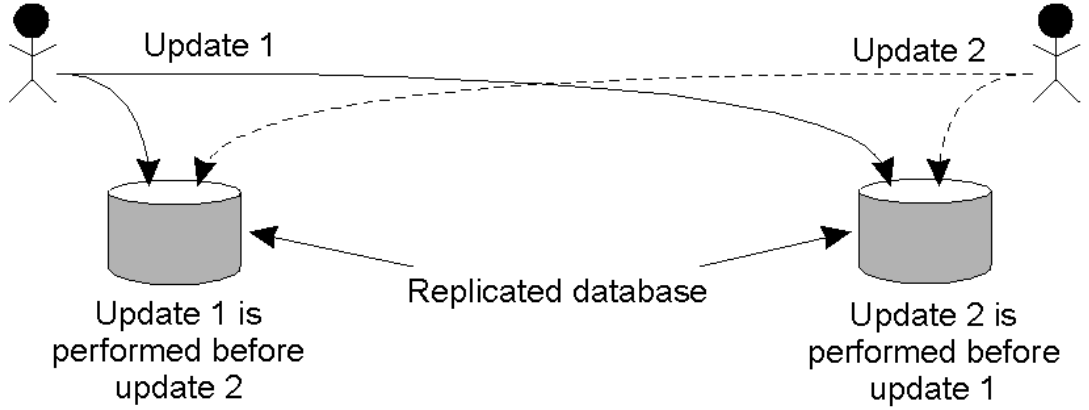
Clock Synchronization

- 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.
- But what will happen in a distributed system in which there were no global agreement on time.
- Suppose that **input.o** has time 2144 and shortly thereafter **input.c** is modified but is assigned time 2143 because the clock on its machine is slightly behind.
- In this case, “make” will not call the compiler.
- The resulting executable binary program will contain a mixture of object files from the old sources and the new sources.



Logical Clock

- For many purpose, it is sufficient that all machines agree on the same time but it is not essential that this time also agree with the real time of the world.
- For a certain class of algorithms, it is the internal consistency of the clock that matters, not whether they are particularly close to the real time.
- For such algorithms **logical clock** is conventional to speak.
- **Observation:** It may be sufficient that every node agrees on a current time – that time need not be 'real' time.
- Taking this one step further, in some cases, it is adequate that two systems simply agree on the order in which system events occurred.



Lamport's Logical Clocks

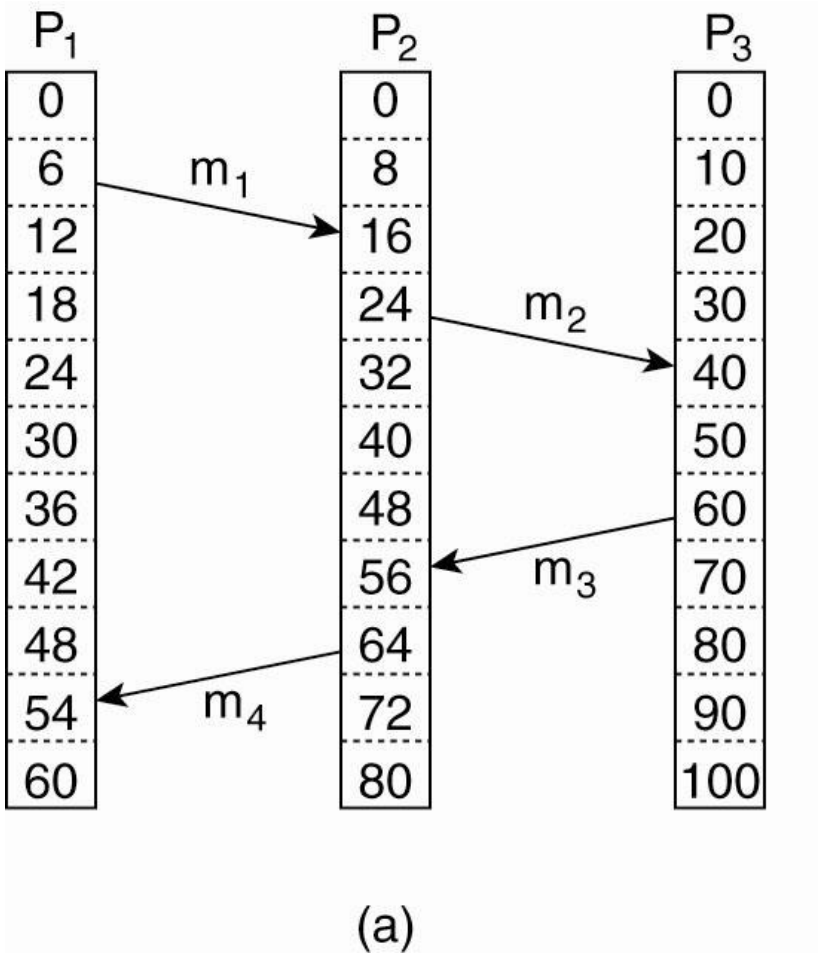
- Lamport showed that although clock synchronization is possible, it is not needed to be absolute. If two processes do not interact, it is not necessary that their clocks be synchronized.
- What usually matters is not that all processes agree on exactly what time it is, but rather that they agree on the order in which events occur.
- We can define a systems as a collection of processes that communicate by exchanging messages. Each process consists of a sequence of events.
- To synchronize logical clocks, Lamport defined a relation called "**happens-before**".
- The "happens-before" relation on process's events (a→b) can be observed directly in two situations:
  1) If a and b are events in 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 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.

Lamport's Logical Clocks

- This "happen-before" relation is **transitive** and can be used to define a partial ordering on the events in a distributed system.
- if a → b and b → c, then a → c.
- Now, we need a way of assigning a time value C(a) (for event a) on which all processes agree, where if a → b then C(a) < C(b).
- In addition, 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.
- Consider three processes. The processes run on different machines, each with its own clock, running at its own speed.
- When the clock has ticked 6 times in process-0, it has ticked 8 times in process-1 and 10 times in process-2. Each clock runs at a constant rate, but the rates are different.

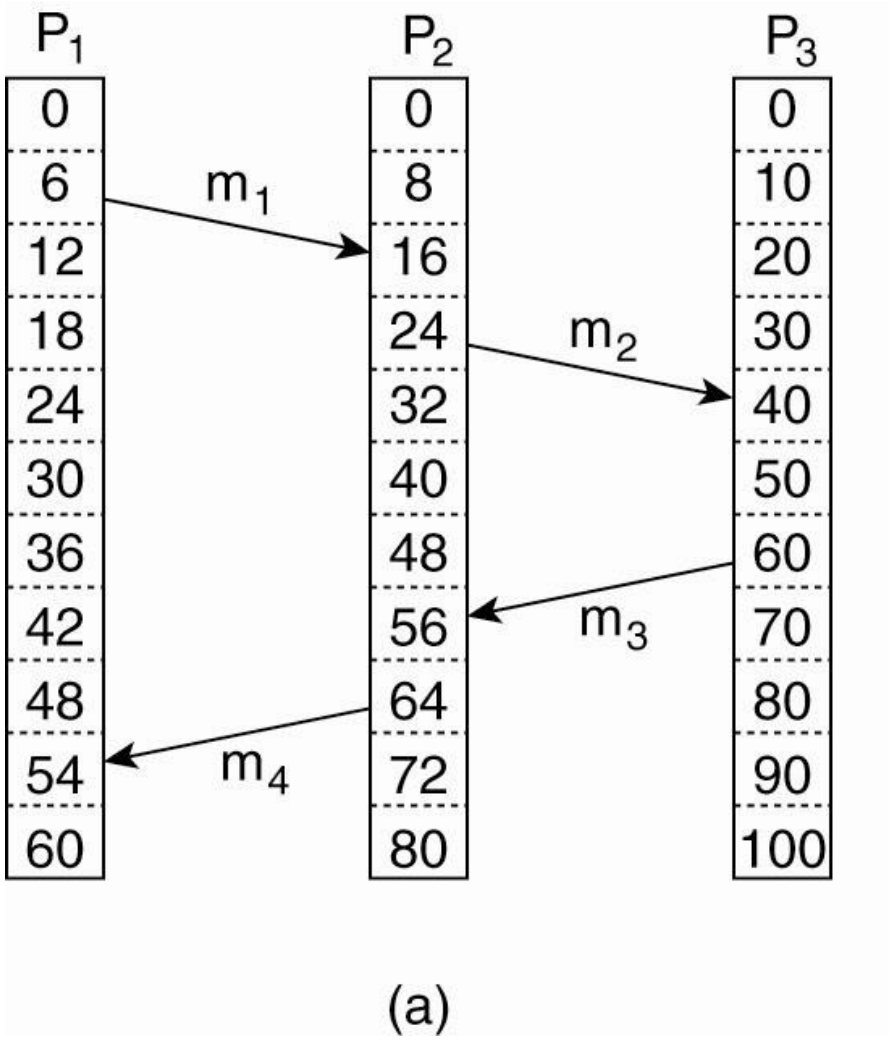
Lamport's Logical Clocks

- At time 6, process-1 sends a message m_1 to process-2. Let us assume it arrives at process-2 at time 16.
- If the message carries the starting time 6, process-2 will conclude that it took 10 ticks to make the journey.
- Similarly message m_2 from process-2 to process-3 takes 16 ticks, which is also possible.



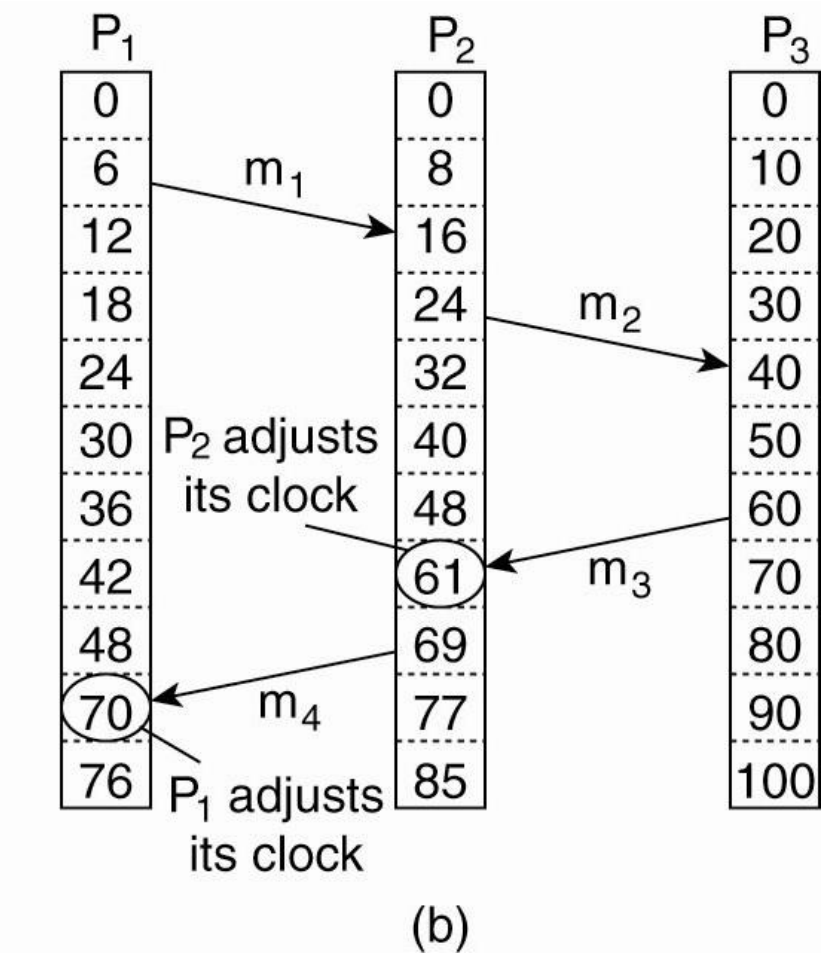
Lamport's Logical Clocks

- Now message m_3 from process-3 to process-2 leaves at 60 and arrives at 56. Again message m_4 from process-2 to process-1 leaves at 64 and arrives at 54, which is clearly impossible.



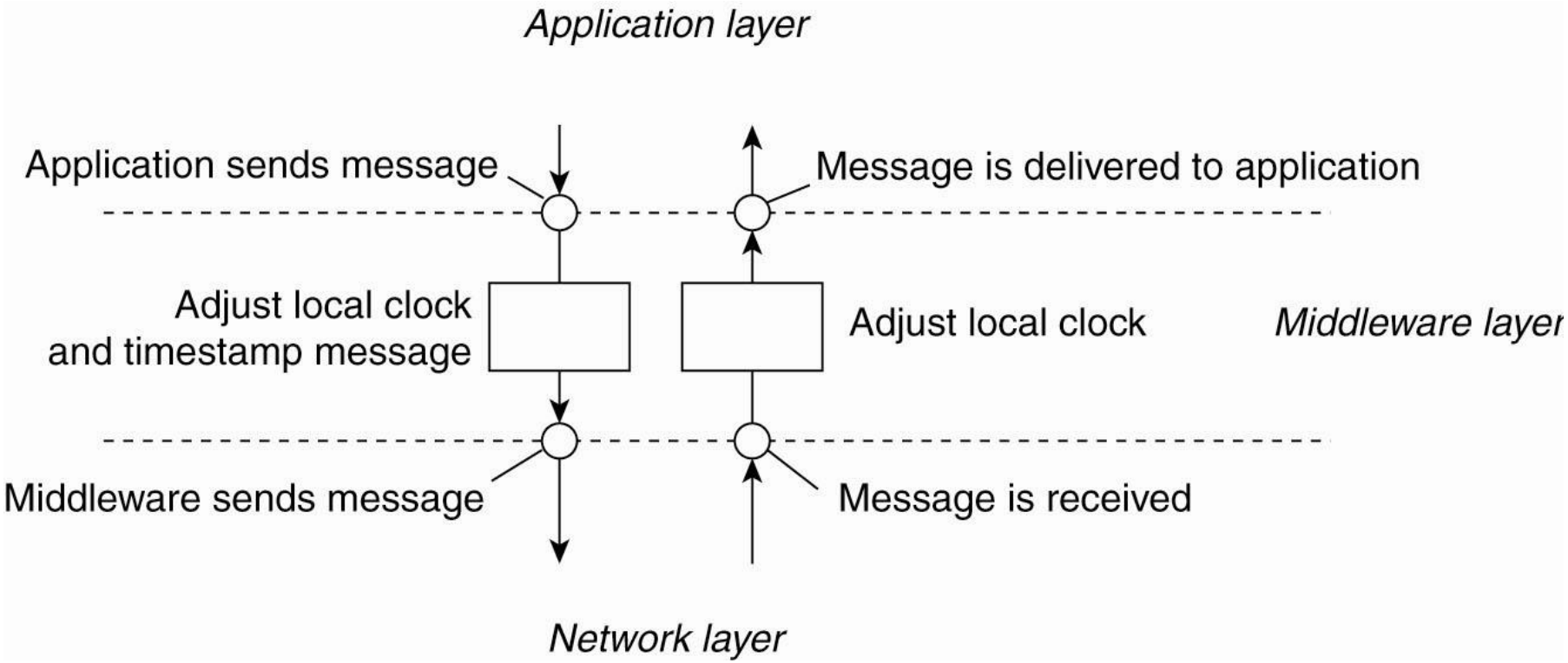
Lamport's Logical Clocks

- Lamport's solution follows directly from the happens-before relation.
- Since, m_3 left at 60, it must arrive at 61 or later. Therefore each message carries a 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 that the sending time.



Lamport's Logical Clocks

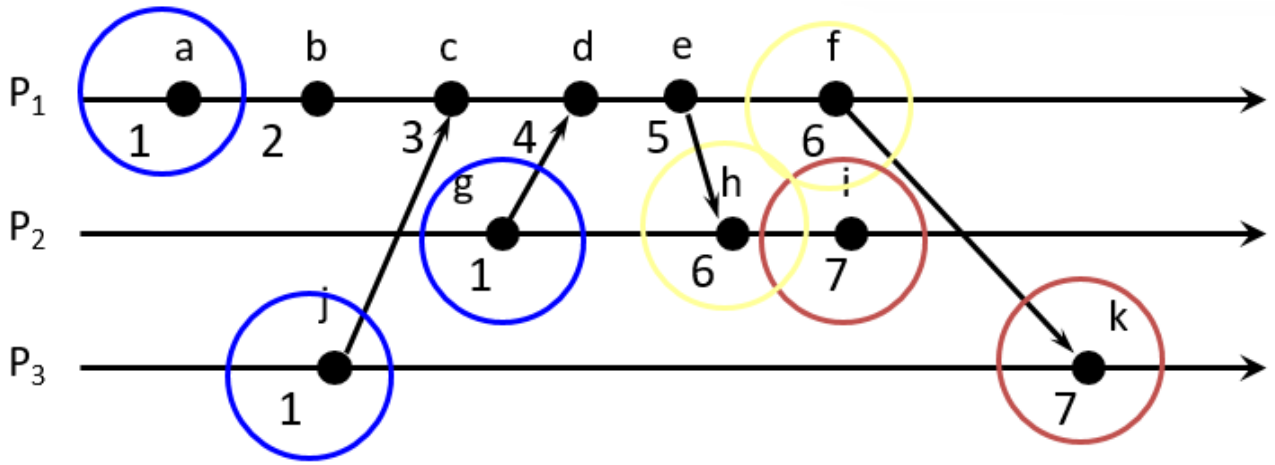
- Figure: The positioning of Lamport's logical clocks in distributed systems:



Problem: Identical Timestamps

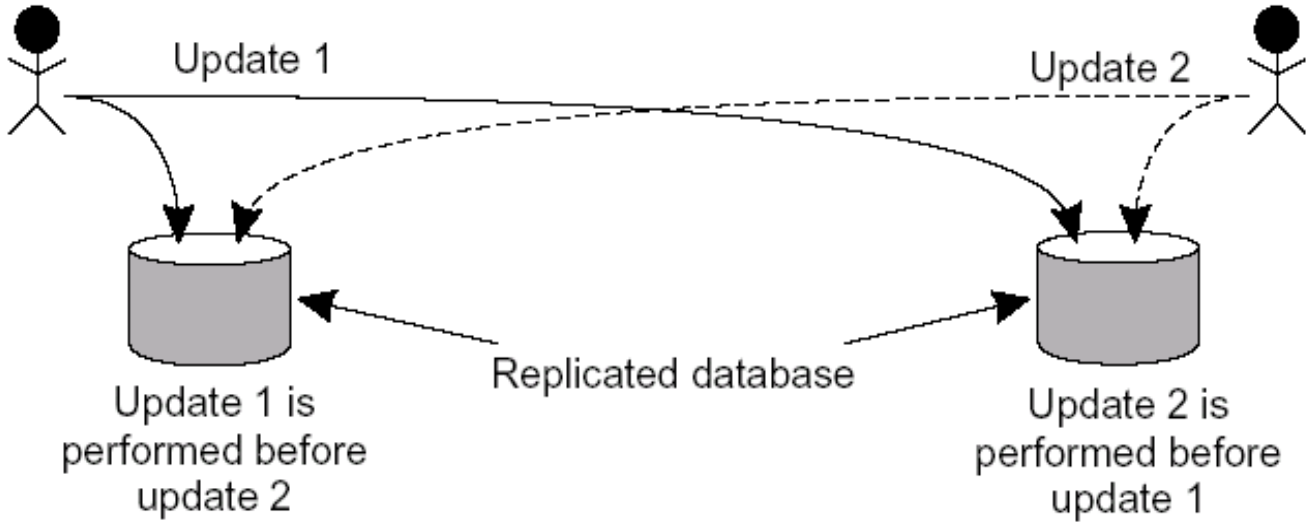
- In some situation, an additional requirement is desirable: no two events occur at exactly the same time.
- To achieve this goal, we can attach the number of the process in which the even occurs to the low-order end of the time, separated by a decimal point.
- Thus, events happen in processes 1 and 2, both with time 1, the former becomes 1.1 and the latter becomes 1.2
- Using this method, we now have a way to assign time to all events in a distributed system subject to the following condition:

1. If a happens before b in the same process then C(a)<C(b).
2. If a and b represent the sending and receiving of a message, respectively, then C(a)<C(b).
3. For all distinctive events a and b, C(a)≠C(b).



Example: Totally-Ordered Multicast

- Let us 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 dhaka and sylhet).
- A customer in Sylhet wants to adds BDT100 to his account (initial value: BDT1000).
- At the same time, a bank employee in Dhaka initiates an update by which the customer's account is to be increased by 1% 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 orders as shown in the figure:



Continue…

- The customer's update operation is performed in Sylhet before the interest update.
- In contrast, the copy of the account in Dhaka replica is first updated with the 1% interest and after that with the 100tk deposit.
- Therefore, the Sylhet database will record a total of BDT 1,111, whereas the Dhaka database records BDT 1,110.
- The problem that we are faced with is that the two update operations should have been performed in the same order at each copy. The important issue is that both copies should be exactly the same.
- Solution is "**totally-ordered multicast**"; that is a multicast operation by which all messages are delivered in the same order to each receiver.

Continue…

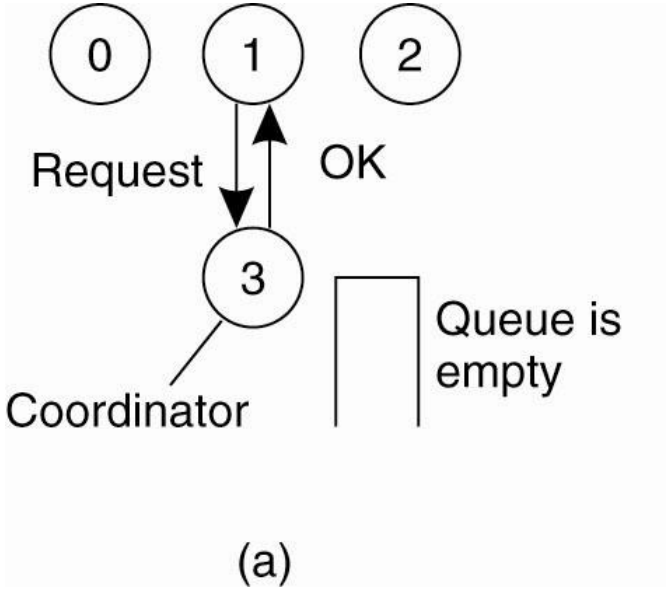
- Lamport's timestamps can be used to implement totally-ordered multicast in a completely distributed fashion.
- Assume, messages from the same sender are received in order sent and no messages are lost.
- Each message multicast is timestamped with the current logical time of sender. All members of the group receive a copy of any message sent, including the sender.
- When a process receives a message, it puts it into a local queue ordered by timestamp. The receiver multicasts an acknowledgement to other processes (with larger timestamp).
- The interesting aspect of this approach is that all processes will eventually have the same copy of the local queue.
- A process can deliver a queued message with timestamp t to the application (removing it and all associated acknowledgements) when that message is at the head of the queue and it has been acknowledged by each other process with a message stamped later than time t.

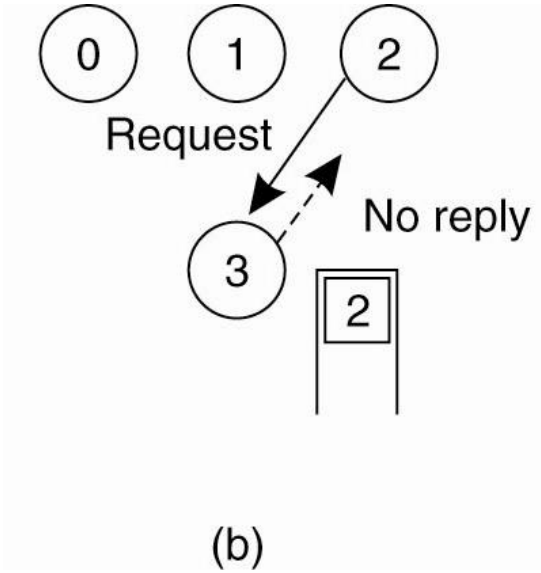
Mutual Exclusion

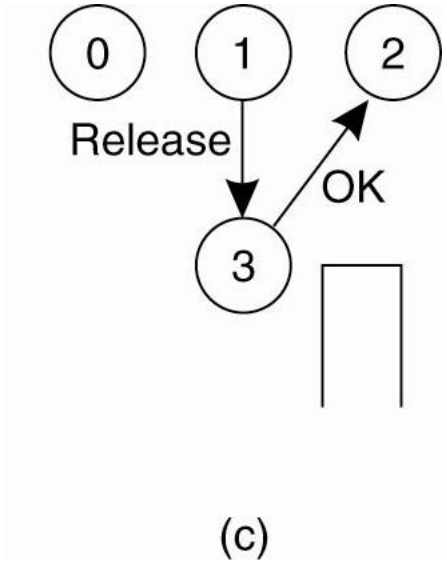
- **Problem:** A number of processes in a distributed system want exclusive access to some shared resource.
- **Basic solutions:**
  - Via a centralized server using semaphore, monitor and similar constructs.
  - Completely distributed, with no topology imposed.
  - Completely distributed, making use of a (logical) ring.

A Centralized Algorithm

- 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.
- When process-1 exits the critical region, it tells the coordinator, who then replies to 2.







A Centralized Algorithm

Drawback:

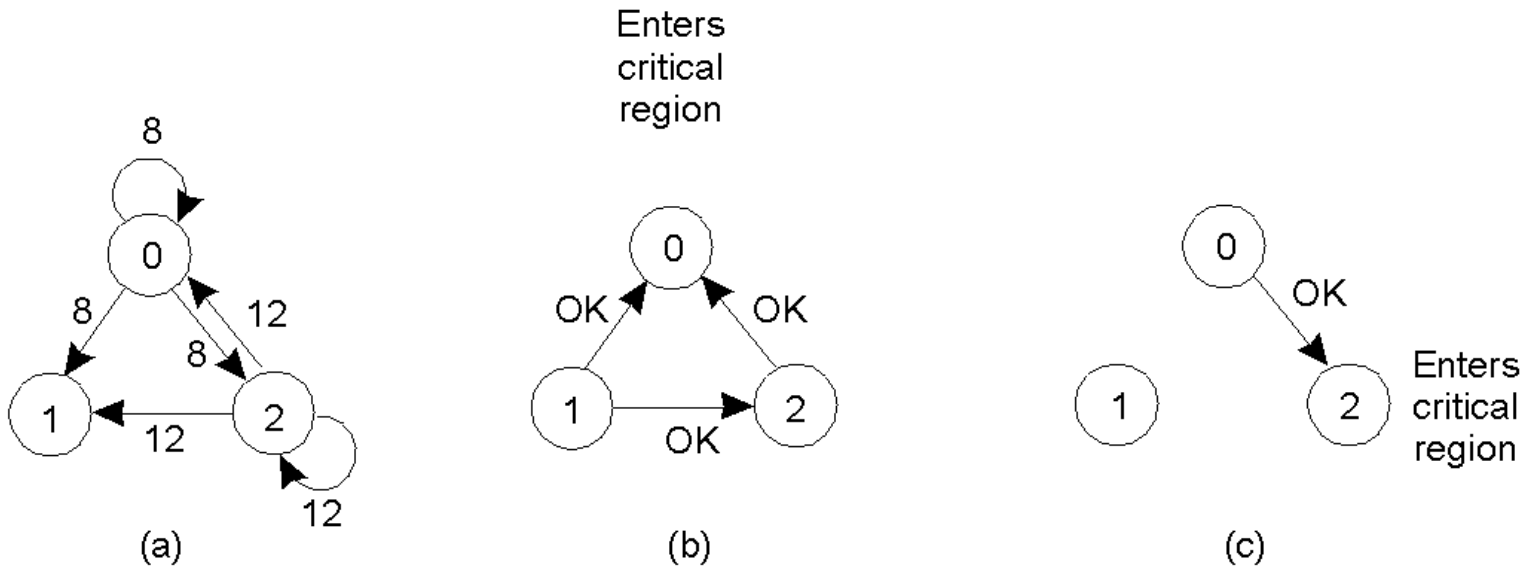
- The coordinator is a single point of failure, so if it crashes, the entire system may go down.
- If processes normally block after making a request, they cannot distinguish a dead coordinator from “permission denied” since in both cases no message comes back.

Distributed Mutual Exclusion: Lamport Clocks

1. When a process wants to enter a critical region, it builds a message containing the name of the critical region it wants to enter, its process number and the current time.
2. It then sends the message to all the processes, conceptually including itself.
3. The sending of the message is assumed to be reliable, that is, every message is acknowledged.
4. When a process receives a request message from another process, the action it takes depends on its state with respect to critical region named in the message.
   a) If the receiver is not in the critical region and does not want to enter it, it sends back an OK message to the sender.
   b) If the receiver is already in the critical region, it does not reply. Instead, it queues the request.
   c) If the receiver wants to enter the critical region but has not yet done so, it compares the timestamp in the incoming message with the one contained in the message that it has sent everyone. The lowest one wins. If the incoming message is lower, the receiver send back an OK message. If its own message has a lower timestamp, the receiver queues the incoming request and sends nothing.

Distributed Mutual Exclusion: Lamport Clocks

5. After sending out requests asking permission to enter a critical region, a process sits back and waits until everyone else has given permission. As soon as the permissions are in, it may enter the critical region.
6. When it exits the critical region, it sends OK message to all processes on its queue and deletes them all from the queue.



a) Two processes want to enter the same critical region at the same moment.

b) Process-0 has the lowest timestamp, so it wins.

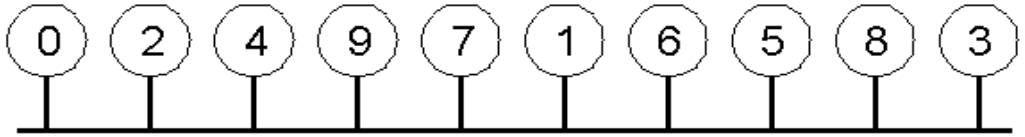
c) When process-0 is done, it sends an OK also, so process-2 can now enter the critical region.

Drawback of the Algorithm

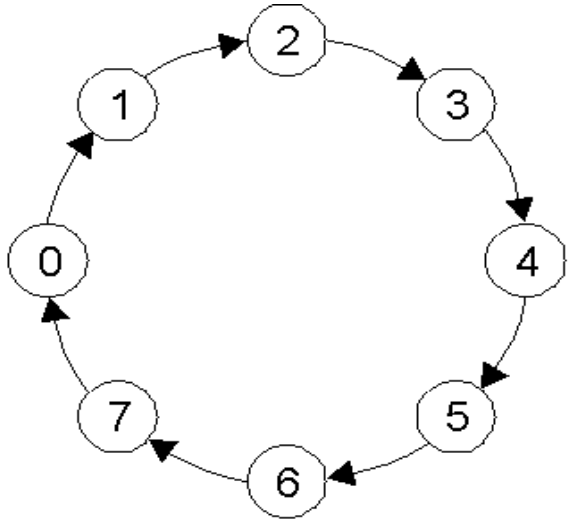
- Here mutual exclusion is guaranteed without deadlock or starvation.
- If the total number of processes in the system is n then the number of request per entry is now 2*(n-1), where (n-1) request messages to all other processes, and subsequently (n−1) OK messages, one from each other process.
- Now, the single point of failure has been replaced by n point of failure.
- If any process crashes, it will fail to respond to requests. This silence will be interpreted (incorrectly) as denial of permission; thus blocking all subsequent attempts by all processes to enter all critical regions.

Distributed Mutual Exclusion: Token Rings

- An unordered group of processes on a network.
- A logical ring constructed in software in which each process is assigned a position in the ring.
- Algorithm works by passing a token around the ring. When a process holds the token, it decides if it needs to access the resource at this time. If so, it holds the token while it does so, passing the token on once done.
- Problems if the token is ever 'lost' – token loss may also be difficult to detect.



(a)



(b)

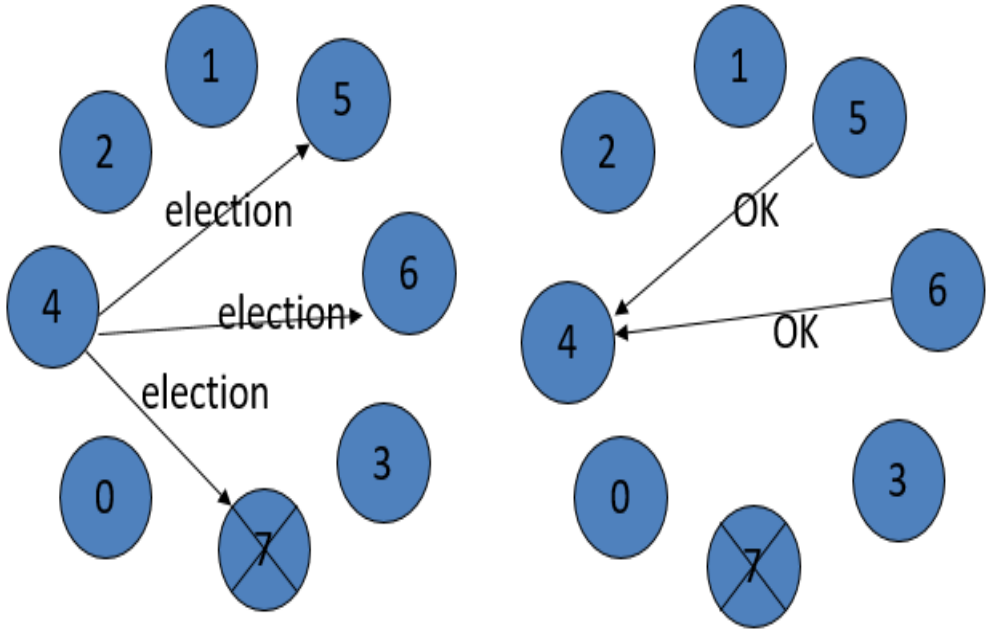
Need for a Coordinator

- Many algorithms used in distributed systems require a coordinator.
- For example, the centralized mutual exclusion algorithm.
- In general, all processes in the distributed system are equally suitable for the role.
- Election algorithms are designed to choose a coordinator.
- Any process can "**call an election**" (initiate the algorithm to choose a new coordinator).
  - ✓ There is no harm (other than extra message traffic) in having multiple concurrent elections.
- Elections may be needed when the system is initialized, or if the coordinator crashes or retires.

The Bully Algorithm

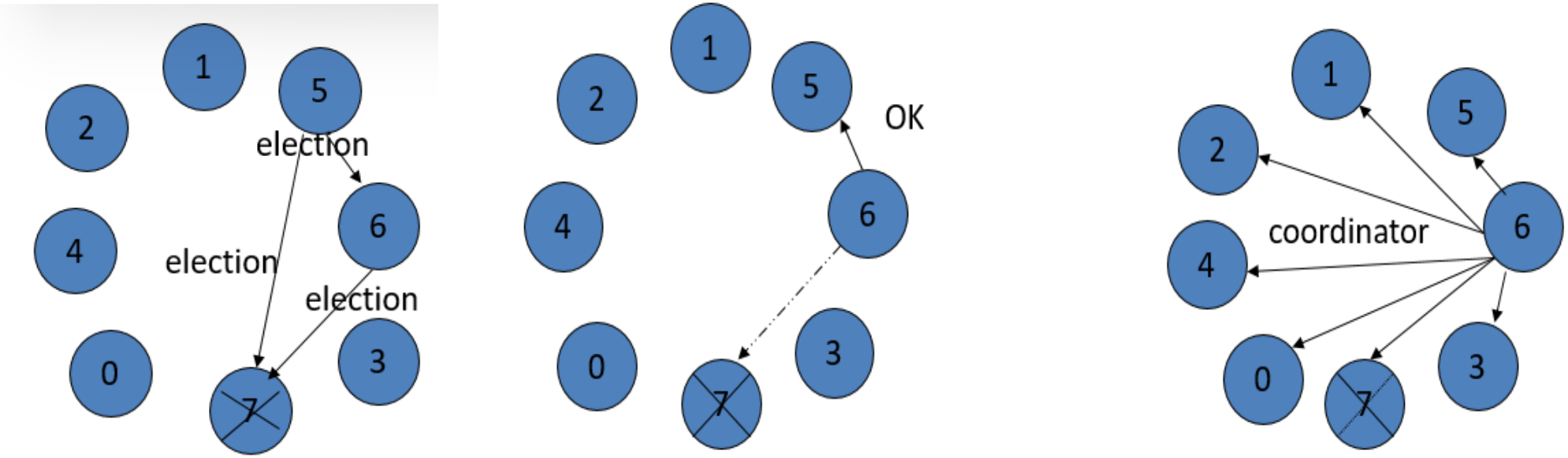
- Process p calls an election when it notices that the coordinator is no longer responding.
- High-numbered processes “bully” low-numbered processes out of the election, until only one process remains.
- When a crashed process reboots, it holds an election. If it is now the highest-numbered live process, it will win.

- Process p sends an election message to all higher-numbered processes in the system.
- If no process responds, then p becomes the coordinator.
- If a higher-level process (q) responds, it sends p a message that terminates p’s role in the algorithm.



The Bully Algorithm

- The process q now calls an election (if it has not already done so).
- Repeat until no higher-level process responds. The last process to call an election “wins” the election.
- The winner sends a message to other processes announcing itself as the new coordinator.



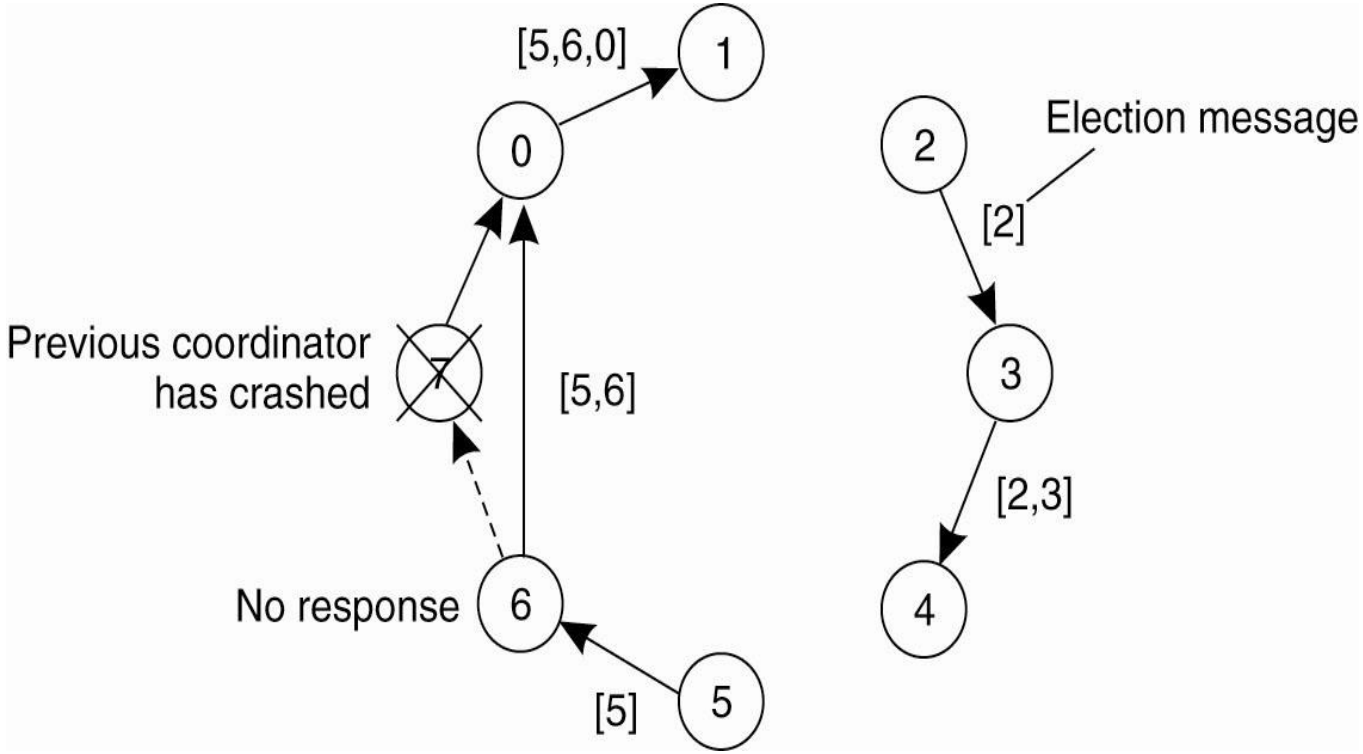
- If 7 comes back on line, it will call an election.

A Ring Algorithm - Overview

- The ring algorithm assumes that the processes are arranged in a logical ring and each process knows who its successor is.
- When any process notices that the coordinator is not functioning, it builds an ELECTION message containing its own process number and sends the message to its successor.
- 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, effectively making itself a candidate to be elected as coordinator.
- Eventually, the message gets back to the process that started it all.
- At this 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 number) and who the members of the new ring are.
- When this message has circulated once, it is removed and everyone goes back to work.

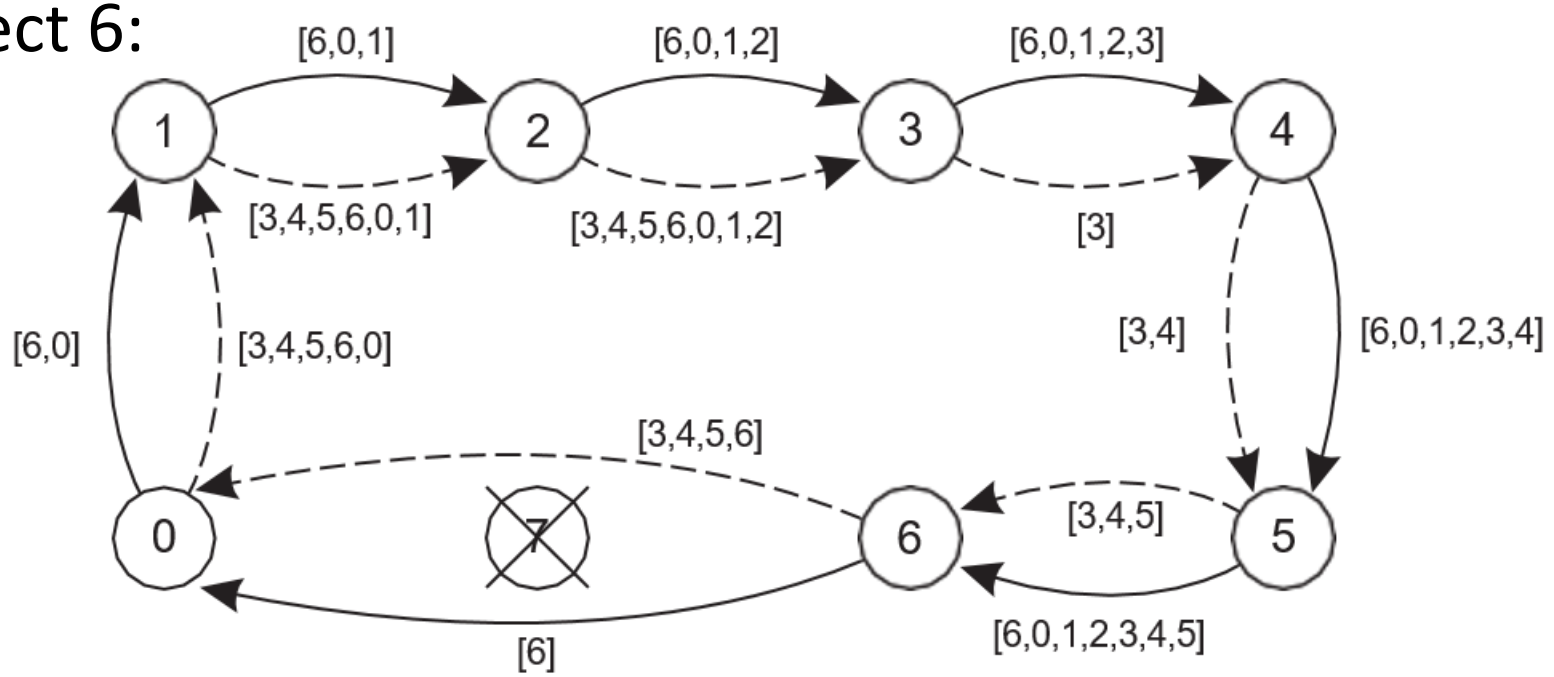
A Ring Algorithm

- P thinks the coordinator has crashed; builds an ELECTION message which contains its own ID number.
- Sends to first live successor.
- Each process adds its own number and forwards to next.
- OK to have two elections at once.



Ring Algorithm - Details

- When the message returns to p, it sees its own process ID in the list and knows that the circuit is complete.
- P circulates a COORDINATOR message with the new high number.
- Here, both 3 and 6 elect 6:
  [5,6,0,1,2,3,4]
  [2,3,4,5,6,0,1]



The solid line shows the election messages initiated by P6; The dashed one the messages by P3

Thank You ☺