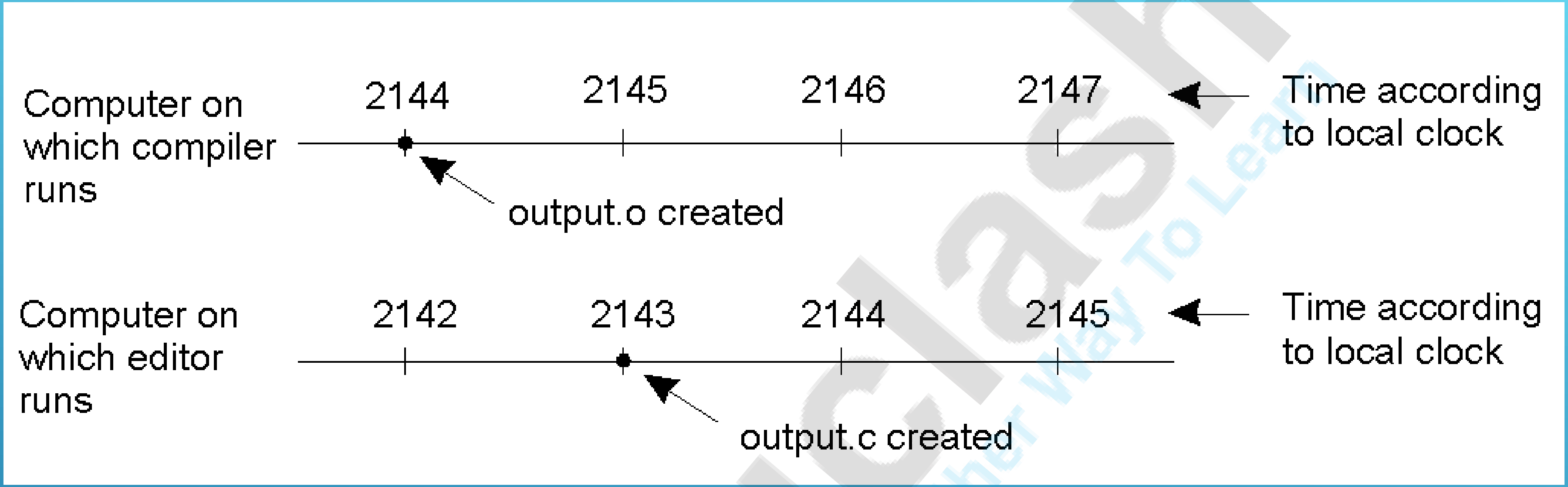
CLOCK SYNCHRONIZATION

Clock synchronization

- Every computing system needs a timer mechanism called a computer clock, to keep track of current time, accounting purposes such as time spent on a process, CPU utilization, Disk I/O etc.
- In a distributed system an application may have processes that concurrently run on multiple nodes of the system
- For correct results several such distributed applications require that the clocks of the nodes are synchronized with each other
- **A distributed on-line reservation system to be fair, the only remaining seat booked almost simultaneously from two different nodes of the system should be offered to the one who booked first, even if the time difference between the two are small**
- Consider the situation in the next slide

Clock synchronization (Cont'd)


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

- **When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time**

Physical Clocks

- The computer timer is usually a precisely machined quartz crystal, when kept under tension, 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 called *clock tick* is generated and the counter is reloaded from the holding register content
- The value in the holding register is taken in such a way that 60 clock ticks occur in every second
- Though the crystal oscillates at fixed frequency, slight differences in crystals result in difference in the rate at which two clocks run, which leads to drift in clock.
- Secondly each crystal has a built in error in oscillation depending on various environmental factors

Requirement for Clock Synchronization

- Synchronization of computer clocks with real time or external clocks:
  - This type of synchronization is mainly required for real time applications
  - External clock synchronization allows the system to exchange information about the timing of events with an external source.
  - An external time source often used for synchronizing computer clocks with real time is the *Coordinated Universal Time (UTC)*
  - The UTC is an international standard
  - Many standard bodies disseminate UTC signals by radio, telephone and satellite
  - Commercial devices known as *time providers* are available to receive and interpret these signals

Clock synchronization (Cont'd)

  - Computers equipped with time provider devices can synchronize their clocks with these time signals
- Another method is mutual (or internal) synchronization of clocks of different nodes of the system
  - This type of synchronization is mainly required for those applications that require a consistent view of time across all nodes of a distributed system as well as for time duration of distributed activities
  - Note that externally synchronized clocks are also internally synchronized
  - Converse is not true as they might drift arbitrarily far from external time over the passage of time

Issues in Clock synchronization

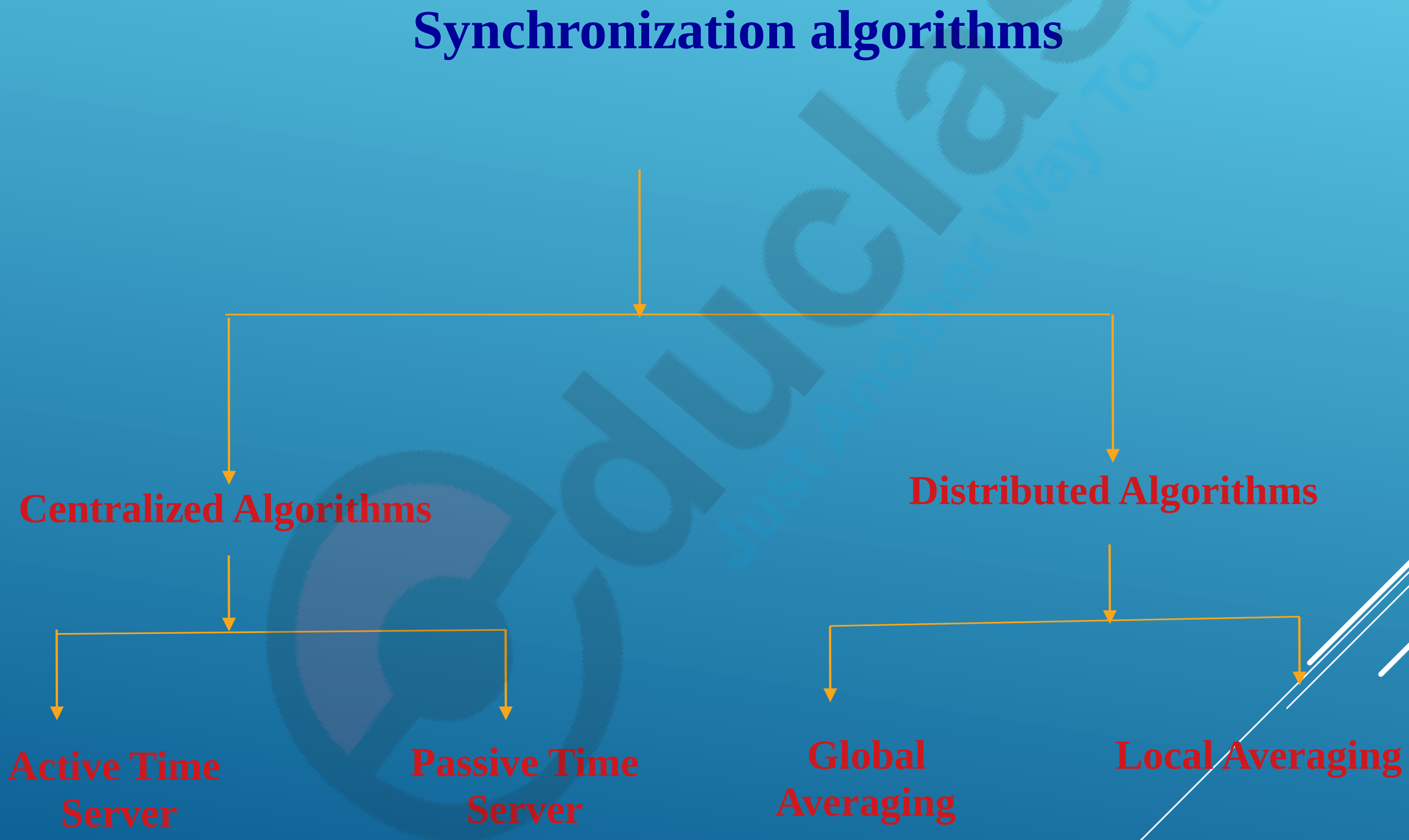
- We have seen that no two clocks can be perfectly synchronized
- In reality, two clocks are said to be synchronized, if the difference in the time value is less than a Specified constant.
- The difference in time values of two clocks is called *clock skew*
- Clock synchronization requires each node to read the other nodes' clock values
- The actual method used to read other clocks differs from one algorithm to another

Issues in Clock synchronization (Cont'd)

- *Errors occur mainly because of unpredictable communication delays during message passing used to deliver a clock signal or clock message from one node to another*
    - It is almost impossible to obtain the upper bound to delay.
- *Another important issue in clock synchronization is that time must not run backwards, this may cause serious problems such as repetition of certain operations that may be hazardous in many cases*
    - One way to do this is to make the interrupt routine in clock more intelligent
- For example, this can be done by increasing the timer holding counter in the clock oscillator to a new higher value so that interrupt is generated slower for given period of time, and that the synchronization is done smoothly but over a period of time

SYNCHRONIZATION ALGORITHMS

Synchronization algorithms



Centralized Algorithms

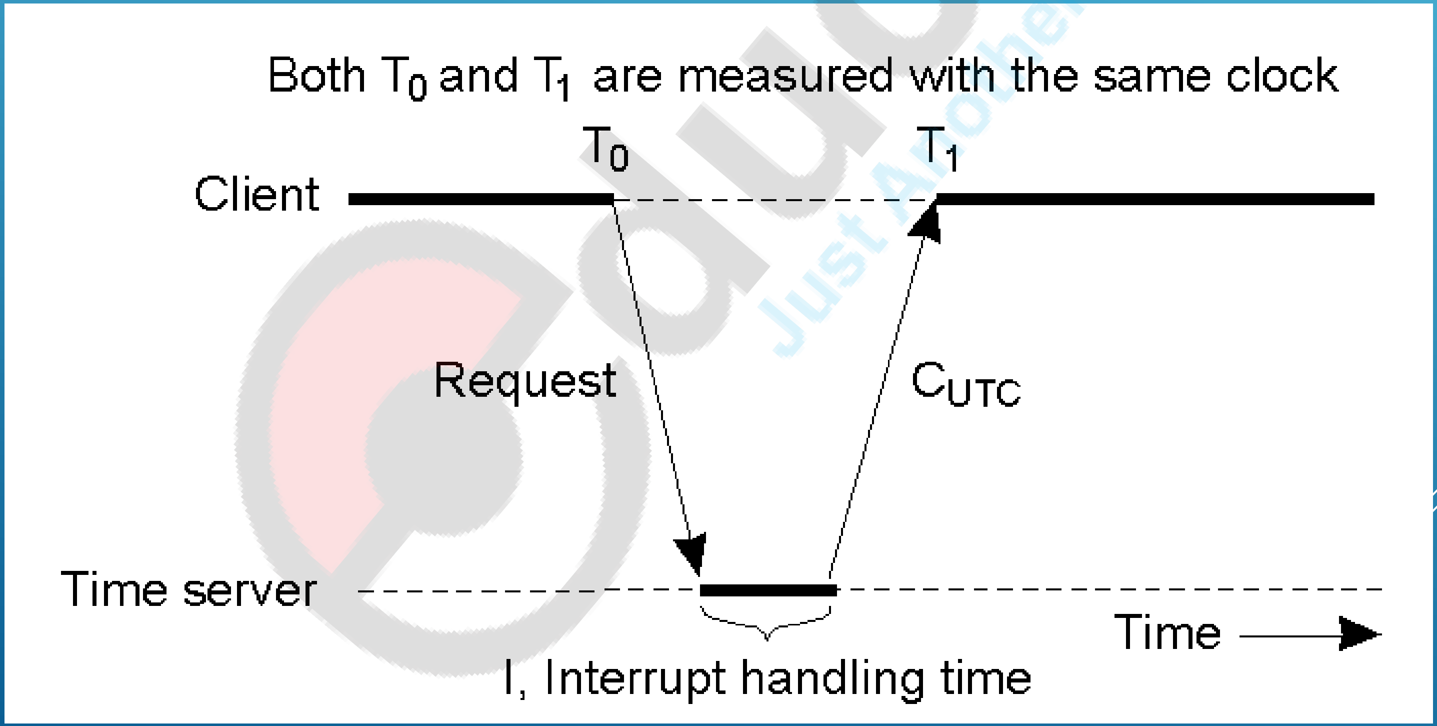
- In centralized algorithm one node has a real-time receiver
- This node is usually called *time server node*, and the clock on this node is regarded as correct and used as the reference time
- The goal of the algorithm is to keep the clocks of all other nodes synchronized with the clock time of the time server node
- Keep the clocks of all nodes synchronized with the clock time of the time server node, a real time receiver
  - *Passive time server centralized algorithm*
  - *Active time server centralized algorithm*
- Both these algorithms suffer from the drawbacks
  - Single point of failure
  - Poor scalability

Passive Time Server Centralized

- In this method each node periodically sends message ("time=?") to the time server
- Server quickly responds by sending ("time = T") where T is the current time on the time server (refer the slide on the next page)
- As a first approximation, when the sender receives the reply, it can adjust its clock to T
- This has two problem, one major, which we have discussed earlier that time can not run backwards and hence the change has to be introduced gradually by changing interrupt timer as discussed earlier
- The minor problem is that it takes non zero time for the server's reply to get back to the sender, it may be even large if there is a network problem

Cristian's Algorithm

- It is simple enough for the sender to record accurately the interval between sending the message T_0 and receiving reply T_1 measured using the same clock
- Getting the current time from a time server


Both T_0 and T_1 are measured with the same clock
T_0
T_1
Client
Request
C_{UTC}
Time server
Time
I, Interrupt handling time

Cristian's Algorithm (Cont'd)

- In the absence of any other information, the best estimate, after receiving message client adjusts time to
  - $T + (T_1 - T_0)/2$; message propagation time one way is $(T_1 - T_0)/2)$
  - This estimate can be further improved if it is known approximately how long it takes the server to handle the interrupt(request) and process the incoming message
  - $T + (T_1 - T_0 - I)/2$, I is time taken by time server to handle time request message
  - Cristian suggested making multiple measurements of $T_1 - T_0$
  - Those values which exceed a threshold are discarded and an average of the rest are used to estimate the correction factor
  - This will give good estimate of the average network congestion delays, however, their frequency itself will add to the network load

Active Time Server Centralized

- Time server node periodically broadcasts clock time (“time=T”)
- The other nodes receive the broadcast message and use the clock time in the message for correcting their own clocks
- Each node has a prior knowledge of the approximate time for the propagation (T_a)
- Client adjusts time to $T + T_a$
- *Drawbacks*
  - It is not fault tolerant in case message reaches too late at a node, its clock will be adjusted to wrong value
  - Requires broadcast facility to be supported by the network
- Drawbacks overcome by *Berkeley algorithm*

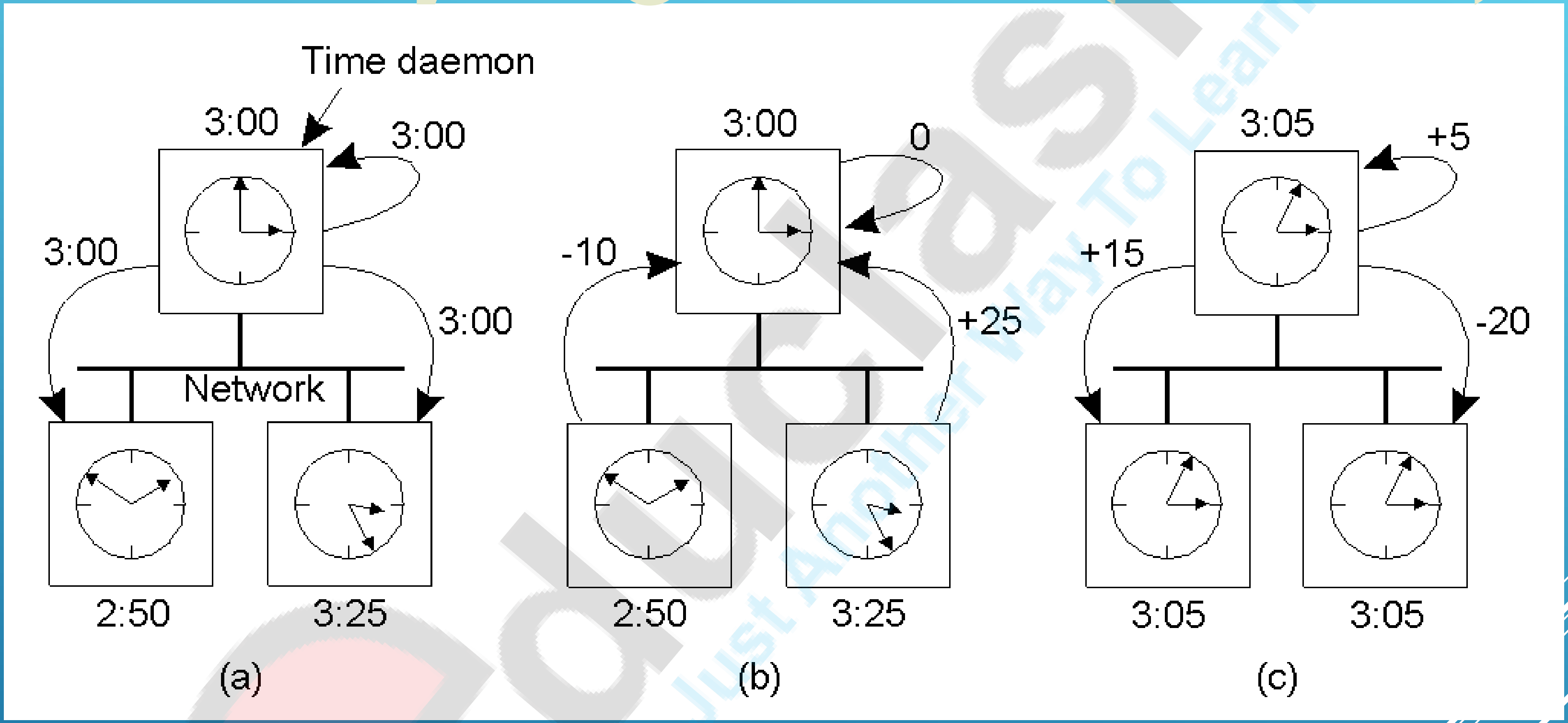
Berkeley Algorithm

- It is used for *internal clock synchronization* of a group
- The time *server (actually a time daemon) is active, polling every machine from time to time to ask what time it is there* ("time=?")
- Each computer in the group sends its clock value to the server
- Server has prior knowledge of propagation time from different node to server
- Based on this knowledge, it first readjusts the clock values of the reply messages
- It then takes fault tolerant average of the clock values of all computers (including its own)

Berkeley Algorithm (Cont'd)

- To take this fault tolerant average, the time server chooses a subset of all clock values that do not differ from one another by more than a specified amount, and the average is taken only from the clock values in this subset
- This approach eliminates readings from unreliable clocks, whose clock values could have significant adverse effect if an ordinary average is taken
- The calculated average is the current time to which all the clocks should be readjusted
- The time server readjusts its own clock to this value
- However, instead of sending the calculated time back to the other computers, the time server sends the amount by which each individual computer's clock requires adjustment

Berkeley Algorithm (Cont'd)



1. The time daemon asks all the other machines for their clock values
2. The machines answer
3. The time daemon tells everyone how to adjust their clock

Distributed Algorithms

- Remember that external synchronization also results in internal synchronization
- i.e., if each node's clock is independently synchronized with real time, all the clocks in system will remain mutually synchronized
- Hence a simple solution is to equip each node of the system with real time receiver so that each node's clock is independently synchronized with real time
- Separate internal synchronization is not required in this approach
- However in reality, due to the inherent inaccuracy in the real time clocks, different clocks produce different time
- Hence internal synchronization is performed for better accuracy

Distributed Algorithms (Cont'd)

- *Global averaging distributed algorithms*
- The clock process at each node broadcasts its local time in the form of special *"resync"* message when its local time equals T_0+iR for some integer i, where T_0 is a fixed time in the past agreed upon by all nodes and R is the system parameter that depends on such factors like total number of nodes in the system, max allowable drift rate and so on
- All broadcasts do not happen simultaneously due to difference in local clocks running at slightly different rates
- Broadcasting node waits for time T, where T is the parameter to be determined by the algorithm and during which it *collects "resync" messages by other nodes* & records time of receipt according to its own clock
- At the end of waiting time, it estimates the skew of its clock with respect to other nodes on the basis of times at which it received *"resync"* messages

Global Averaging Distributed Algorithms

- Calculate *fault tolerant average of estimated skews & use it to correct its own local clock* before restart of next "resync" interval
- The *global averaging algorithms* differ mainly in the manner in which the fault-tolerant average of the estimated skews is calculated
- Two commonly used algorithms are:
  - The simplest algorithm is to take the average of the estimated skews and use it as the correction of the local clock
  - However to limit the effect of faulty clocks on the average value, the estimated skews greater than the threshold are set to zero before computing the average of the estimated skews
  - In another algorithm each node limits the impact of faulty clocks by discarding m highest and m lowest estimates skews and then calculating average of the remaining skews
  - The value of m is based on total number of clocks (nodes)

Localized Averaging Distributed Algorithms

- Global averaging algorithms do not scale well because they require the network to support broadcast facility and also because large amount message traffic generated
- Hence they are suitable for small networks, especially those that have fully connected topology (in which each node has direct communication link to every other node)
- The localized averaging algorithms attempt to overcome these drawbacks of the global averaging algorithms
- In this approach nodes of a DS are logically arranged in some kind of pattern, such as ring or a grid
- Periodically, *each node exchanges its clock time with its neighbors* in the ring, grid etc
- It then sets its clock time to the average of its own clock time and the clock time of its neighbors

Localized Averaging Distributed Algorithms

- Two popular services for synchronizing clocks and for providing timing information over a wide variety of inter connected networks are the Distributed Time Service (DTS) and Network Time Protocol (NTP)
- NTP is used in Internet for clock synchronization

Numerical

- A distributed system has 3 nodes N1, N2 and N3 each having its own clock. The clocks at N1, N2 and N3 tick 495, 500 and 505 times per millisecond. The system uses external synchronization mechanism in which all nodes receive real time every 20 seconds from external file source and readjust their clocks. What is the maximum clock skew that will occur.

No. of clock ticks in 1 sec
N1 – 495,000
N2 – 500,000
N3 – 505,000
Maximum skew in 1 sec = 10,000 ticks(505,000-495,000)
Thus, skew in 20 sec = 2,00,000
Assuming the average time is 500,000 ticks in 1 sec.
Then 2,00,000 ticks leads to skew of = 2,00,000/500,000 = 0.4 sec

Numerical

- A distributed system has 3 nodes N1, N2 and N3 each having its own clock. The clocks at N1, N2 and N3 tick 600, 750 and 820 times per millisecond. The system uses external synchronization mechanism in which all nodes receive real time every 30 seconds from external file source and readjust their clocks. What is the maximum clock skew that will occur.

No. of clock ticks in I sec
N1 – 600,000
N2 – 750,000
N3 – 820,000
Maximum skew in 1 sec = 220,000 ticks(820,000-600,000)
Thus, skew in 30 sec = 6600000
Assuming the average time is 723333 approx ticks in 1 sec.
Then skew = 6600000/723333 = 9.12 sec approx

EVENT ORDERING

Event Ordering

- Lamport Observed that, for a certain class of algorithms, it is the internal consistency of the clocks that matters, not whether they are particularly close to the real time
- It is sufficient to ensure that all events that occur in a DS be totally ordered in a manner that is consistent with an observed behavior
- If two processes do not interact, it is *not necessary to keep their clocks synchronized* but rather that they *agree on the order in which events occur*
- To synchronize logical clocks, Lamport defined a new relation called *happened before* and introduced the concept of logical clocks for ordering of events based on *happened before* relation

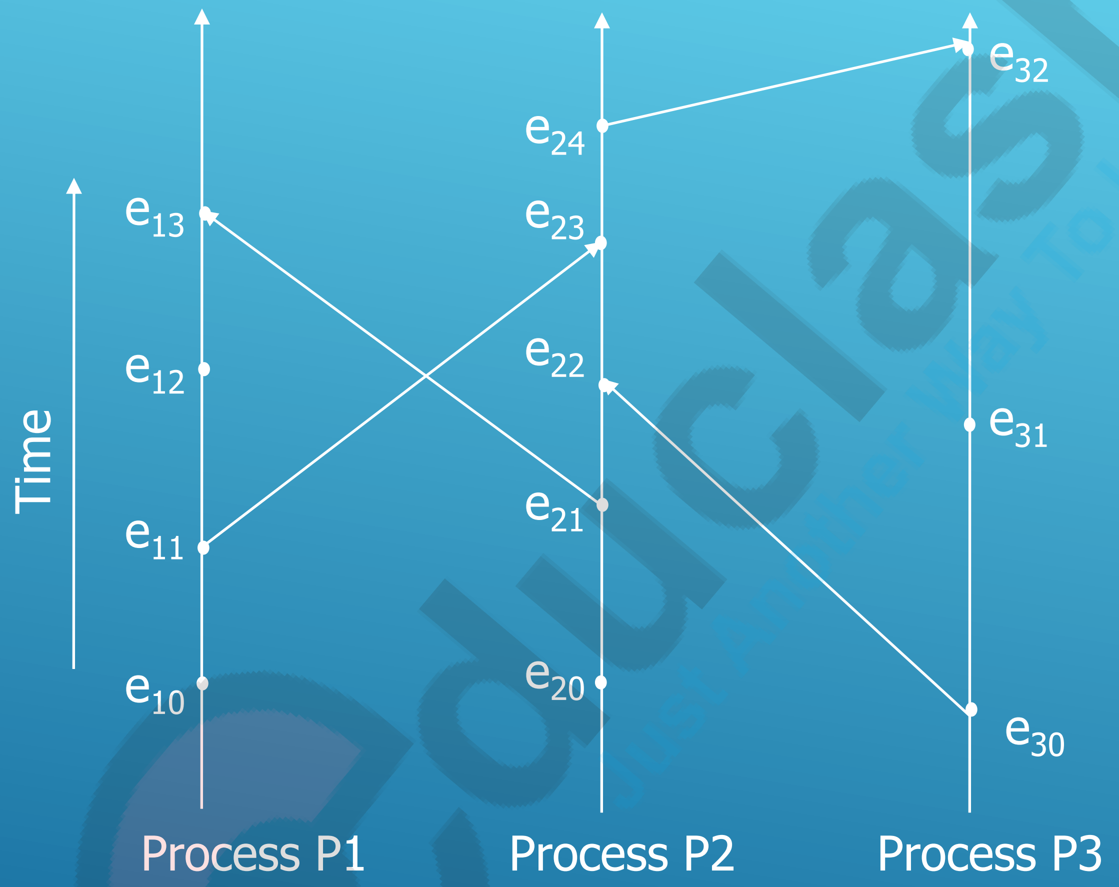
Happened Before Relation

- The happened-before relation on a set of events satisfy the following conditions:
  - 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 and this condition holds from the law of causality because a receiver can not receive until the sender has sent it
  - If a → b and b → c, then a → c *(transitive relation)*
- Note that in a physically meaningful system an event cannot occur before itself, i.e., a → a not true for any event a

Happened Before Relation (Cont'd)

- *Concurrent Events* - events a and b are concurrent (a||b) if neither $a \rightarrow b$ nor $b \rightarrow a$ is true; i.e., they are not related by the happened- before relation
- i.e., The two events are concurrent if neither can causally affect the other
- Because of this reason happened before relation is some times referred to as the relationship of causal ordering

Space-time Diagram for three process



In this diagram vertical line denotes process, each dot on the vertical line denote an event in the corresponding process and line denotes a message transfer from one process to another in the direction of the arrow

Space-time Diagram for three process

- A space-time diagram (fig. on previous slide) is used to illustrate the concepts of happened-before relation and concurrent events
- For two events a and b, $a \rightarrow b$ is true if and only if, there exists a path from a to b by moving forward in time along the process and message lines in the direction of the arrows

Causally ordered events

$(e_{10} \rightarrow e_{11})$, $(e_{20} \rightarrow e_{24})$, $(e_{11} \rightarrow e_{23})$, $(e_{21} \rightarrow e_{13})$

$(e_{30} \rightarrow e_{24})$ (since $e_{30} \rightarrow e_{22}$ & $e_{22} \rightarrow e_{24}$)

$(e_{11} \rightarrow e_{32})$ (since $e_{11} \rightarrow e_{23}$, $e_{23} \rightarrow e_{24}$ & $e_{24} \rightarrow e_{32}$)

- Two events a and b are concurrent if and only if, no path exists either from a to b or from b to a

Space-time Diagram for three process

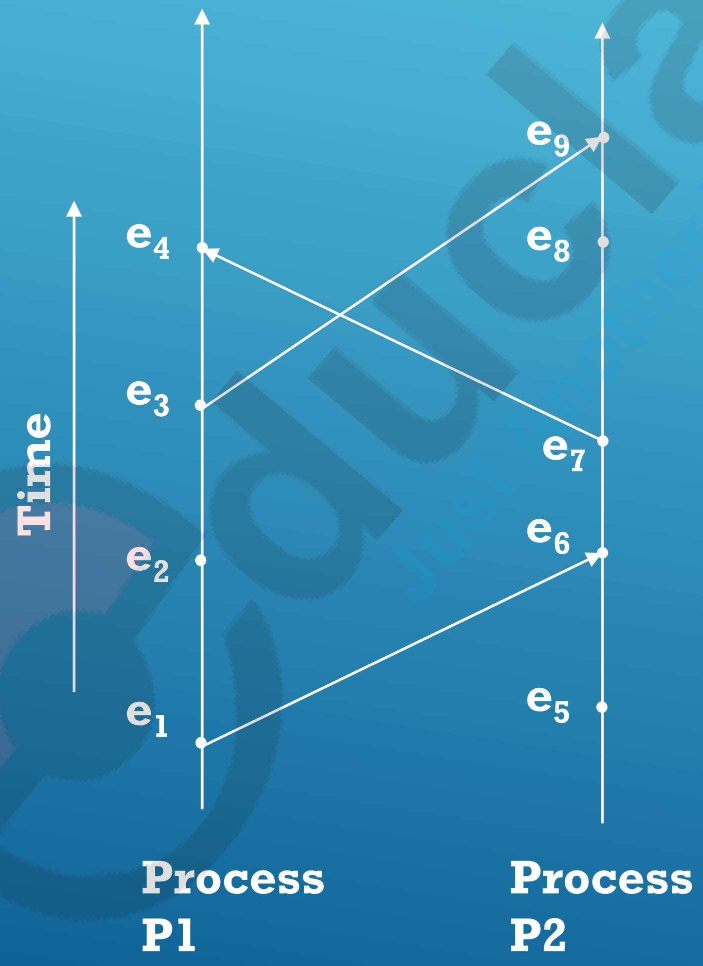
Therefore, the *Concurrent events* in the example are

$(e_{12}, e_{20}), (e_{21},e_{30}), (e_{10},e_{30}), (e_{12},e_{32}), (e_{13},e_{22})$

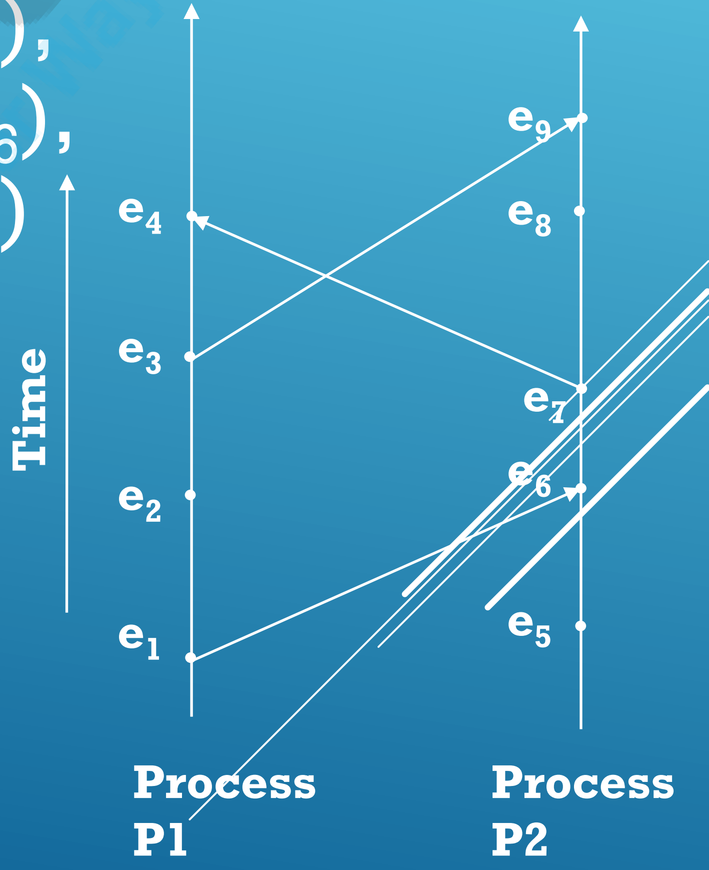
- What we need is a way of measuring time such that for every event a, we can assign it a time value C(a) on which consistency can be maintained.

Question

- List all pairs of concurrent events according to happened before relation



- **Causally ordered events** –$(e_1 \rightarrow e_2)$, $(e_2 \rightarrow e_3)$, $(e_3 \rightarrow e_4)$, $(e_5 \rightarrow e_6)$, $(e_6 \rightarrow e_7)$, $(e_7 \rightarrow e_8)$, $(e_8 \rightarrow e_9)$, $(e_1 \rightarrow e_6)$, $(e_3 \rightarrow e_9)$, $(e_7 \rightarrow e_4)$
- **Concurrent events** - (e_1,e_5), (e_2,e_5), (e_3,e_5), (e_4,e_5), (e_2,e_6), (e_2,e_7), (e_2,e_8), (e_2,e_9), (e_3,e_5), (e_3,e_6), (e_3,e_7), (e_3,e_8), (e_8,e_4), (e_4,e_9)



Logical Clocks

- To determine that an event *a* happened before an event *b*, either a common clock or a set of perfectly synchronized clocks are needed
- We know that neither of these are available in a distributed system
- Lamport provided a solution to this problem using logical clock concept
- The logical clocks concept is a way to associate a timestamp (which may be simply a number independent of any clock time) with each system event, so that events that are related to each other by the happened-before relation (directly or indirectly) can be properly ordered in that sequence

Logical Clocks (Cont'd)

- Actually the clocks may be implemented by a set of counters with out any timing mechanism
- The logical clocks of a system can be considered to be correct if the events of the system that are related to each other by the happened-before relation can be properly ordered using these clocks
- Hence, Timestamps assigned to events by logical clocks must satisfy the following clock condition:
  For any two events a and b, *if* $a \rightarrow b$, *then* $C(a) < C(b)$

Implementation of Logical Clocks

- From the definition of happened-before relation, following conditions must hold:

C$_1$: if a and b are two events within the same process P$_i$ and a occurs before b, then $C_i(a) < C_i(b)$

C2: If *a is the sending of a message* by process P$_i$ and *b is the receipt* of that message by process P$_j$ then $Ci(a) < Cj(b)$

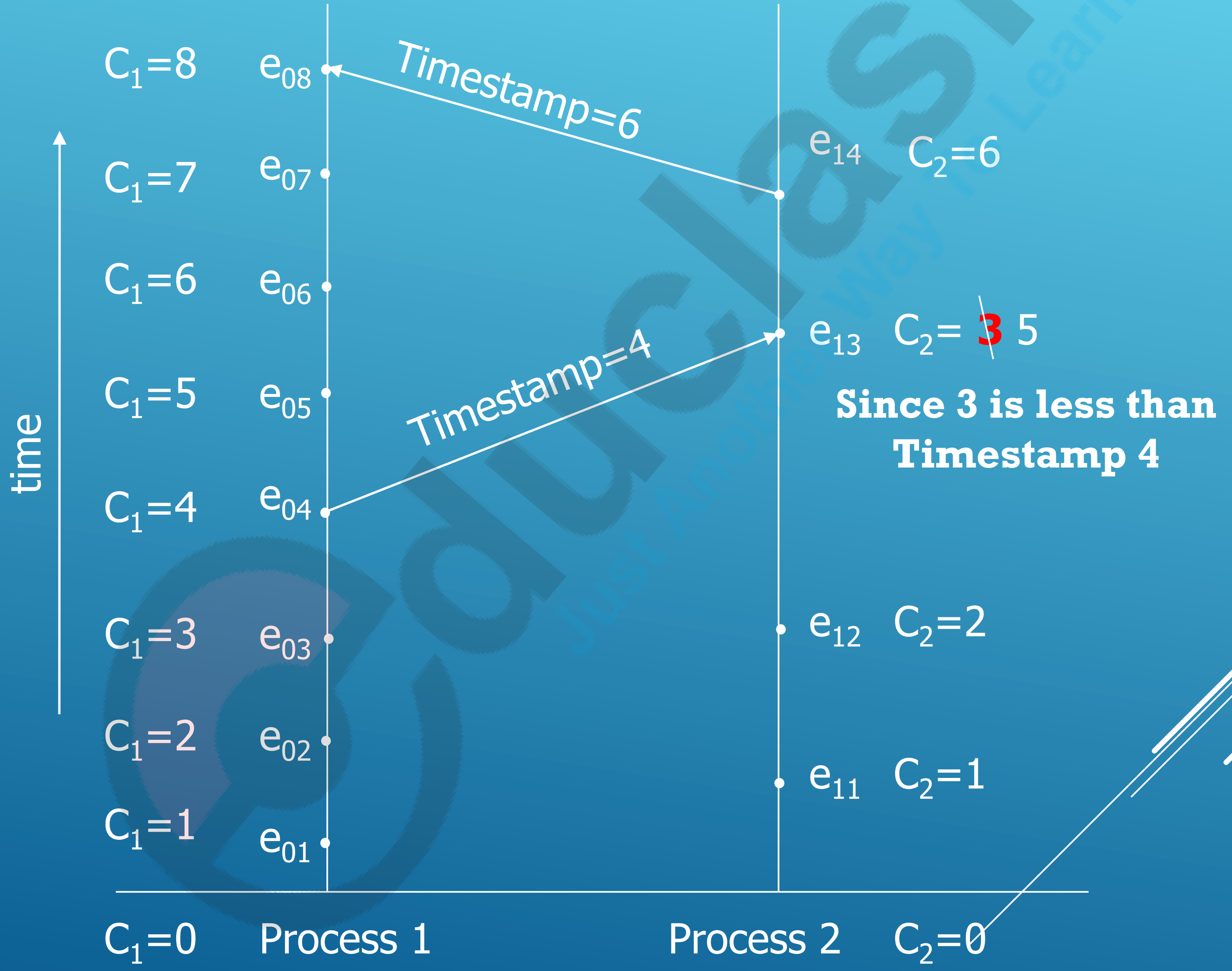
- In addition to these conditions, which are necessary to satisfy the clock condition the following condition is necessary for the correct functioning of the system

  C3: A clock Ci associated with a process P$_i$ must always go forward, never backward

Implementation of Logical Clock using Counters

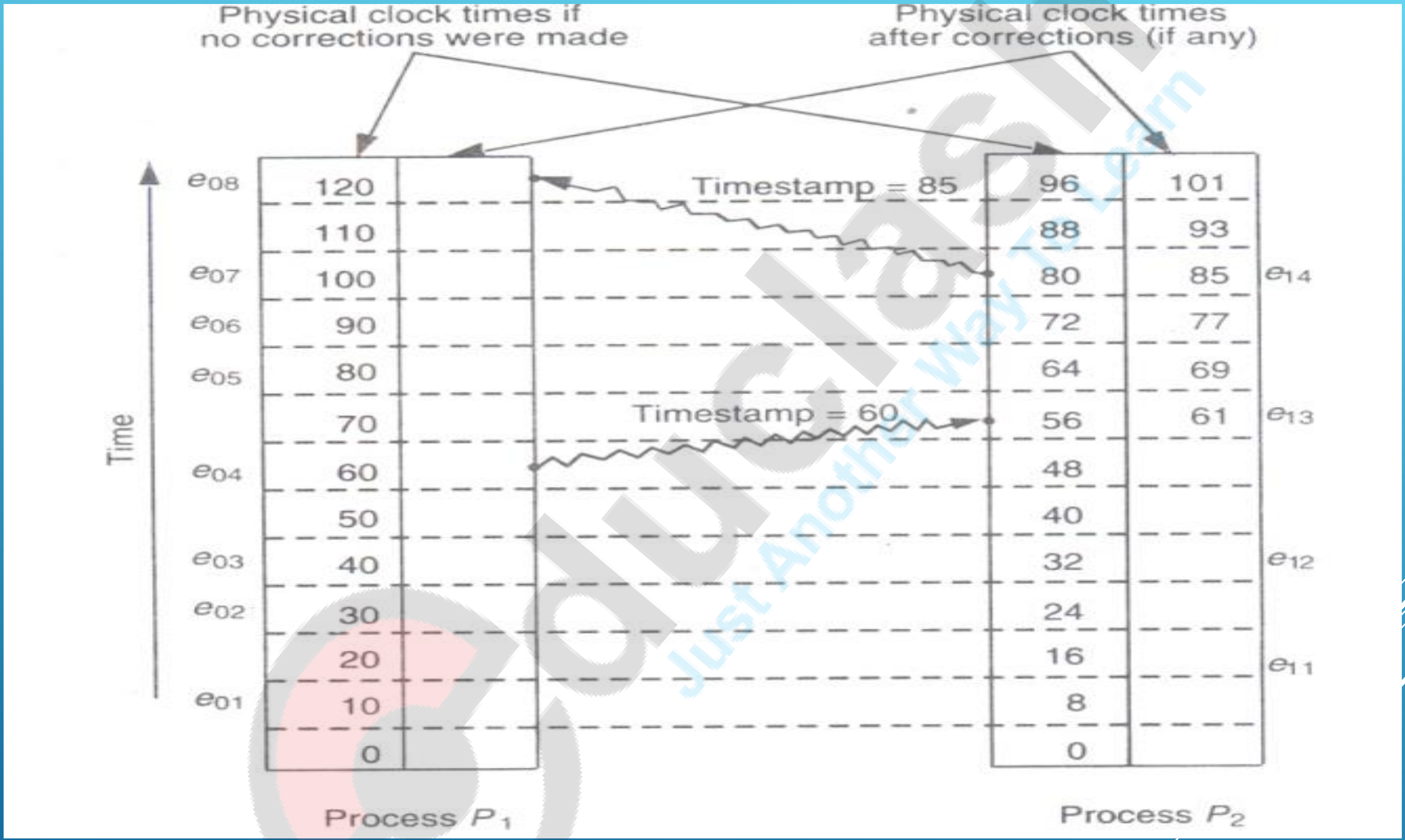
- As shown in the fig. on next slide, two processes P_1 and P_2 each have counters C_1 and C_2 respectively
- These counters act as logical clocks
- Counters initialized to zero & incremented by 1 whenever an event occurs in that process
- On sending of a message, the process includes the incremented value of the counter in the message

Implementation of Logical Clock using Counters



Since 3 is less than Timestamp 4

Using Physical Clocks (Cont'd)



1. Two processes each with its clock. They run at different rates
2. Lamport's algorithm Corrects the clock

GLOBAL STATE

Global State

- Sometimes, it is necessary to collect the current status of a distributed computation, which is known as the global state(eg., to see whether a system is in deadlock or to apply checkpoints).
- The global state of a DS actually consists of local state of each process, together with the messages which are in transit.
- A local state may consist of only those records which forms part of database excluding temporary records.
- An effective way of recording global state is distributed snapshot(reflects consistent state of a system).
- Snapshot should not record inconsistent messages, such as, recording of message receipt but not the corresponding message sending.

MUTUAL EXCLUSION

Mutual Exclusion

- There are several resources in a system that must not be used simultaneously by multiple processes, if the program operation is to be correct
- For example a file may not be updated by multiple processes
- Hence, exclusive access to such shared resource by a process must be ensured
- This exclusiveness of access is called mutual exclusion between processes
- The sections program that require exclusive access to shared resources are referred to as critical sections

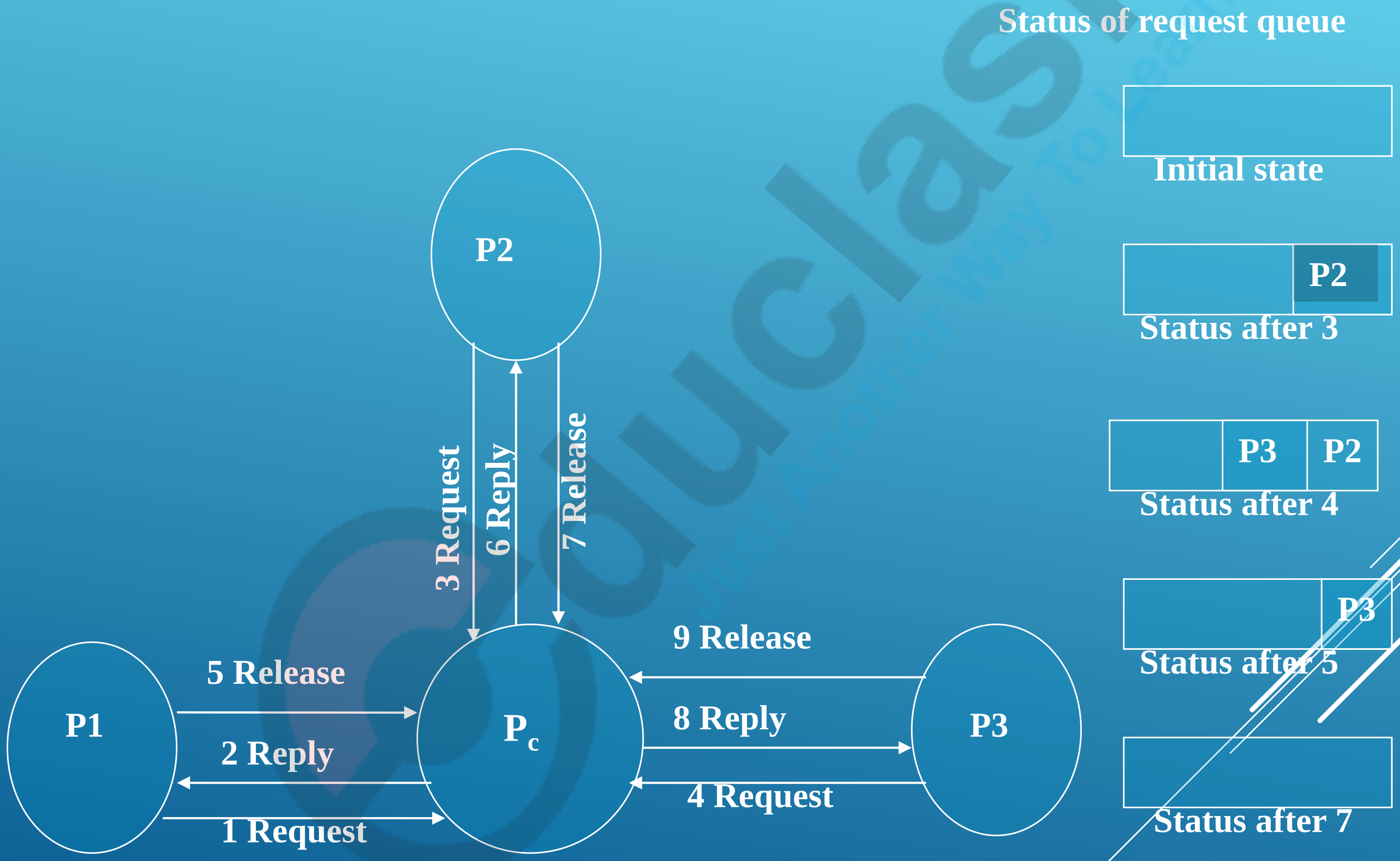
Mutual Exclusion (Cont'd)

- Conditions to be satisfied by mutual exclusion:
  - ***Mutual exclusion*** - Given a shared resource accessed by multiple concurrent processes, at any time *only one process should access the resource*
  - A process that has been granted the resource must release it before it can be granted to another process
  - ***No starvation*** – If every process that is granted resource eventually releases it, every request will be eventually granted
- In a single processor system mutual exclusion, critical regions are protected using semaphores, monitors and similar constructs
- There are three basic approaches discussed here to achieve mutual exclusion in distributed systems

Centralized Approach

- The most straightforward way to achieve mutual exclusion in a distributed system is to simulate uniprocessor system
- One process is elected as the coordinator
- Coordinator coordinates entry to critical section
- Every process wanting to enter critical section need seek permission from the coordinator
- If more than one process concurrently seek permission to enter the same critical section, the coordinator allows only one process to enter critical section in accordance with some scheduling algorithm and the remaining processes are put on a queue
- Ensures no starvation as uses first come, first served policy

Centralized Approach



Centralized Approach (Cont'd)

- On completion of the critical region activity, the process immediately releases the critical region and informs the coordinator accordingly (fig. on the prev. slide gives the sequence of operation for requests by 3 processes and a process coordinator)
- This algorithm ensures mutual exclusion because at a time only one process is allowed to enter a critical section
- The main advantage of the system is that it is simple to implement
- Requires only 3 messages per request for critical section – request, reply, release
- Suffers from usual drawbacks of centralized schemes, namely single point of failure & performance bottleneck
- Another problem is, the requesting process has no way of knowing whether it is in the queue or the coordinator has crashed

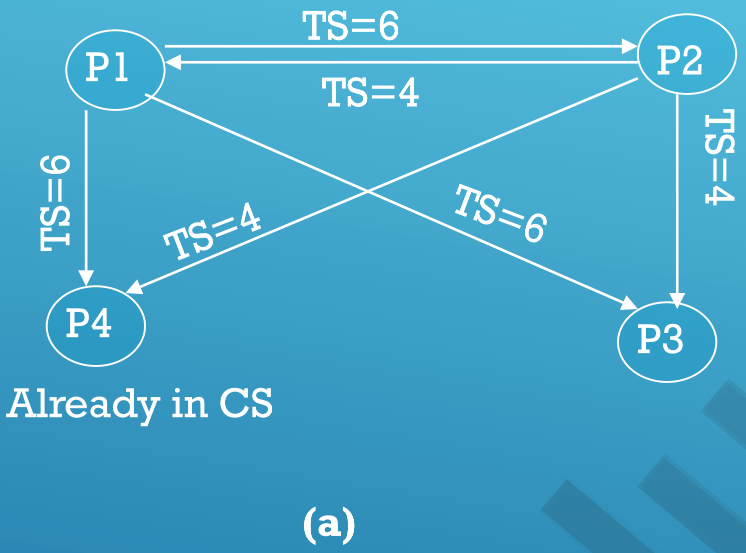
Distributed Approach

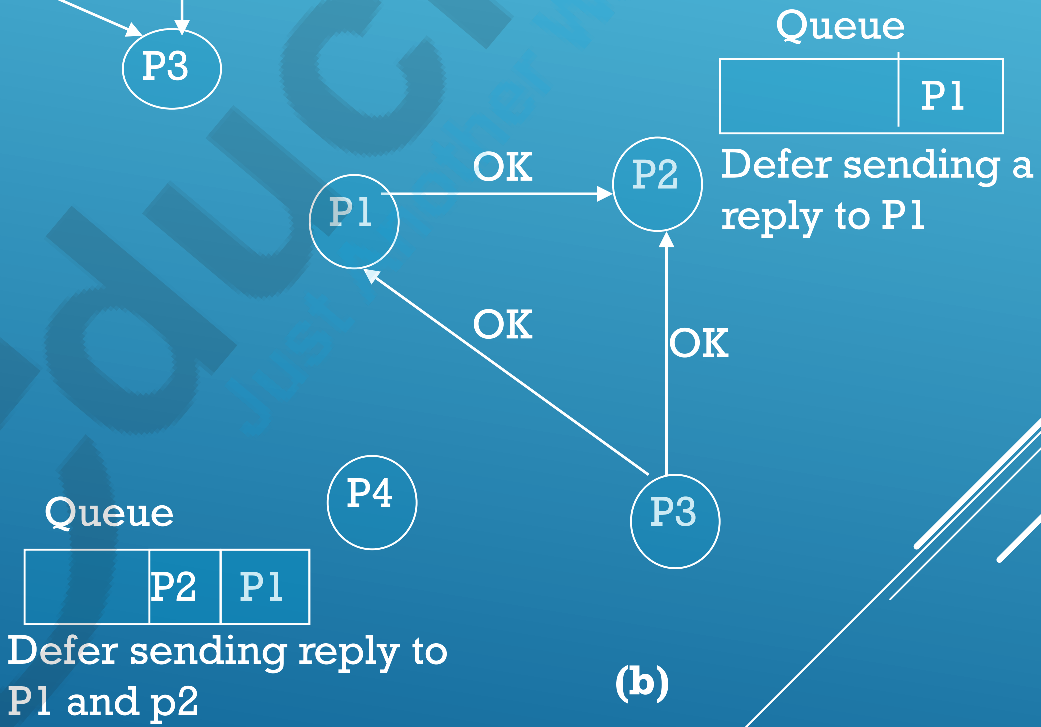
- In the distributed approach, the decision making for mutual exclusion is distributed across the entire system
- All processes that want to enter the same critical region, cooperate with each other before reaching a decision on which process will enter the critical region next
- Ricart & Agrawala's Algorithm
  - When a process wants to enter the CS, it sends a request message to all other process, and when it receives reply from all processes, then only it is allowed to enter the CS
  - The request message contains following information:
    - *Process identifier of the process*
    - *Name of critical section that the process wants to enter*
    - *Unique time stamp* generated by process for request message

Distributed Approach (Cont'd)

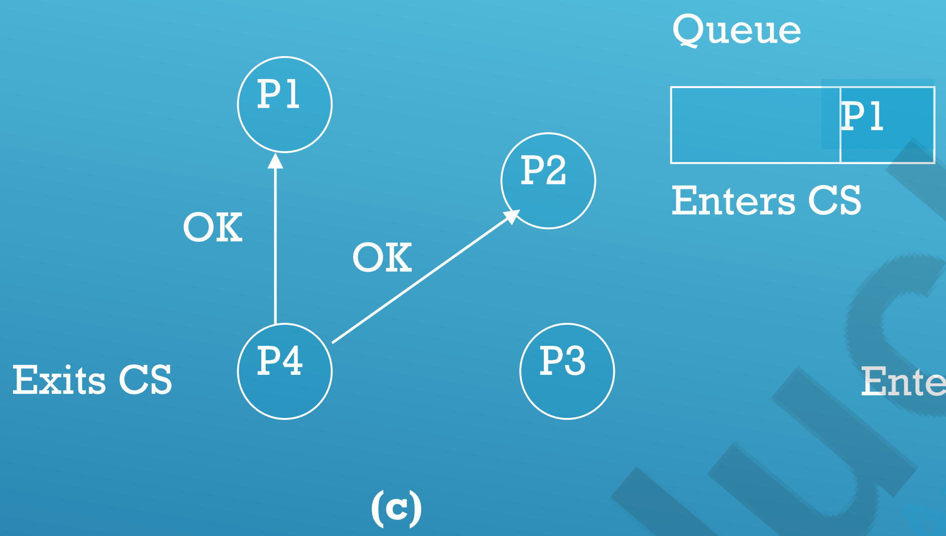
- The decision whether receiving process replies immediately to a request message or defers its reply is based on three rules:
  - If receiver process is itself currently executing in the critical section, then it queues the message and defers its reply
  - If receiver process is neither in the critical section nor is waiting for its turn to enter its critical section, it immediately sends a reply
  - If receiver process itself is waiting to enter critical section, then it compares its own request timestamp with the timestamp in request message
    - If its own request timestamp is greater than timestamp in request message, then it sends a reply immediately
    - Otherwise, the reply is deferred and queues receives request message

Distributed Approach (Cont'd)

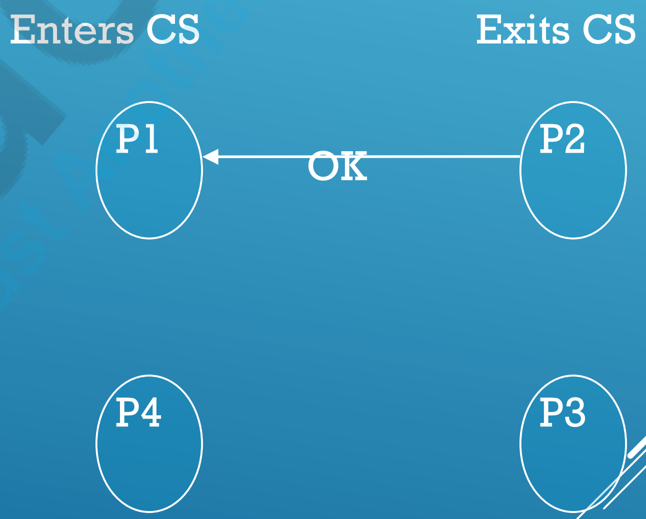




Distributed Approach (Cont'd)



(c)



(d)

Drawbacks

- No. of messages passed becomes the bottleneck with heavy communication traffic and the requirement that all processes must participate in a critical section entry request by any process
- All requesting processes have to wait indefinitely for the reply from the failed system
- A simple modification to the algorithm is instead of remaining silent by deferring the sending reply message in cases when permission can not be granted immediately, the receiver sends "permission denied" reply message to the requesting process and then sends an OK message when the permission can be granted
- If no reply is received from a system within timeout period, it is assumed to have crashed by the sending process
- The *processes need to know the identity of all other processes in the system*, which makes the dynamic addition and removal of processes more complex

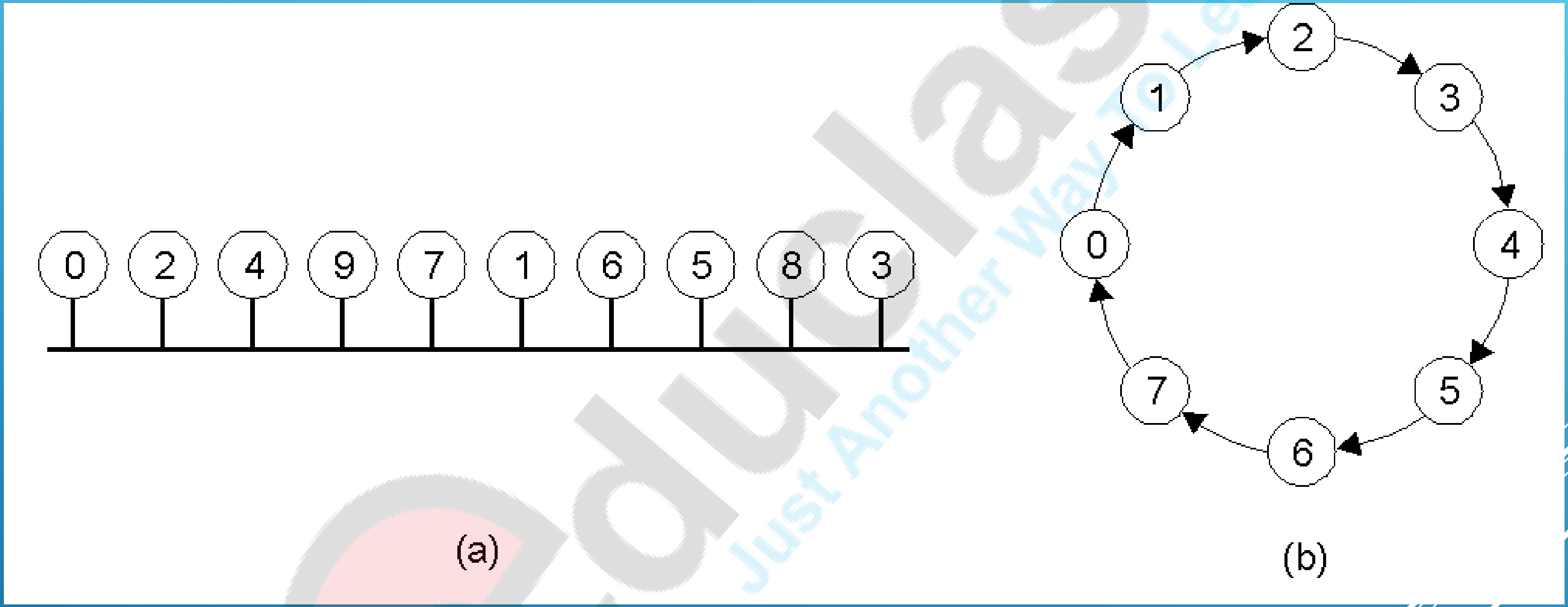
Token Passing Approach

- In this method, mutual exclusion is achieved by using a single token that is circulated among the processes in system in clockwise or anti-clockwise manner.
- Token is special kind of message that entitles its holder to enter a critical section
- For fairness, processes in a system are organized in logical ring as shown in the fig. in next slide
- Ring positions may be allocated in numerical order of network addresses or some other means
- It is passed from process k to k+1 in point-to-point messages
- When a process acquires a token from its neighbor, it checks if it wants to enter a critical region and acts as follows

Token Passing Approach (Cont'd)

- If it wants to enter a critical section, it keeps the token, enters the critical section, does all the work it needs to, and leaves the region
- After it has exited, it passes the token along the ring
- It is not permitted to enter a second critical region using the same token, it must wait until it gets the token again
- If it does not want to enter a critical section, it just passes the token along the region to its neighbor process
- Hence if none of the process is interested in entering a critical section, the token simply keeps circulating around the ring
- The correctness of this algorithm is easy to see, so only one process can actually be in a critical region

Token Passing Approach (Cont'd)



a) An unordered group of processes on a network.

b) A logical ring constructed in software.

Token Passing Approach (Cont'd)

- Since the token circulates among the processes in a well defined order, starvation can not occur
- Once a process decides it wants to enter the critical region, at worst it will have to wait for every other process to enter and leave the critical region
- Drawbacks:
  - A process failure in the system causes the logical ring to break
  - Then a new ring has to be established to ensure the continued circulation of the token among other processes
  - This requires detection of the failed process and dynamic reconfiguration of the logical ring
  - Detecting dead process is easy, when a neighbor tries to give it the token but fails

Token Passing Approach (Cont'd)

- That dead process can be removed from the group and passes the token to the process after it or next alive process in the sequence
- When a process becomes alive after recovery, it simply informs the neighbor previous to it in the ring so that it gets the token in the next round of circulation

- Lost token
  - If token is lost, a new token must be generated
  - Must have mechanism to detect & regenerate a lost token
  - Designate one of the processes in the ring as *monitor* process
  - Monitor periodically circulates "who has token" message on the ring
  - The message rotates round the ring from one process to another

Token Passing Approach (Cont'd)

- All the processes pass this message to their neighbor process, except the process that has the token
- This process, on receipt of the message, writes its identifier in a special field before passing it to its neighbor
- On return of the message, monitor checks process identifier field. If empty generate new token & passes it around the ring
- Multiple monitors can be used to take care of the failure of the monitor
- An election among them can decide who generates the lost token

ELECTION ALGORITHMS

Election Algorithm

- Several distributed algorithms require that there be a coordinator process in the system that performs some type of coordination activity needed for the smooth running of other processes in the system
- For example, Coordinator for centralized algorithm, monitor process in Token ring approach for mutual exclusion
- Since all other processes in the system has to interact with the coordinator, they all must agree on who the coordinator is
- If the coordinator process fails for whatever reason, a new coordinator process must be selected to take up the job of the failed coordinator
- Election algorithms are meant for *electing a coordinator* process from among the currently running processes in such a manner that at any instance of time there is a single coordinator for all processes in the system

Election Algorithm (Cont'd)

- One of the parameters commonly selected for election algorithm is the priority number of the process
- Election algorithms based on following assumptions:
  - *Each process has unique priority number*
  - Whenever an election is held, the *process having highest priority* among currently active processes is elected as the coordinator
  - On recovery, *a failed process can take appropriate actions to rejoin* the set of active processes
- Therefore, whenever initiated, an election algorithm basically finds out which of the currently active processes has the highest priority number and informs this to all other active processes

Bully Algorithm

- This algorithm was proposed by Garcia – Molina in 1982
- It assumes that every process knows priority of every other process in the system
- But what the processes do not know is which ones are currently up and which ones are currently down
- When a process P_i sends a request message to the coordinator and does not receive a reply within a fixed time period, it assume that the coordinator has failed
- $P_{i,}$ then *initiates an election by sending an election message to every process having priority higher than itself*

Bully Algorithm (Cont'd)

- If P_i does not receive any response to its election message within a fixed timeout period, it assumes that among the currently active processes it is the one with the highest priority number
- Hence, it takes up job of the coordinator & sends message (*let us call it the coordinator message*) to all processes with lower priority number than itself, informing that from now on it being new coordinator
- When a process P_j receives an election message from a process having a lower priority than itself, it sends a response message (*say alive message*) to the sender informing that it is alive and will take over the election activity
- The process P_i does not take any further action and just waits to receive the final result (coordinator message) from the new coordinator of final result of the election it initiated

Bully Algorithm (Cont'd)

- Now P_j holds an election, if it is not already holding one by repeating the above process
- In this way, the election activity gradually move on to the process that has the highest priority number among the currently active processes and eventually wins the election and becomes new coordinator
- As part of the recovery action, this method required that a failed process (say P_k) must initiate an election on recovery
- If the current coordinator's priority no is higher than that of P_k then the current coordinator will win the election initiated by P_k and will continue to be the coordinator
- On the other hand if P_k's priority no is higher than that of current coordinator, it will not receive any response to its election message

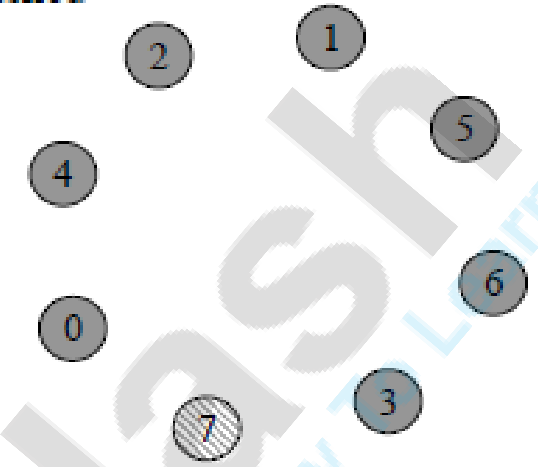
Bully Algorithm (Cont'd)

- So it winds up the election and takes over the coordinator's job from the current coordinator by informing all the processes with lower priority number that it is the new coordinator from now on
- Hence active processors with the highest priority always win the election
- Therefore the algorithm is called Bully Algorithm as the process with the higher priority always forces its way to become the coordinator
- It may also be noted here that if the processes having highest priority number recovers after a failure, it does not initiate an election because it knows from its list of priority numbers that all other processes have lower priority numbers than its own

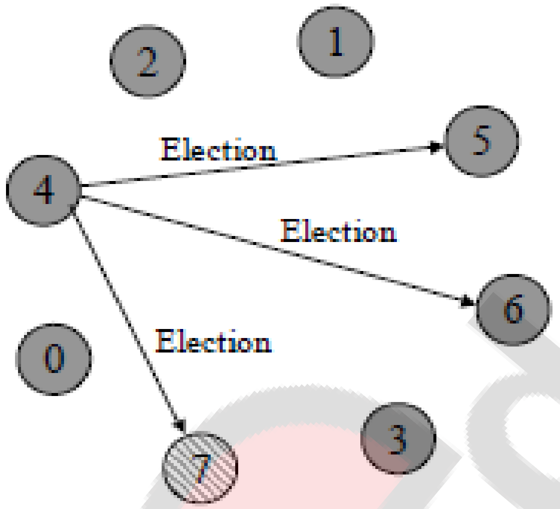
Bully Algorithm (Cont'd)

- Hence on recovery it just sends a coordinator message to all other processes and bullies the current coordinator into submission
- Let us new see the working of this algorithm with an example
  - The group consist of 8 processes numbered 0 to 7 with process 7 as the coordinator
  - Suppose process 7 just crashes
  - process 4 notices it first, so it sends an election message to all the processes higher than it, namely 5, 6 and 7
  - Processes 5 and 6 respond with OK, as shown in figure in the next slide
  - Process 5 and 6 hold election and 6 says OK and takes over election
- At this time process 6 knows that 7 is dead as it does not get response for its election message and it is the winner

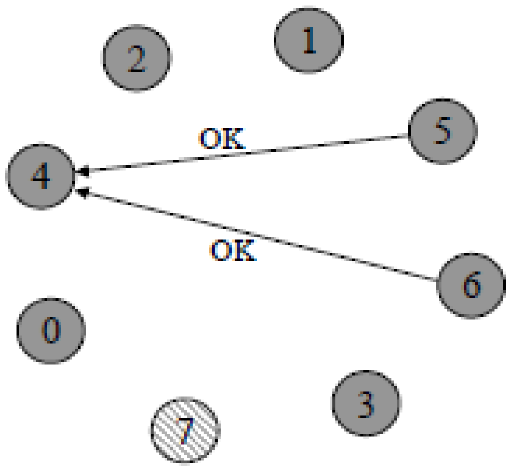
◆ Example: processes 0-7, 4 detects that 7 has crashed



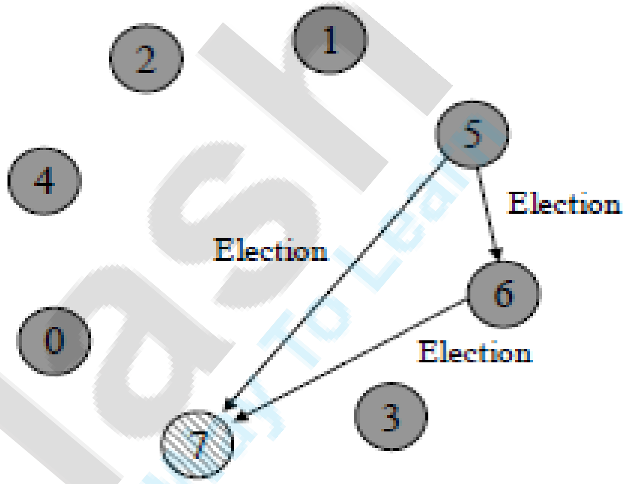
◆ Example: process 4 holds an election



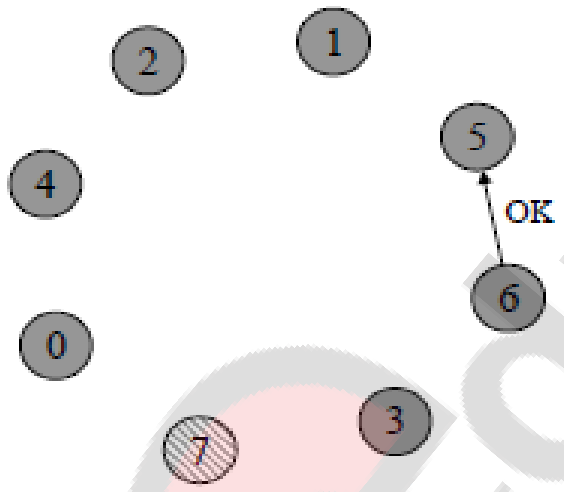
◆ Example: processes 5 and 6 respond with OK



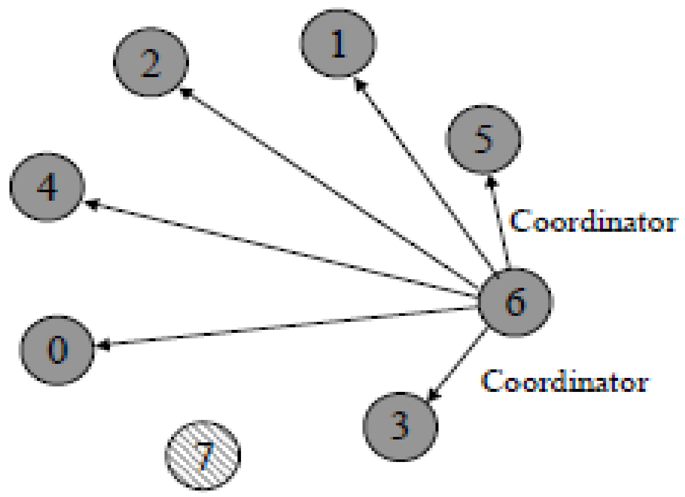
◆ Example: Processes 5 and 6 hold elections



◆ Example: process 6 sends OK



◆ Example: process 6 is the new Coordinator



Bully Algorithm (Cont'd)

- Suppose now process 7 recovers from the crash, and it knows that it is the process with the highest priority number, it sends a coordinator message to all processes and elects itself as the new coordinator

Ring Algorithm

- Another election algorithm is based on the use of ring concept
- It is assumed that all the processes in the system are physically or logically organized in a ring so that each process knows who its successor is.
- The ring is unidirectional in the sense that all the messages related to the election algorithm are always passed only in one direction (clockwise / anticlockwise)
- Every process in the system knows the structure of the ring, so that while trying to circulate a message over the ring, if the successor of the sender process is down, the sender can skip over the successor, or the one after that, until an active member is located

Ring Algorithm (Cont'd)

- When a process say P_i sends a request message to the current coordinator and does not receive a reply within the fixed timeout period, it assumes that the coordinator has crashed
- Hence, it initiates an election by sending an election message to its successor (actually to the first successor that is currently active)
- This message contains the priority number of the process P_i
- On receiving the election message, the successor appends its own priority number to message, making itself a candidate to be elected as coordinator and passes on the next active member in the ring
- In this manner the election message circulates the ring from one active process to another and eventually returns back to P_i
- It recognizes it as its own message by the first priority as its priority

Ring Algorithm (Cont'd)

- When the process P_i receives its own election message, the message contains the list of priority numbers of all processes that are currently active
- Therefore from this list, it elects the process having the highest priority number as the new coordinator
- It then circulates a coordinator message over the ring to inform all the other active process who the new coordinator is
- When the coordinator message comes back to process P_i after completing its one round along the ring, it is removed by process P_i
- When the process P_j recovers from failure, it creates an enquiry message and sends it to its successor
- The message contains identity of process P_j

Ring Algorithm (Cont'd)

- If the successor is not the coordinator, it simply forwards the enquiry message to its own successor
- In this way, the enquiry message moves forward along the ring until it reaches the current coordinator
- On receiving enquiry message, the current coordinator sends a reply to process P_j informing that it is the current coordinator
- Note that in this algorithm two or more processes may almost simultaneously discover that the coordinator has crashed and then each one may circulate an election message over the ring
- Though results in a little waste of network bandwidth, it does not cause any problem because every process will receive the same list of active processes and all of them select the same coordinator

Comparison of election algorithms

- In the bully algorithm when the process having lowest priority detects coordinator failure and initiates an election in the system having total n processes, altogether n-2 elections are performed one after another for the initiated one, i.e., all the processes, except the active process with the high priority and the crashed process
- However in ring topology irrespective of which process detects failure of the coordinator and initiates an election, an election always requires 2(n-1) messages i.e., one round for election message and one round for coordinator message

Comparison of election algorithms

- In Bully algorithm the failed process has to initiate an election on recovery
- On the other hand in the ring algorithm, a failed process does not initiate an election on recovery, but simply searches for the current coordinator
- In conclusion ring algorithm is more efficient compared bully algorithm and easier to implement

UNIVERSITY QUESTIONS

- Explain 'Happened Before' relations.
- Name various clock synchronization algorithms used in details. Explain in detail.
- Explain the concept of logical clocks and their importance in distributed systems.
- A clock of a computer system must never run backward. Explain how this issue can be handled in an implementation of logical clocks concepts.
- Election Algorithms (Note)
- Explain bully and Ring Algorithms.
- External synchronization automatically leads to internal synchronization but converse is not true. Explain.