



**AALIM MUHAMMAD SALEGH COLLEGE OF ENGINEERING**  
APPROVED BY AICTE, NEW DELHI & AFFILIATED BY ANNA UNIVERSITY CHENNAI  
NAAC ACCREDITED INSTITUTION  
ANNA UNIVERSITY COUNSELLING CODE 1101



**INSTITUTION'S  
INNOVATION  
COUNCIL**  
(Ministry of HRD Initiative)



**DEPARTMENT OF INFORMATION TECHNOLOGY**

**R2017 - SEMESTER V**

**CS3551- DISTRIBUTED COMPUTING**

**UNIT III - DISTRIBUTED MUTEX**

**AND DEADLOCK**

**CLASS NOTES**

## **UNIT – III : Syllabus**

### **COURSE OBJECTIVES:**

- To describe distributed mutual exclusion and distributed deadlock detection techniques

### **COURSE OUTCOMES:**

Upon the completion of this course, the student will be able to

CO3: Use resource sharing techniques in distributed systems (K3)

### **SYLLABUS:**

#### **UNIT III      DISTRIBUTED MUTEX AND DEADLOCK**

**10**

Distributed Mutual exclusion Algorithms: Introduction – Preliminaries – Lamport’s algorithm – Ricart- Agrawala’s Algorithm — Token-Based Algorithms – Suzuki-Kasami’s Broadcast Algorithm; Deadlock Detection in Distributed Systems: Introduction – System Model – Preliminaries – Models of Deadlocks – Chandy-Misra-Haas Algorithm for the AND model and OR Model.

**TOTAL TOPICS: 10**

# **1. DISTRIBUTED MUTUAL EXCLUSION ALGORITHMS:**

## **INTRODUCTION**

### **1.1 Introduction to Mutual exclusion**

- **Mutual exclusion** is a concurrency control property which is introduced to prevent race conditions.
- It is the requirement that a process cannot enter its critical section while another concurrent process is currently present or executing in its critical section i.e only one process is allowed to execute the critical section at any given instance of time.

#### **Definition: Mutual exclusion or Mutex**

- Mutual exclusion ensures that concurrent access of processes to a shared resource or data is serialized, that is, executed in a mutually exclusive manner.
- Mutual exclusion in a distributed system states that only one process is allowed to execute the critical section (CS) at any given time.
- A mutual exclusion (mutex) is a program object that prevents multiple threads from accessing the same shared resource simultaneously.
- A shared resource in this context is a code element with a critical section, the part of the code that should not be executed by more than one thread at a time.
- Mutex is a unit of code that averts contemporaneous access to shared resources.
- Mutual exclusion is concurrency control's property that is installed for the objective of averting race conditions.
- Examples of mutual exclusion are:  
locks, recursive locks, RW locks, semaphores, etc.

### **1.1.1 Examples of Mutual Exclusion**

There are many types of mutual exclusion, some of them are mentioned below :

**a) Locks :**

- It is a mechanism that applies restrictions on access to a resource when multiple threads of execution exist.

**b) Recursive lock :**

- It is a certain type of mutual exclusion (mutex) device that is locked several times by the very same process/thread, without making a deadlock.
- While trying to perform the "lock" operation on any mutex may fail or block when the mutex is already locked, while on a recursive mutex the operation will be a success only if the locking thread is the one that already holds the lock.

**c) Semaphore :**

- It is an abstract data type designed to control the way into a shared resource by multiple threads and prevents critical section problems in a concurrent system such as a multitasking operating system.
- They are a kind of synchronization primitive.

**d) Readers writer (RW) lock :**

- It is a synchronization primitive that works out reader-writer problems. It grants concurrent access to the read-only processes, and writing processes require exclusive access.
- This conveys that multiple threads can read the data in parallel however exclusive lock is required for writing or making changes in data. It can be used to manipulate access to a data structure inside the memory.

### **1.2 Why is Mutual Exclusion Required ?**

- An easy example of the importance of Mutual Exclusion can be envisioned by implementing a linked list of multiple items, considering the fourth and fifth need removal.
- The deletion of the node which sits between the other two nodes is done by modifying the previous node's next reference directing the succeeding node.
- Now, this situation is called a race condition.

- Race conditions can be prevented by mutual exclusion so that updates at the same time cannot happen to the very bit about the list.

### **1.3 Mutual exclusion in single computer system Vs. distributed system**

<b>Single computer system</b>	<b>Distributed System</b>
<ul style="list-style-type: none"> <li>• In single computer system, memory and other resources are shared between different processes.</li> </ul>	<ul style="list-style-type: none"> <li>• In Distributed systems, we neither have shared memory nor a common physical clock and there for we cannot solve mutual exclusion problem using shared variables.</li> </ul>
<ul style="list-style-type: none"> <li>• The status of shared resources and the status of users is easily available in the shared memory so with the help of shared variable</li> <li>• (For example: <u>Semaphores</u>) mutual exclusion problem can be easily solved.</li> </ul>	<ul style="list-style-type: none"> <li>• To eliminate the mutual exclusion problem in distributed system approach based on message passing is used.</li> <li>• A site in distributed system do not have complete information of state of the system due to lack of shared memory and a common physical clock.</li> </ul>

## **2. PREMLIMIMARIES**

### **2.1 Mutual exclusion in distributed system**

- Mutual exclusion in distributed system assumes that there is group agreement on how a resource or critical section is identified (e.g., a name or number) and that this identifier is passed as a parameter with any requests.
- We also assume that processes can be uniquely identified throughout the system (e.g., using a combination of machine ID and process ID).
- Mutual exclusion should be ensured in the middle of different processes when accessing shared resources.

- The process that is outside the critical section must not interfere with another for access to the critical section.
- **The goal is to get a lock on a resource**: permission to access the resource exclusively.
- When a process is finished using the resource, it releases the lock, allowing another process to get the lock and access the resource.

## **2.2 Components of Mutual exclusion Algorithm**

The requirements of mutual exclusion describe three major compartments:

- i) System model
- ii) Requirements that mutual exclusion algorithms
- iii) Metrics we use to measure the performance of mutual exclusion algorithms.

### **i) System model**

- The system consists of N sites, S1, S2, ..., SN. We assume that a single process is running on each site.
- The process at site Si is denoted by pi.
- A process wishing to enter the CS, requests all other or a subset of processes by sending REQUEST messages, and waits for appropriate replies before entering the CS. While waiting the process is not allowed to make further requests to enter the CS.

### **A site can be in one of the following three states:**

- a. Requesting the Critical Section.
  - b. Executing the Critical Section.
  - c. Neither requesting nor executing the CS (i.e., idle).
- In the 'requesting the CS' state, the site is blocked and can not make further requests for the CS. In the 'idle' state, the site is executing outside the CS.
  - In the token-based algorithms, a site can also be in a state where a site holding the token is executing outside the CS. Such state is referred to as the idle token state.

- At any instant, a site may have several pending requests for CS. A site queues up these requests and serves them one at a time.

**Note:**

- We assume that channels reliably deliver all messages, sites do not crash, and the network does not get partitioned.
- Some mutual exclusion algorithms are designed to handle such situations. Many algorithms use Lamport-style logical clocks to assign a timestamp to critical section requests.
- Timestamps are used to decide the priority of requests in case of a conflict.
- A general rule followed is that the smaller the timestamp of a request, the higher its priority to execute the CS.

**We use the following notations:**

- N denotes the number of processes or sites involved in invoking the critical section,
- T denotes the average Message Time Delay,
- E denotes the average critical section Execution Time.

**ii) Requirements of Mutual exclusion Algorithm**

Any viable mutual exclusion algorithm must satisfy three properties:

- i) **No Deadlock:** Freedom from deadlocks (desirable)
  - Two or more sites should not endlessly wait for any message that will never arrive.
- ii) **No Starvation:** Guarantee mutual exclusion (required)
  - Every site who wants to execute critical section should get an opportunity to execute it in finite time.
  - Any site should not wait indefinitely to execute critical section while other sites are repeatedly executing critical section
- iii) **Fairness:**
  - Each site should get a fair chance to execute critical section.

- Any request to execute critical section must be executed in the order they are made i.e Critical section execution requests should be executed in the order of their arrival in the system.

iv) **Fault Tolerance:**

- In case of failure, it should be able to recognize it by itself in order to continue functioning without any disruption.

**Some points are need to be taken in consideration to understand mutual exclusion fully**

:

1) It is an issue/problem which frequently arises when concurrent access to shared resources by several sites is involved.

- For example, directory management where updates and reads to a directory must be done atomically to ensure correctness.

2) It is a fundamental issue in the design of distributed systems.

3) Mutual exclusion for a single computer is not applicable for the shared resources since it involves resource distribution, transmission delays, and lack of global information.

**Necessary Conditions for Mutual Exclusion**

**There are four conditions applied to mutual exclusion, which are mentioned below :**

- Mutual exclusion should be ensured in the middle of different processes when accessing shared resources. There must not be two processes within their critical sections at any time.
- Assumptions should not be made as to the respective speed of the unstable processes.
- The process that is outside the critical section must not interfere with another for access to the critical section.
- When multiple processes access its critical section, they must be allowed access in a finite time, i.e. they should never be kept waiting in a loop that has no limits.

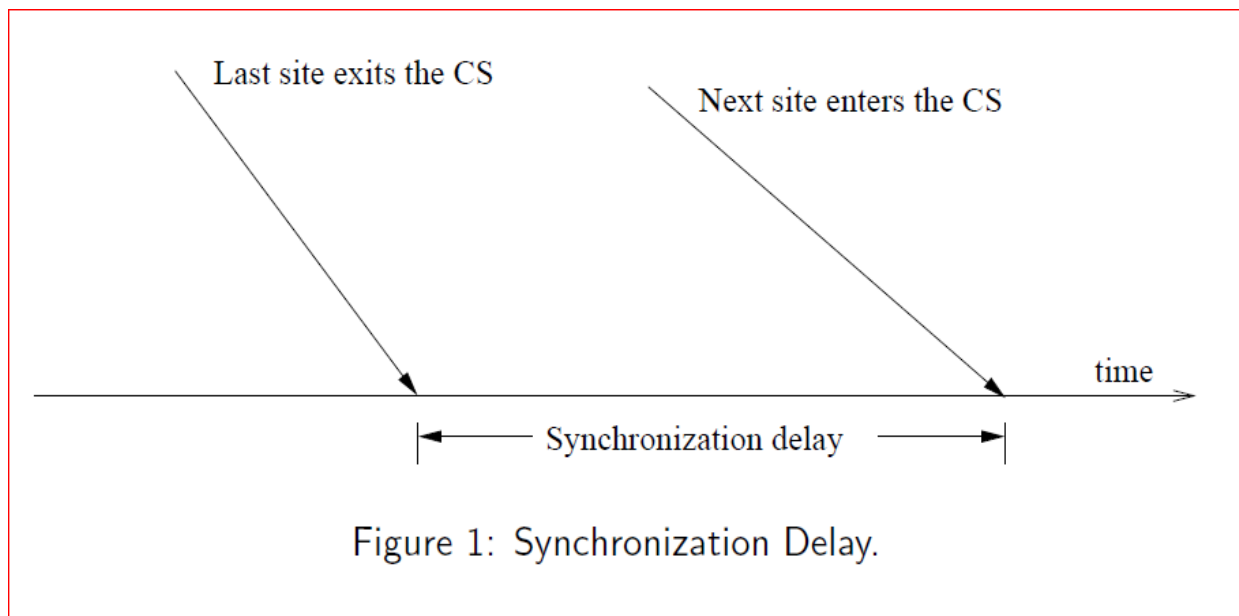
## **2.3 Performance measurements of distributed Mutual Exclusion**

### **Metrics we use to measure the performance of mutual exclusion algorithms**

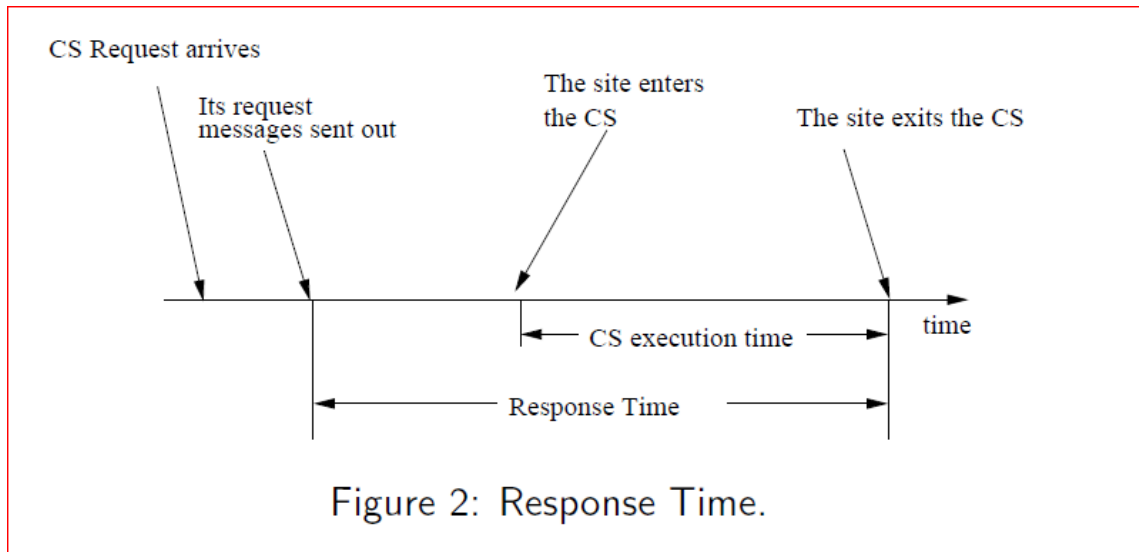
The performance of mutual exclusion algorithms is generally measured by the following four metrics:

a. **Message complexity:** It is the number of messages that are required per CS execution by a site.

b. **Synchronization delay:** After a site leaves the CS, it is the time required and before the next site enters the CS (see Figure 1).



c. **Response time:** It is the time interval a request waits for its CS execution to be over after its request messages have been sent out (see Figure 9.2).



**d. System throughput:** It is the rate at which the system executes requests for the CS. If SD is the synchronization delay and E is the average critical section execution time, then the throughput is given by the following equation:

$$\text{System Throughput} = 1 / (\text{SD} + \text{E})$$

Generally, the value of a performance metric fluctuates statistically from request to request and we generally consider the average value of such a metric.

**Low and High Load Performance:** The load is determined by the arrival rate of CS execution requests. Two special loading conditions, viz., “**low load**” and “**high load**”.

- Under **low load** conditions, there is seldom more than one request for the critical section present in the system simultaneously.

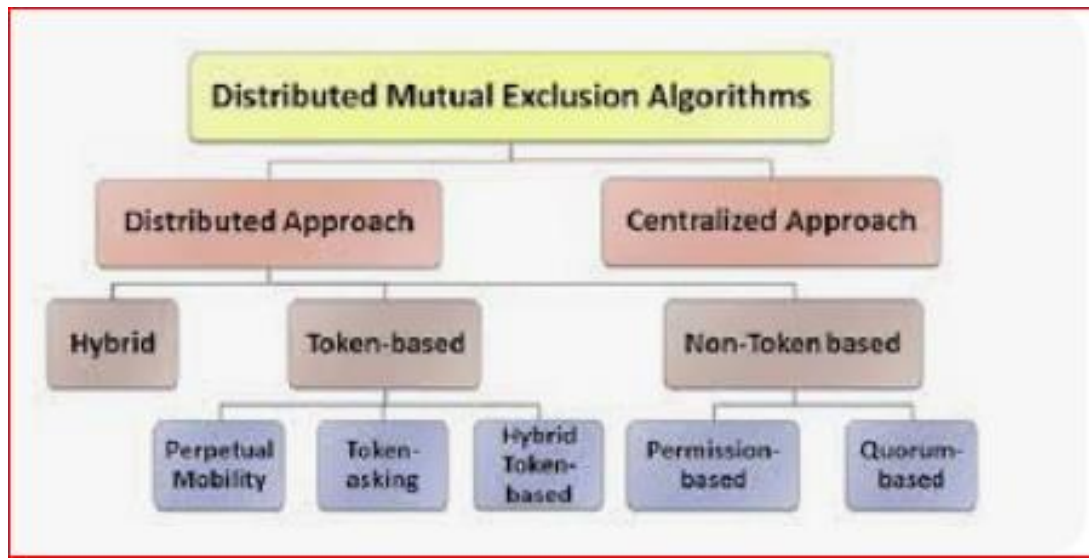
Under **heavy load** conditions, there is always a pending request for critical section at a site.

- As we know shared variables or a local kernel can not be used to implement mutual exclusion in distributed systems.
- Message passing is a way to implement mutual exclusion.

## **2.4 Categories of mutual exclusion algorithms**

- Dijkstra described and solved the mutual exclusion problem.

- Dijkstra stated that any solution to the mutual exclusion problem must satisfy 4 constraints.
- Dijkstra's algorithm guaranteed mutual exclusion and was deadlock-free, but it did not guarantee fairness.
- That is, it was possible that a process seeking access to its critical section waited indefinitely while others entered and exited their critical sections.



Types of mutual exclusion algorithms:-

- a) Centralized Algorithm
- b) Distributed Algorithm

**a) Centralized Algorithm :-**

- As its name implies, there is single coordinator which handles all the requests to access the shared data.
- Every process takes permission to the coordinator, if coordinator agree and will give permission then a particular process can enter into CS.
- Coordinator will maintain a queue and keep all the requests in order.

**Benefits**

- i) It is Fair algorithm; it follow FIFO algorithm concept for giving permission.
- ii) It is very simple algorithm in context of implementation.
- iii) We can use this scheme for general resource allocation. Shortcomings
- iv) No multiple point failures, No fault tolerant.
- v) Uncertainty between No-reply and permission denied.
- vi) Performance bottleneck because of single coordinator in a large system.

**b) Distributed algorithm :-**

- In distributed algorithm , there is no coordinator.
- For permission to enter into CS, Every process communicates to other process.

**These algorithms are sectioned into two parts –**

- i) Non-Token based algorithms
- ii) Token based algorithms

**Contention (Non-Token) based algorithms :-**

In this algorithm, process converse with a group of other process to select who should execute the critical section first.

These algorithms also divided into 2 parts:

1. Timestamp based
2. Voting scheme

**Controlled (TOKEN) BASED ALGORITHMS :**

- Token-based algorithms are the algorithm in which a site is allowed to enter its CS if it possesses the token.
- This token is unique among the processes.
- Instead of timestamps Token based algorithms use sequence numbers to differentiate between old and existing requests.
- Usually FIFO message delivery is not adopted.

**Below are the three approaches based on message passing to implement mutual exclusion in distributed systems:**

- a) Token Based Algorithm

- b) Non-token based approach
- c) c) Quorum based approach

#### **a) Token Based Algorithm**

- A unique token is shared among all the sites.
  - The central server algorithm simulates a single processor system.
  - One process in the distributed system is elected as the coordinator.
  - When a process wants to enter a resource, it sends a request message identifying the resource, if there are more than one, to the coordinator.
  - If a site possesses the unique token, it is allowed to enter its critical section.
  - If nobody is currently in the section, the coordinator sends back a grant message (Figure 1b) and marks that process as using the resource.
  - If, however, another process has previously claimed the resource, the server simply does not reply, so the requesting process is blocked.
  - The coordinator keeps state on which process is currently granted the resource and a which processes are requesting the resource.
- **Token-based algorithm handle the failure of a site that possesses the token in this way:**
    - If a site that possesses the token fails, then
      - the token is lost until the site recovers or another site generates a new token.
    - In the meantime, no site can enter the critical section.
  - This approach uses sequence number to order requests for the critical section.
  - Each requests for critical section contains a sequence number. This sequence number is used to distinguish old and current requests.
  - This approach insures Mutual exclusion as the token is unique

#### **Two basic Timestamp-based Contention algorithms are:**

##### **a) LAMPORT'S ALGORITHM :**

- Lamport designed a distributed MUTEX algorithm on the basis of his concept of logical clock .

- This is a algorithm which is a non-token based scheme.
- Non-token based protocols use timestamps to order requests for CS.

#### **b) RICART-AGRAWALA ALGORITHM:**

- Ricart-agrawala algorithm is an expansion and optimization of Lamport's protocol.
- This algorithm is also for MUTEX and it is a non-token based algorithm.
- This algorithm combines the RELEASE and REPLY message of lamport's algorithm and decreases the complexity of the algorithm by (N-1).

#### **b) Non-token based approach**

- A site communicates with other sites in order to determine which sites should execute critical section next. This requires exchange of two or more successive round of messages among sites.
- This approach use timestamps instead of sequence number to order requests for the critical section.
- Whenever a site make request for critical section, it gets a timestamp.
  - Timestamp is also used to resolve any conflict between critical section requests.

#### **The non-token-based approach ensure fairness among the sites**

- The non-token-based approach uses a logical clock to order requests for the critical section.
- Each site maintains its own logical clock, which gets updated with each message it sends or receives.
- This ensures that requests are executed in the order they arrive in the system, and that no site is unfairly prioritized.
- All algorithm which follows non-token based approach maintains a logical clock. Logical clocks get updated according to Lamport's scheme  
Example : Ricart-Agrawala Algorithm

#### **c) Quorum based approach**

- A quorum is a subset of sites that a site requests permission from to enter the critical section.
- The quorum is determined based on the size and number of overlapping subsets among the sites.
- Instead of requesting permission to execute the critical section from all other sites, Each site requests only a subset of sites which is called a **quorum**.
- Any two subsets of sites or Quorum contains a common site.
- This common site is responsible to ensure mutual exclusion

Example : Maekawa's Algorithm

### **Difference between Token based and Non-Token based Algorithms in Distributed System:**

S.No.	Token Based Algorithms	Non-Token Based Algorithms
1.	In the Token-based algorithm, a unique token is shared among all the sites in Distributed Computing Systems.	In Non-Token based algorithm, there is no token even not any concept of sharing token for access.
2.	Here, a site is allowed to enter the Critical Section if it possesses the token.	Here, two or more successive rounds of messages are exchanged between sites to determine which site is to enter the Critical Section next.
3.	The token-based algorithm uses the sequences to order the request for the Critical Section and to resolve the conflict for the simultaneous requests for the System.	Non-Token based algorithm uses the timestamp (another concept) to order the request for the Critical Section and to resolve the conflict for the simultaneous requests for the System.
4.	The token-based algorithm produces less message traffic as compared to Non-	Non-Token based Algorithm produces more message traffic as compared to

S.No.	Token Based Algorithms	Non-Token Based Algorithms
	Token based Algorithm.	the Token-based Algorithm.
5.	They are free from deadlock (i.e. here there are no two or more processes are in the queue in order to wait for messages that will actually can't come) because of the existence of unique token in the distributed system.	They are not free from the deadlock problem as they are based on timestamp only.
6.	Here, it is ensured that requests are executed exactly in the order as they are made in.	Here there is no surety of execution order.
7.	Token-based algorithms are more scalable as they can free your server from storing session state and also they contain all the necessary information which they need for authentication.	Non-Token based algorithms are less scalable than the Token-based algorithms because server is not free from its tasks.
8.	Here the access control is quite Fine-grained because here inside the token roles, permissions and resources can be easily specifying for the user.	Here the access control is not so fine as there is no token which can specify roles, permission, and resources for the user.
9.	Token-based algorithms make authentication quite easy.	Non-Token based algorithms can't make authentication easy.
10.	<p><b><u>Examples of Token-Based Algorithms are:</u></b></p> <p>(i) Singhal's Heuristic Algorithm</p> <p>(ii) <u>Raymonds Tree Based Algorithm</u></p>	<p><b><u>Examples of Non-Token Based Algorithms are:</u></b></p> <p>(i) <u>Lamport's Algorithm</u></p> <p>(ii) <u>Ricart-Agarwala Algorithm</u></p>

S.No.	Token Based Algorithms	Non-Token Based Algorithms
	(iii) <u>Suzuki-Kasami Algorithm</u>	(iii) <u>Maekawa's Algorithm</u>

\*\*\*\*\*

### **3. LAMPORT'S ALGORITHM**

#### **3.1 Lamport's Algorithm for Mutual Exclusion in Distributed System**

##### **3.1 Lamport's Logical Clocks:**

- The algorithm of Lamport timestamps is a simple algorithm used to determine the order of events in a distributed computer system.
- As different nodes or processes will typically not be perfectly synchronized, this algorithm is used to provide a partial ordering of events with minimal overhead, and conceptually provide a starting point for the more advanced vector clock method.

For synchronization of logical clocks, Lamport established a relation termed as "**happens-before" relation.**

1. Happens- before relation **is a transitive relation,**

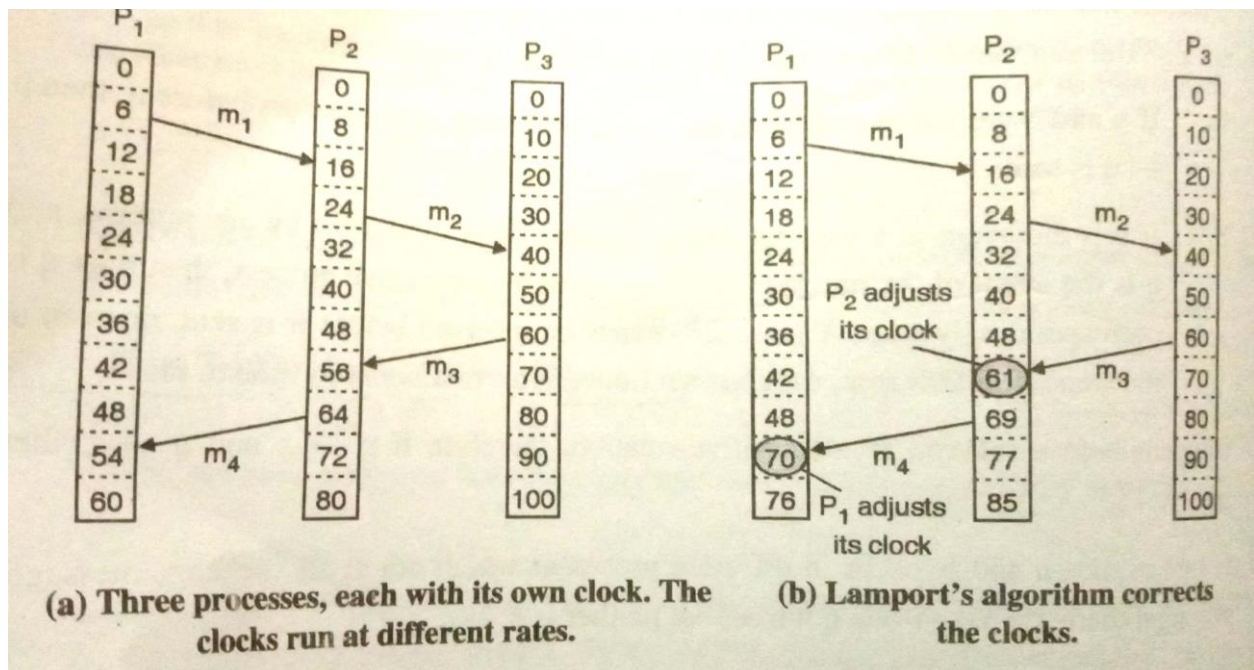
**therefore if  $p \rightarrow \rightarrow q$  and  $q \rightarrow \rightarrow r$ , then  $p \rightarrow \rightarrow r$ .**

2. If two events, a and b, occur in different processes which not at all exchange messages amongst them, then  **$a \rightarrow \rightarrow b$  is not true, but neither is  $b \rightarrow \rightarrow a$  which is antisymmetry.**

**The algorithm follows some simple rules:**

- A process increments its counter before each event in that process.
- When a process sends a message, it includes its counter value with the message.

- On receiving a message, the counter of the recipient is updated, if necessary, to the greater of its current counter and the timestamp in the received message.
- The counter is then incremented by 1 before the message is considered received.



- **Lamport's Distributed Mutual Exclusion Algorithm** is a permission based algorithm proposed by Lamport as an illustration of his synchronization scheme for distributed systems.
- In permission based timestamp is used to order critical section requests and to resolve any conflict between requests.
- In Lamport's Algorithm critical section requests are executed in the increasing order of timestamps i.e a request with smaller timestamp will be given permission to execute critical section first than a request with larger timestamp.

### 3.2 Basic messaging mechanisms in Mutual exclusion algorithms:

- Three type of messages (**REQUEST**, **REPLY** and **RELEASE**) are used and communication channels are assumed to follow FIFO order.
- A site send a **REQUEST** message to all other site to get their permission to enter critical section.

- A site send a **REPLY** message to requesting site to give its permission to enter the critical section.
- A site send a **RELEASE** message to all other site upon exiting the critical section.
- Every site  $S_i$ , keeps a queue to store critical section requests ordered by their timestamps. **request\_queue<sub>i</sub>** denotes the queue of site  $S_i$
- A timestamp is given to each critical section request using Lamport's logical clock.
- Timestamp is used to determine priority of critical section requests. Smaller timestamp gets high priority over larger timestamp. The execution of critical section request is always in the order of their timestamp.

### **3.3 Lamport Algorithm:**

- **To enter Critical section:**
  - When a site  $S_i$  wants to enter the critical section, it sends a request message **Request(ts<sub>i</sub>, i)** to all other sites and places the request on **request\_queue<sub>i</sub>**. Here,  $Ts_i$  denotes the timestamp of Site  $S_i$
  - When a site  $S_j$  receives the request message **REQUEST(ts<sub>i</sub>, i)** from site  $S_i$ , it returns a timestamped REPLY message to site  $S_i$  and places the request of site  $S_i$  on **request\_queue<sub>j</sub>**
- **To execute the critical section:**
  - A site  $S_i$  can enter the critical section if it has received the message with timestamp larger than **(ts<sub>i</sub>, i)** from all other sites and its own request is at the top of **request\_queue<sub>i</sub>**
  -
- **To release the critical section:**
  - When a site  $S_i$  exits the critical section, it removes its own request from the top of its request queue and sends a timestamped **RELEASE** message to all other sites
  - When a site  $S_j$  receives the timestamped **RELEASE** message from site  $S_i$ , it removes the request of  $S_i$  from its request queue

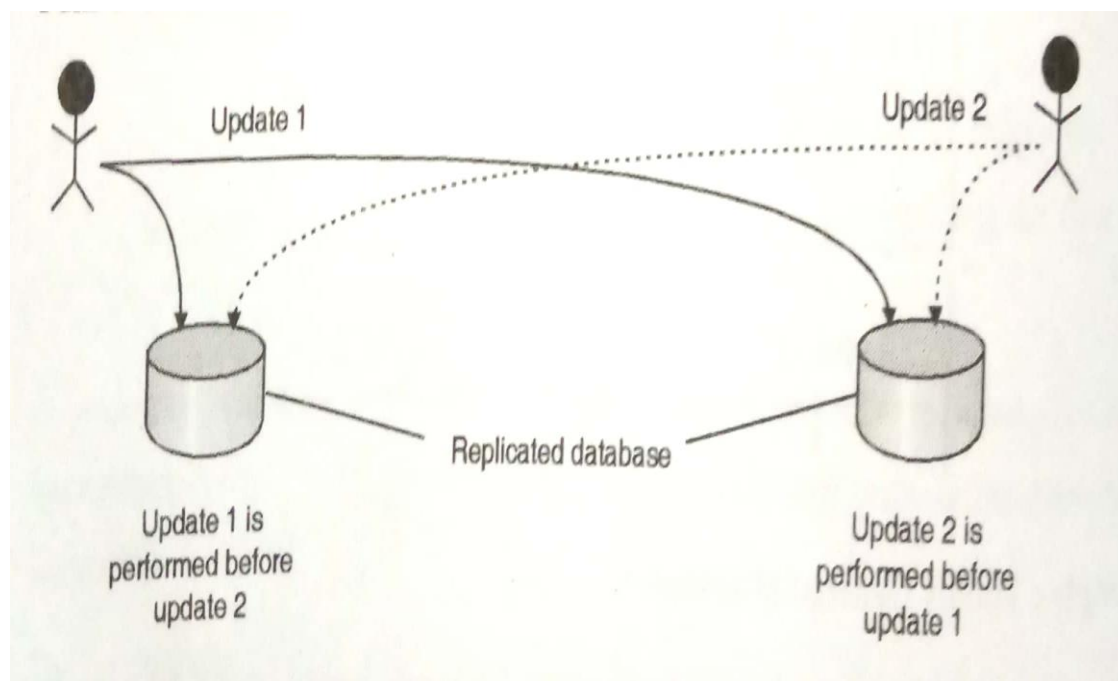
### **Message Complexity:**

Lamport's Algorithm requires invocation of  $3(N - 1)$  messages per critical section execution.

These  $3(N - 1)$  messages involves **Concurrent events in Lamport timestamps.**

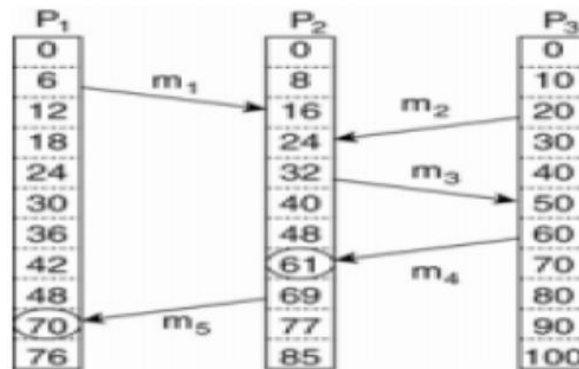
### 1. Totally Ordered Multicast

- We need to guarantee that concurrent updates on a replicated database are seen in the same order everywhere. This requires a totally-ordered multicast.
- Update 1: add 100 to an account (initial value = 1000)
- Update 2: add 1% interest to account.
- Lamport's logical clocks can be used to implement totally-ordered multicast in a completely distributed fashion
- Lamport's logical clocks can be used to implement totally-ordered multicast in a completely distributed fashion



### 1. Causality

- Lamport's logical clocks: If  $A \rightarrow B$  then  $L(A) < L(B)$ , Reverse is not true. Nothing can be said about events by comparing timestamps.
- Need to capture causality: If  $A \rightarrow B$  then A causally precedes B. Need a timestamping mechanism such that:  $T(A) < T(B)$  if A causally precedes B



Event A:  $m_1$  is received at  $t=16$

Event B:  $m_2$  is sent at  $t=20$

$L(A) < L(B)$ , but A does not causally precede B.

### 3.4 Timestamp in Lamport clocks

- Timestamp in Lamport clocks allow processes to assign sequence numbers (“timestamps”) to messages and other events so that all cooperating processes can agree on the order of related events.
- There is no assumption of a central time source and no concept of *when* events took place.
- Events are causally related if one event may potentially influence the outcome of another event. For instance, events in one process are causally related.
- If a process A sends a message to process B, all events that occurred on process A before the message was sent causally precede all the events that occur on B after B received the message.
- The central concept with logical clocks is the **happened-before** relation:  $a \rightarrow b$  represents that event  $a$  occurred before event  $b$ .
- This order is imposed upon consecutive events at a process and also upon a message being sent before it is received at another process.
- Beyond that, we can use the transitive property of the relationship to determine causality: **if  $a \rightarrow b$  and  $b \rightarrow c$  then  $a \rightarrow c$ .**

- If there is no **causal** relationship between two events (e.g., they occur on different processes that do not exchange messages or have not yet exchanged messages, even indirectly), then the events are **concurrent**.

Lamport's algorithm states that every event is timestamped (assigned a sequence number) and each message carries a timestamp of the sender's clock (sequence number).

A message comprises two events:

- (1) at the sender, we have the event of sending the message and
- (2) at the receiver, we have the event of receiving the message.

- The clock is a process-wide counter (e.g., a global variable) and is always incremented before each event.
  - When a message arrives, if the receiver's clock is less than or equal to the message's timestamp, the clock is set to the *message timestamp + 1*.
- This ensures that the timestamp marking the event of a received message will always be greater than the timestamp of that sent message.

### **3.5 Lamport Clocks**

#### **3.5.1 What are Lamport clocks?**

- Lamport clocks represent time logically in a distributed system.
- They are also known as logical clocks.
- The idea behind Lamport clocks is to disregard physical time and capture just a "happens-before" relationship between a pair of events.
- Each process maintains a single Lamport timestamp counter.

- Each event in the process is tagged with a value from this counter.
- The counter is incremented before the event timestamp is assigned.
- If a process has four events,  $a, b, c, d$ , the events would get Lamport timestamps of 1, 2, 3, 4, respectively.

### Why use Lamport clocks?

- Time synchronization is a key problem in distributed systems.
- Time is used to order events across servers. Using physical clocks to order events is challenging because real synchronization is impossible and clocks experience skew.
- A **clock skew** is when different clocks run at different rates, so we cannot assume that time  $t$  on node  $a$  happened before time  $t + 1$  on node  $b$ .
- Instead of employing physical time, Leslie Lamport proposed logical clocks that capture events' orderings through a "happens-before" relationship.

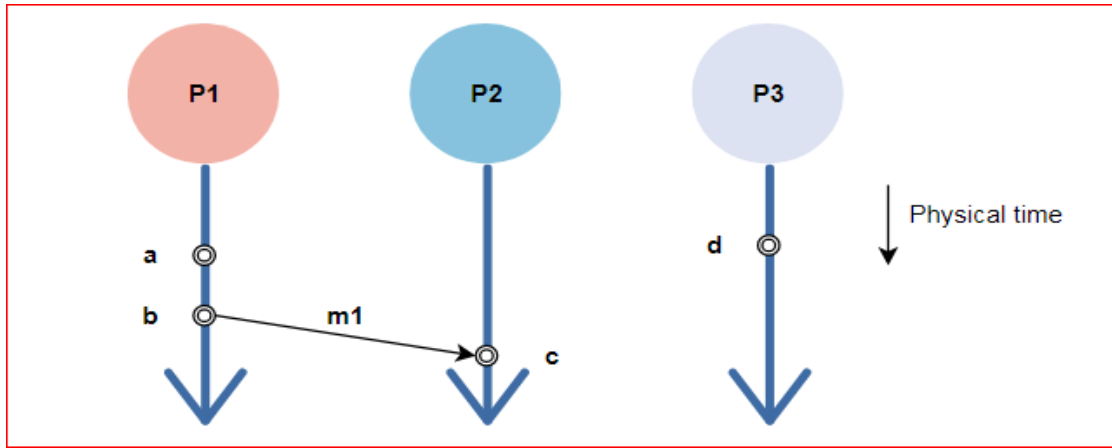
### Defining the happens-before relation

An **event** is something happening at a server node (sending or receiving messages, or a local execution step). If an event  $a$  happens before  $b$ , we write it as  $a \rightarrow b$ .

There are three conditions in which we can say an event  $a$  happens before  $b$ :

- If it is the same node and  $a$  occurs before  $b$ , then  $a \rightarrow b$
- If  $c$  is a message receipt of  $b$ , then  $b \rightarrow c$
- **Transitivity**: If  $a \rightarrow b$  and  $b \rightarrow c$ , then  $a \rightarrow c$

The following diagram illustrates the happens-before relation:



- A happens-before relation does not order all events.
- For instance, the events **a** and **d** are not related by  $\rightarrow$ .
- Hence, they are **concurrent**. Such events are written as **a** || **d**.

### 3.5.2 Implementing Lamport clocks

- Lamport clocks tag events in a distributed system and order them accordingly.
- We seek a clock time  $C(a)$  for every event **a**.
- The clock condition is defined as follows:

If **a**  $\rightarrow$  **b**, then  $C(a) < C(b)$ .

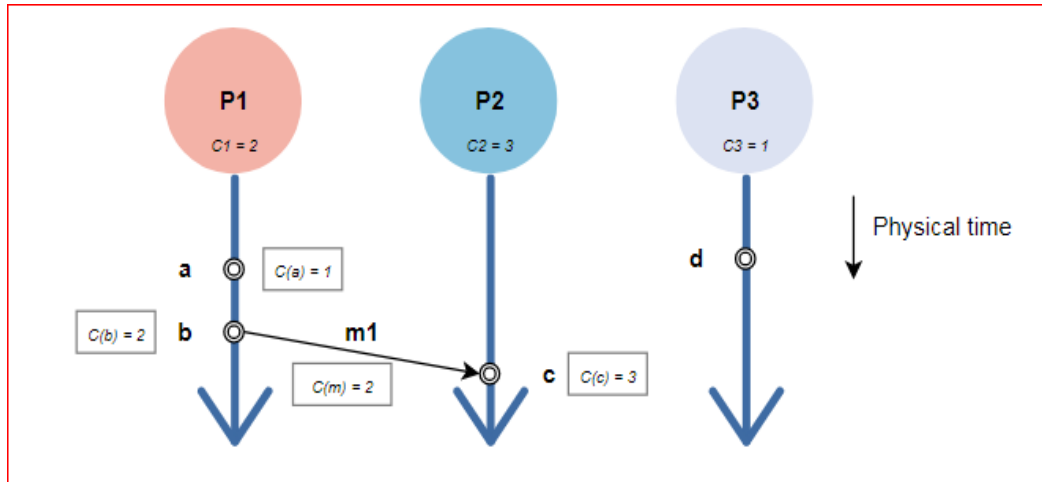
Each process maintains an **event counter**.

This event counter is the local Lamport clock.

#### The Lamport clock algorithm works in the following way:

- Before the execution of an event, the local clock is updated. This can be explained by the equation  $C_i = C_{i+1}$ , where  $i$  is the process identifier.
- When a message is sent to another process, the message contains the process' local clock,  $C_m$ .
- When a process receives a message  $m$ , it sets its local clock to  $1 + \max(C_i, C_m)$ .

The following diagram illustrates how the Lamport clock algorithm works:

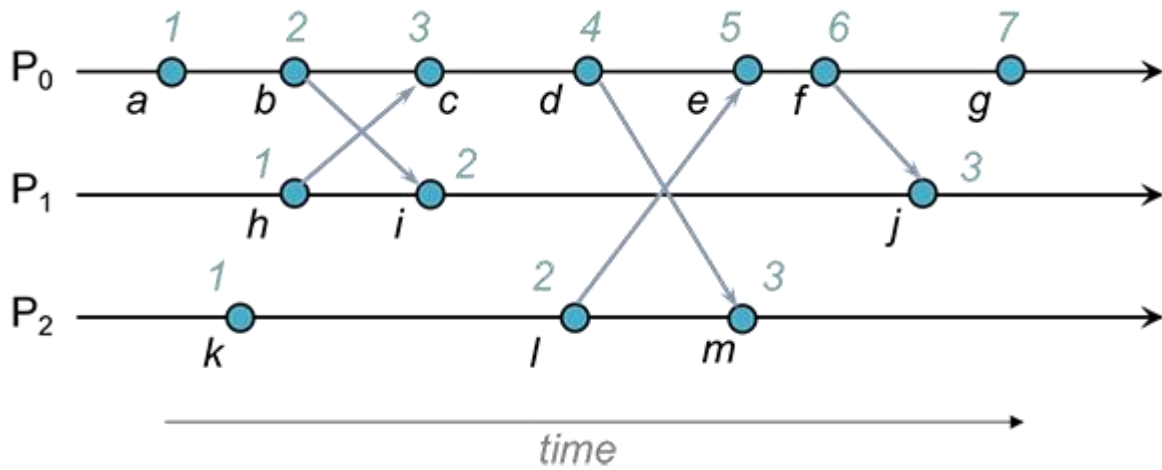


- In the example above, all local clocks start from a value of 0. Before the execution of an event, the local clock increments by 1.
- Notice that P2's clock starts from 0, but on the arrival of the message **m1** from P1, it updates its clock in accordance with the third rule mentioned above, i.e.,  $1 + \max(C_i, C_m)$  where  $C_i = 0$  and  $C_m = 2$ .
- Hence, P2's final clock value is 3.

**Note:** **d** is an event on P3, so,  $C(d) = 1$ , where **d** is parallel to **a**.

Let's look at an example.

- The figure below shows a bunch of events on three processes.
- Some of these events represent the sending of a message, others represent the receipt of a message, while others are just local events (e.g., writing some data to a file).
- With these per-process incrementing assignments, we get the clock values shown in the figure.



### Clock Assignment

This simple incrementing counter does not give us results that are consistent with causal events. If event  $a$  happened before event  $b$  then we expect  $clock(a) < clock(b)$ .

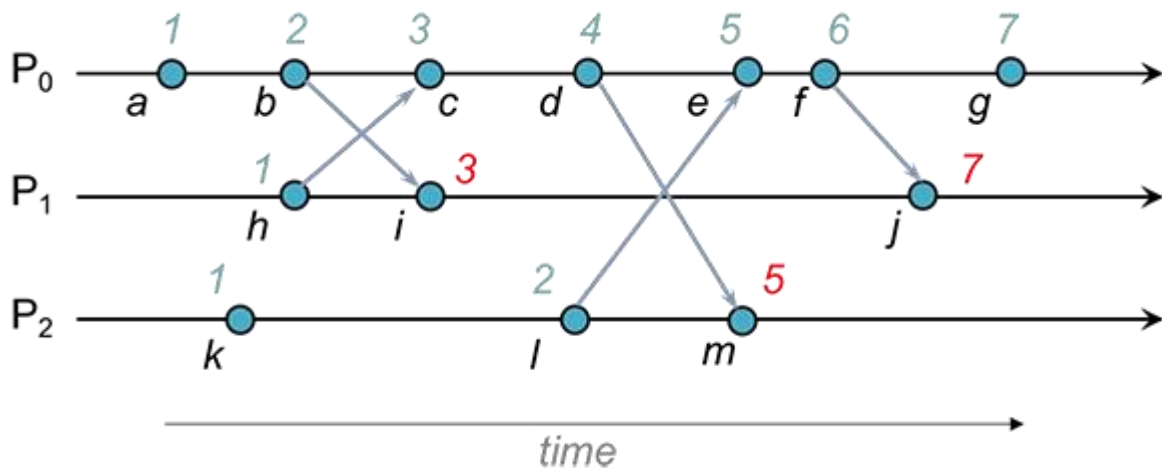
To make this work, Lamport timestamp generation has an extra step.

If an event is the sending of a message then the timestamp of that event is sent along with the message. If an event is the receipt of a message then the algorithm instructs you to compare the current value of the process' timestamp counter (which was just incremented before this event) with the timestamp in the received message.

If the timestamp of the received message is greater than or equal to that of the event, the event and the process' timestamp counter are both updated with the value of the timestamp in the received message plus one.

This ensures that the timestamp of the received event and all further timestamps on that process will be greater than that of the timestamp of the event of sending the message as well as all previous messages on that process.

- In the figure below, event  $i$  in process  $P_1$  is the receipt of the message sent by event  $b$  in  $P_0$ . If event  $i$  was just a normal local event, the  $P_1$  would assign it a timestamp of 2.
- However, since the received timestamp is 2, which is greater than or equal to 2, the timestamp counter is set to  $2+1$ , or 3.
- Event  $i$  gets the timestamp of 3. This preserves the relationship  $b \rightarrow i$ , that is,  $b$  happened before  $i$ .
- A local event after  $i$  would get a timestamp of 4 because the process  $P_1$ 's counter was set to 3 when the timestamp for  $i$  was adjusted.
- Event  $c$  in process  $P_0$  is the receipt of the message sent at event  $h$ . Here, the timestamp of  $c$  does not need to be adjusted.
- The timestamp in the message is 1, which is less than the event timestamp of 3 that  $P_0$  is ready to assign to  $c$ .
- If event  $j$  was a local event, it would get the next higher timestamp on  $P_1$ : 4.
- However, it is the receipt of a message that contains a timestamp of 6, which is greater than or equal to 4, so the event gets tagged with a timestamp of  $6+1 = 7$ .



### Lamport Clock Assignment

- With Lamport timestamps, we are assured that two causally-related events will have timestamps that reflect the order of events.
  - For example, event  $h$  happened before event  $m$  in the Lamport causal sense.
  - The chain of causal events is  $h \rightarrow c$ ,  $c \rightarrow d$ , and  $d \rightarrow m$ .

- Since the *happened-before* relationship is transitive, we know that  $h \rightarrow m$  ( $h$  happened before  $m$ ).

Lamport timestamps reflect this.

- The timestamp for  $h$  (1) is less than the timestamp for  $m$  (7).
- However, just by looking at timestamps we cannot conclude that there is a causal happened-before relation.
- For instance, because the timestamp for  $k$  (1) is less than the timestamp for  $i$  (3) does not mean that  $k$  happened before  $i$ .
- Those events happen to be concurrent but we cannot discern that by looking at Lamport timestamps.
- We need need to employ a different technique to be able to make that determination. That technique is the use of *vector timestamps*.
- One result of Lamport timestamps is that multiple events on different processes may all be tagged with the same timestamp.
- We can force each timestamp to be unique by suffixing it with a globally unique process number.
- While these new timestamps will not relate to real time ordering, each will be a unique number that can be used for consistent comparisons of timestamps among multiple processes (e.g., if we need to make a decision on who gets to access a resource based on comparing two timestamps).

**A second deficiency with Lamport timestamps is that, by looking at timestamps,** one cannot determine whether there is a causal relationship between two events.

For example, just because event  $a$  has a timestamp of 5 and event  $b$  has a timestamp of 6, it does not imply that event  $a$  happened before event  $b$ .

\*\*\*\*\*

## **4. RICART-AGRAWALA ALGORITHM**

## 4.1 Ricart-Agrawala algorithm

### What is the Ricart-Agrawala algorithm?

- Ricart–Agrawala algorithm is an algorithm for mutual exclusion in a distributed system proposed by Glenn Ricart and Ashok Agrawala.
- This algorithm is an extension and optimization of Lamport's Distributed Mutual Exclusion Algorithm.
- Like Lamport's Algorithm, it also follows permission-based approach to ensure mutual exclusion. In this algorithm:
- Two type of messages ( **REQUEST** and **REPLY**) are used and communication channels are assumed to follow FIFO order.
- A site send a **REQUEST** message to all other site to get their permission to enter the critical section.
- A site send a **REPLY** message to another site to give its permission to enter the critical section.
- A timestamp is given to each critical section request using Lamport's logical clock.
- Timestamp is used to determine priority of critical section requests. Smaller timestamp gets high priority over larger timestamp.
- The execution of critical section request is always in the order of their timestamp.

### Algorithm:

- **To enter Critical section:**
  - When a site  $S_i$  wants to enter the critical section, it send a timestamped **REQUEST** message to all other sites.
  - When a site  $S_j$  receives a **REQUEST** message from site  $S_i$ , It sends a **REPLY** message to site  $S_i$  if and only if
    - Site  $S_j$  is neither requesting nor currently executing the critical section.

- In case Site  $S_j$  is requesting, the timestamp of Site  $S_i$ 's request is smaller than its own request.
- **To execute the critical section:**
  - Site  $S_i$  enters the critical section if it has received the **REPLY** message from all other sites.
- **To release the critical section:**
  - Upon exiting site  $S_i$  sends **REPLY** message to all the deferred requests.

#### **Message Complexity:**

- Ricart–Agrawala algorithm requires invocation of  $2(N - 1)$  messages per critical section execution.
- These  $2(N - 1)$  messages involves
  - $(N - 1)$  request messages
  - $(N - 1)$  reply messages

#### **Advantages of the Ricart-Agrawala Algorithm:**

- i) **Low message complexity:** The algorithm has a low message complexity as it requires only  $(N-1)$  messages to enter the critical section, where  $N$  is the total number of nodes in the system.
- ii) **Scalability:** The algorithm is scalable and can be used in systems with a large number of nodes.
- iii) **Non-blocking:** The algorithm is non-blocking, which means that a node can continue executing its normal operations while waiting to enter the critical section.

#### **Drawbacks of Ricart–Agrawala algorithm:**

- i) **Unreliable approach:**

- Failure of any one of node in the system can halt the progress of the system. In this situation, the process will starve forever.
- The problem of failure of node can be solved by detecting failure after some timeout.

ii) **Performance:**

- Synchronization delay is equal to maximum message transmission time
- It requires  $2(N - 1)$  messages per Critical section execution

**What is the difference between Lamport algorithm and Ricart-Agrawala algorithm?**

- Both the Ricart & Agrawala and Lamport algorithms are contention-based algorithms.
- With Lamport's algorithm, everyone immediately responds to a mutual exclusion request message whereas with the Ricart & Agrawala, a process that is using the resource will delay its response until it is done.

## **5. SUZUKI – KASAMI ALGORITHM**

### **5.1 Suzuki–Kasami Algorithm for Mutual Exclusion in Distributed System**

- **Suzuki–Kasami algorithm** is a token-based algorithm for achieving mutual exclusion in distributed systems.
- The Main idea Completely connected network of processes
- There is one token in the network.
- The holder of the token has the permission to enter CS. Any other process trying to enter CS must acquire that token.
- Thus the token will move from one process to another based on demand. I want to enter CS.
- This is modification of Ricart–Agrawala algorithm, a permission based (Non-token based) algorithm which uses **REQUEST** and **REPLY** messages to ensure mutual exclusion.

- In token-based algorithms, A site is allowed to enter its critical section if it possesses the unique token.
- Non-token based algorithms uses timestamp to order requests for the critical section where as sequence number is used in token based algorithms.
- Each requests for critical section contains a sequence number.
- This sequence number is used to distinguish old and current requests.

## **5.2 Data structure and Notations:**

\_An array of integers  $RN[1...N]$

A site  $S_i$  keeps  $RN_i[1...N]$ , where  $RN_i[j]$  is the largest sequence number received so far through **REQUEST** message from site  $S_i$ .

- An array of integer  $LN[1...N]$   
This array is used by the token.  $LN[j]$  is the sequence number of the request that is recently executed by site  $S_j$ .
- A queue  $Q$   
This data structure is used by the token to keep record of ID of sites waiting for the token

## **5.1 Algorithm:**

\_To enter Critical section:

- When a site  $S_i$  wants to enter the critical section and it does not have the token then it increments its sequence number  $RN_i[i]$  and sends a request message **REQUEST(i, sn)** to all other sites in order to request the token.  
Here **sn** is update value of  $RN_i[i]$
- When a site  $S_j$  receives the request message **REQUEST(i, sn)** from site  $S_i$ , it sets  $RN_j[i]$  to maximum of  $RN_j[i]$  and **sn** i.e  $RN_j[i] = \max(RN_j[i], sn)$ .
- After updating  $RN_j[i]$ , Site  $S_j$  sends the token to site  $S_i$  if it has token and  $RN_j[i] = LN[i] + 1$
- **To execute the critical section:**

- Site  $S_i$  executes the critical section if it has acquired the token.
- **To release the critical section:**  
After finishing the execution Site  $S_i$  exits the critical section and does following:
  - sets  $LN[i] = RN[i]$  to indicate that its critical section request  $RN[i]$  has been executed
  - For every site  $S_j$ , whose ID is not present in the token queue  $Q$ , it appends its ID to  $Q$  if  $RN[j] = LN[j] + 1$  to indicate that site  $S_j$  has an outstanding request.
  - After above updation, if the Queue  $Q$  is non-empty, it pops a site ID from the  $Q$  and sends the token to site indicated by popped ID.
  - If the queue  $Q$  is empty, it keeps the token

### **Message Complexity:**

- The algorithm requires 0 message invocation if the site already holds the idle token at the time of critical section request or maximum of N message per critical section execution.
- This N messages involves
  - (N – 1) request messages
  - 1 reply message

### **Drawbacks of Suzuki–Kasami Algorithm:**

**Non-symmetric Algorithm:** A site retains the token even if it does not have requested for critical section. According to definition of symmetric algorithm

“No site possesses the right to access its critical section when it has not been requested.”

### **Performance:**

Synchronization delay is 0 and no message is needed if the site holds the idle token at the time of its request.

- In case site does not holds the idle token, the maximum synchronization delay is equal to maximum message transmission time and a maximum of N message is required per critical section invocation.

\*\*\*\*\*

## **6. DEADLOCK**

### **6.1 Introduction to Deadlock**

- Deadlock is a fundamental problem in distributed systems.
- Deadlock refers to the state when two processes compete for the same resource and end up locking the resource by one of the processes and the other one is prevented from acquiring that resource.
- A process may request resources in any order, which may not be known a priori and a process can request resource while holding others.
- If the sequence of the allocations of resources to the processes is not controlled, deadlocks can occur.
- A deadlock is a state where a set of processes request resources that are held by other processes in the set.

Three commonly used strategies to handle deadlocks are as follows:

- i) Avoidance: Resources are carefully allocated to avoid deadlocks.
- ii) Prevention: Constraints are imposed on the ways in which processes request resources in order to prevent deadlocks.
- iii) Detection and recovery: Deadlocks are allowed to occur and a detection algorithm is used to detect them. After a deadlock is detected, it is resolved by certain means.

#### **Types of Distributed Deadlock:**

There are two types of Deadlocks in Distributed System:

**i) Resource Deadlock:** A resource deadlock occurs when two or more processes wait permanently for resources held by each other.

A process that requires certain resources for its execution, and cannot proceed until it has acquired all those resources.

It will only proceed to its execution when it has acquired all required resources.

It can also be represented using AND condition as the process will execute only if it has all the required resources.

## **ii) Communication Deadlock:**

- On the other hand, a communication deadlock occurs among a set of processes when they are blocked waiting for messages from other processes in the set in order to start execution but there are no messages in transit between them.
- When there are no messages in transit between any pair of processes in the set, none of the processes will ever receive a message.
- This implies that all processes in the set are deadlocked. Communication deadlocks can be easily modeled by using WFGs to indicate which processes are waiting to receive messages from which other processes.
- Hence, the detection of communication deadlocks can be done in the same manner as that for systems having only one unit of each resource type.

## **7. System Model**

### **7.1 Introduction to System Model**

- A distributed program is composed of a set of  $n$  asynchronous processes  $p_1, p_2, \dots, p_i, \dots, p_n$  that communicates by message passing over the communication network.
- Without loss of generality we assume that each process is running on a different processor.
- The processors do not share a common global memory and communicate solely by passing messages over the communication network.
- There is no physical global clock in the system to which processes have instantaneous access.
- The communication medium may deliver messages out of order, messages may be lost garbled or duplicated due to timeout and retransmission, processors may fail and communication links may go down.

**We following assumptions make the:**

- The systems have only reusable resources.
- Processes are allowed to make only exclusive access to resources.
- There is only one copy of each resource.

**A process can be in two states:**

- i) running or
- ii) blocked

- In the running state (also called active state), a process has all the needed resources and is either executing or is ready for execution.
- In the blocked state, a process is waiting to acquire some resource.

**Wait-For-Graph (WFG)**

- The state of the system can be modeled by directed graph, called a wait for graph (WFG). In a WFG , nodes are processes and there is a directed edge from node P1 to mode P2 if P1 is blocked and is waiting for P2 to release some resource.
- A system is deadlocked if and only if there exists a directed cycle or knot in the WFG.
- Figure 1 shows a WFG, where process P11 of site 1 has an edge to process P21 of site 1 and P32 of site 2 is waiting for a resource which is currently held by process P21.
- At the same time process P32 is waiting on process P33 to release a resource.
- If P21 is waiting on process P11, then processes P11, P32 and P21 form a cycle and all the four processes are involved in a deadlock depending upon the request model.

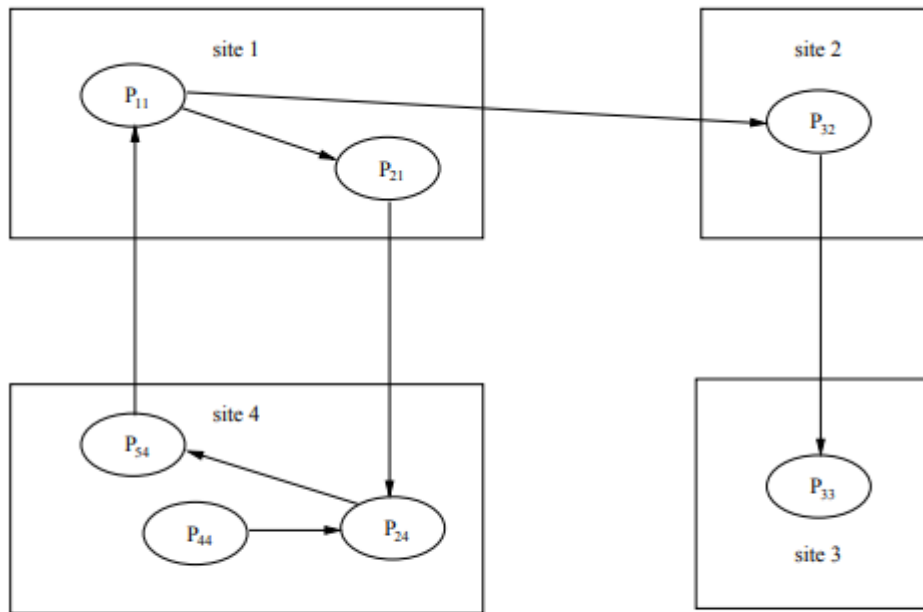


Figure 1: An Example of a WFG

## **8. PRELIMINARIES**

### **8.1 Preliminaries on Deadlock Handling Strategies**

- There are three strategies for handling deadlocks, viz.,
  - i) deadlock prevention,
  - ii) deadlock avoidance, and
  - iii) deadlock detection.
- Handling of deadlock becomes highly complicated in distributed systems because no site has accurate knowledge of the current state of the system and because every inter-site communication involves a finite and unpredictable delay.
- Deadlock prevention is commonly achieved either by having a process acquire all the needed resources simultaneously before it begins executing or by preempting a process which holds the needed resource.
- This approach is highly inefficient and impractical in distributed systems.

- In deadlock avoidance approach to distributed systems, a resource is granted to a process if the resulting global system state is safe (note that a global state includes all the processes and resources of the distributed system).
- However, due to several problems, deadlock avoidance is impractical in distributed systems.
- Deadlock detection requires examination of the status of process-resource interactions for presence of cyclic wait.
- Deadlock detection in distributed systems seems to be the best approach to handle deadlocks in distributed systems.

## **8.2 Issues in Deadlock Detection**

Deadlock handling using the approach of deadlock detection entails addressing two basic issues: First, detection of existing deadlocks and second resolution of detected deadlocks.

Detection of deadlocks involves addressing two issues:

- i) Maintenance of the WFG and
- ii) searching of the WFG for the presence of cycles (or knots).

### **Correctness Criteria:**

A deadlock detection algorithm must satisfy the following two conditions:

#### **(i) Progress (No undetected deadlocks):**

- The algorithm must detect all existing deadlocks in finite time.
- In other words, after all wait-for dependencies for a deadlock have formed, the algorithm should not wait for any more events to occur to detect the deadlock.

#### **(ii) Safety (No false deadlocks):**

- The algorithm should not report deadlocks which do not exist (called phantom or false deadlocks).

### **8.3 Resolution of a Detected Deadlock**

- Deadlock resolution involves breaking existing wait-for dependencies between the processes to resolve the deadlock.
- It involves rolling back one or more deadlocked processes and assigning their resources to blocked processes so that they can resume execution.

## **9. MODELS OF DEADLOCKS**

- The models of deadlocks are explained based on their hierarchy.
- The diagrams illustrate the working of the deadlock models. Pa, Pb, Pc, Pd are passive processes that had already acquired the resources
- . Pe is active process that is requesting the resource.
- Distributed systems allow many kinds of resource requests.
- A process might require a single resource or a combination of resources for its execution.
- Introducing a hierarchy of request models starting with very restricted forms to the ones with no restrictions whatsoever.
- This hierarchy shall be used to classify deadlock detection algorithms based on the complexity of the resource requests they permit.

### **9.1 Single Resource Model**

- A process can have at most one outstanding request for only one unit of a resource.
- The maximum out-degree of a node in a WFG for the single resource model can be 1, the presence of a cycle in the WFG shall indicate that there is a deadlock.

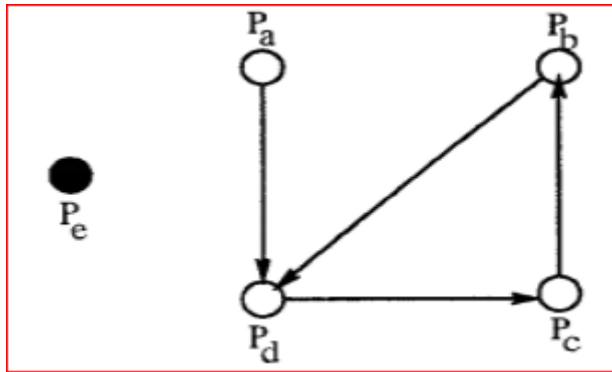


Fig: Deadlock in single resource model

- Distributed systems allow several kinds of resource requests.

## **9.2 The AND Model**

In the AND model, a process can request for more than one resource simultaneously and the request is satisfied only after all the requested resources are granted to the process. The out degree of a node in the WFG for AND model can be more than 1.

The presence of a cycle in the WFG indicates a deadlock in the AND model.

Since in the single-resource model, a process can have at most one outstanding request, the AND model is more general than the single-resource model.

### **AND Model**

- In the AND model, a passive process becomes active (i.e., its activation condition is fulfilled) only after a message from each process in its dependent set has arrived.
- In the AND model, a process can request more than one resource simultaneously and the request is satisfied only after all the requested resources are granted to the process.
- The requested resources may exist at different locations.
- The out degree of a node in the WFG for AND model can be more than 1.
- The presence of a cycle in the WFG indicates a deadlock in the AND model.
- Each node of the WFG in such a model is called an AND node.
- In the AND model, if a cycle is detected in the WFG, it implies a deadlock but not vice versa. That is, a process may not be a part of a cycle, it can still be deadlocked.

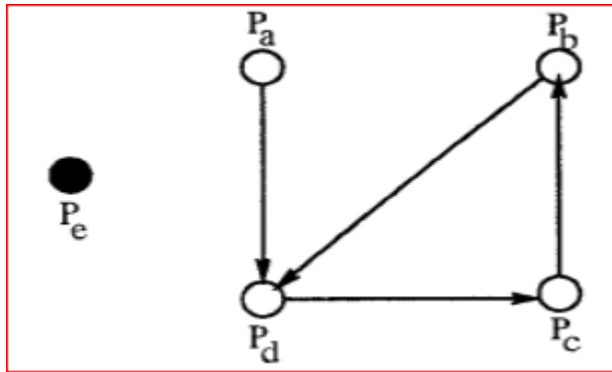


Fig : Deadlock in AND model

- Consider the example WFG described in the Figure 1.
- P11 has two outstanding resource requests. In case of the AND model, P11 shall become active from idle state only after both the resources are granted.
- There is a cycle P11->P21->P24->P54->P11 which corresponds to a deadlock situation. That is, a process may not be a part of a cycle, it can still be deadlocked. Consider process P44 in Figure 1.
- It is not a part of any cycle but is still deadlocked as it is dependent on P24 which is deadlocked.

### **9.3 The OR Model**

- A process can make a request for numerous resources simultaneously and the request is satisfied if any one of the requested resources is granted.
- Presence of a cycle in the WFG of an OR model does not imply a deadlock in the OR model.
- In the OR model, the presence of a knot indicates a deadlock.

Consider example in Figure 1:

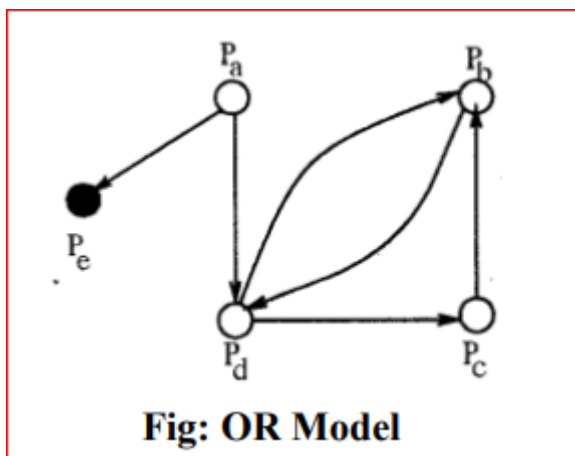
- If all nodes are OR nodes, then process P11 is not deadlocked because once process P33 releases its resources, P32 shall become active as one of its requests is satisfied.
- After P32 finishes execution and releases its resources, process P11 can continue with its processing. In the OR model, the presence of a knot indicates a deadlock

### **Deadlock in OR model:**

a process  $P_i$  is blocked if it has a pending OR request to be satisfied.

With every blocked process, there is an associated set of processes called dependent set.

- A process shall move from an idle to an active state on receiving a grant message from any of the processes in its dependent set.
- A process is permanently blocked if it never receives a grant message from any of the Processes in its dependent set.
- A set of processes  $S$  is deadlocked if all the processes in  $S$  are permanently blocked.
- In short, a process is deadlocked or permanently blocked, if the following conditions are met:
  1. Each of the process in the set  $S$  is blocked.
  2. The dependent set for each process in  $S$  is a subset of  $S$ .
  3. No grant message is in transit between any two processes in set  $S$ .
- A blocked process  $P$  in the set  $S$  becomes active only after receiving a grant message from a process in its dependent set, which is a subset of  $S$ .



### **The AND-OR Model**

- A generalization of the previous two models (OR model and AND model) is the AND-OR model.
- In the AND-OR model, a request may specify any combination of and and or in the resource request.
- For example, in the AND-OR model, a request for multiple resources can be of the form  $x$  and  $(y \text{ or } z)$ .
- To detect the presence of deadlocks in such a model, there is no familiar construct of graph theory using WFG.
- Since a deadlock is a stable property, a deadlock in the AND-OR model can be detected by repeated application of the test for OR-model deadlock.

### **The $(P_q)$ Model**

- The  $(P_q)$  model (called the P-out-of-Q model) allows a request to obtain any  $k$  available resources from a pool of  $n$  resources.
- It has the same expressive power as the AND-OR model.
- However,  $(P_q)$  model lends itself to a much more compact formation of a request.
- Every request in the  $(P_q)$  model can be expressed in the AND-OR model and vice-versa. Note that AND requests for  $p$  resources can be stated as  $(P_p)$  and OR requests for  $p$  resources can be stated as  $(P_1)$ .

### **Unrestricted Model**

In the unrestricted model, no assumptions are made regarding the underlying structure of resource requests.

Only one assumption that the deadlock is stable is made and hence it is the most general model.

### ***This model helps separate concerns:***

Concerns about properties of the problem (stability and deadlock) are separated from underlying distributed systems computations (e.g., message passing versus synchronous communication).

- The **Chandy–Misra–Haas algorithm resource model** checks for deadlock in a distributed system.
- It was developed by K. Mani Chandy, Jayadev Misra and Laura M Haas.
- Chandy-Misra-Haas's Distributed Deadlock Detection Algorithm is an edge chasing algorithm to detect deadlock in distributed systems.
- In edge chasing algorithm, a special message called probe is used in deadlock detection.
- A probe is a triplet (i, j, k) which denotes that process P<sub>i</sub> has initiated the deadlock detection and the message is being sent by the home site of process P<sub>j</sub> to the home site of process P<sub>k</sub>.
- The probe message circulates along the edges of WFG to detect a cycle.
- When a blocked process receives the probe message, it forwards the probe message along its outgoing edges in WFG.
- A process P<sub>i</sub> declares the deadlock if probe messages initiated by process P<sub>i</sub> returns to itself.

### **Locally dependent**

- Consider the n processes P<sub>1</sub>, P<sub>2</sub>, P<sub>3</sub>, P<sub>4</sub>, P<sub>5</sub>, ..., P<sub>n</sub> which are performed in a single system (controller).
- P<sub>1</sub> is locally dependent on P<sub>n</sub>, if P<sub>1</sub> depends on P<sub>2</sub>, P<sub>2</sub> on P<sub>3</sub>, so on and P<sub>n-1</sub> on P<sub>n</sub>.
- That is, if  $P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow \dots \rightarrow P_n$ , then  $P_1$  is locally dependent on  $P_n$ .

- If  $P_1$  is said to be locally dependent to itself if it is locally dependent on  $P_n$  and  $P_n$  depends on  $P_1$ :
- i.e. if  $\{P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow \dots \rightarrow P_n \rightarrow P_1\}$ , then  $\{P_1\}$  is locally dependent on itself.

### **Description**

---

- The algorithm uses a message called  $\text{probe}(i,j,k)$  to transfer a message from controller of process  $P_j$  to controller of process  $P_k$ .
- It specifies a message started by process  $P_i$  to find whether a deadlock has occurred or not. Every process  $P_j$  maintains a boolean array *dependent* which contains the information about the processes that depend on it.
- Initially the values of each array are all "false".

### **Controller sending a probe**

- Before sending, the probe checks whether  $P_j$  is locally dependent on itself. If so, a deadlock occurs.
- Otherwise it checks whether  $P_j$  and  $P_k$  are in different controllers, are locally dependent and  $P_j$  is waiting for the resource that is locked by  $P_k$ .
- Once all the conditions are satisfied it sends the probe.

### **Controller receiving a probe**

- On the receiving side, the controller checks whether  $P_k$  is performing a task. If so, it neglects the probe.
- Otherwise, it checks the responses given  $P_k$  to  $P_j$  and  $\text{dependent}_k(i)$  is false.
- Once it is verified, it assigns true to  $\text{dependent}_k(i)$ .
- Then it checks whether  $k$  is equal to  $i$ . If both are equal, a deadlock occurs, otherwise it sends the probe to next dependent process.

## Algorithm

In pseudocode, the algorithm works as follows:

### **Controller sending a problem**

```
if  $P_j$  is locally dependent on itself
    then declare deadlock
else for all  $P_j, P_k$  such that
    (i)  $P_i$  is locally dependent on  $P_j$ ,
    (ii)  $P_j$  is waiting for ' $P_k$  and
    (iii)  $P_j, P_k$  are on different controllers.
send probe(i, j, k). to home site of  $P_k$ 
```

### **Controller receiving a probe[edit]**

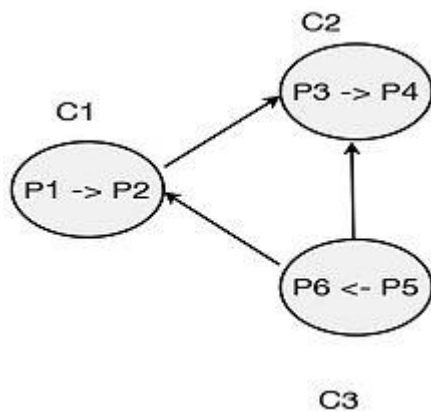
```
if
    (i)  $P_k$  is idle / blocked
    (ii)  $dependent_k(i) = \text{false}$ , and
    (iii)  $P_k$  has not replied to all requests of to  $P_j$ 
then begin
    "dependents"k(i) = true;
    if  $k == i$ 
        then declare that  $P_i$  is deadlocked
    else for all  $P_a, P_b$  such that
        (i)  $P_k$  is locally dependent on  $P_a$ ,
        (ii)  $P_a$  is waiting for ' $P_b$  and
```

(iii)  $P_a, P_b$  are on different controllers.

**send probe(i, a, b).** to home site of  $P_b$

**end**

### Example



occurrence of deadlock in distributed system

- $P_1$  initiates deadlock detection.  $C_1$  sends the probe saying  $P_2$  depends on  $P_3$ . Once the message is received by  $C_2$ , it checks whether  $P_3$  is idle.  $P_3$  is idle because it is locally dependent on  $P_4$  and updates  $dependent_3(2)$  to True.
- As above,  $C_2$  sends probe to  $C_3$  and  $C_3$  sends probe to  $C_1$ . At  $C_1$ ,  $P_1$  is idle so it update  $dependent_1(1)$  to True. Therefore, deadlock can be declared.

### Complexity

Consider that there are "m" controllers and "p" process to perform, to declare whether a deadlock has occurred or not, the worst case for controllers and processes must be visited. Therefore, the solution is  $O(m+p)$ . The time complexity is  $O(n)$ .

### Advantages:

- There is no need for special data structure. A probe message, which is very small and involves only 3 integers and a two dimensional boolean array *dependent* is used in the deadlock detection process.
- At each site, only a little computation is required and overhead is also low
- Unlike other deadlock detection algorithm, there is no need to construct any graph or pass nor to pass graph information to other sites in this algorithm.

- Algorithm does not report any false deadlock (also called phantom deadlock).

### **Disadvantages:**

- The main disadvantage of distributed detection algorithms is that all sites may not be aware of the processes involved in the deadlock which makes resolution difficult. Also, proof of correction of the algorithm is difficult.
- It may detect a false deadlock if there is a delay in message passing or if a message is lost. This can result in unnecessary process termination or resource preemption.
- It may not be able to detect all deadlocks in the system, especially if there are hidden deadlocks or if the system is highly dynamic.
- It is complex and difficult to implement correctly. It requires careful coordination between the processes, and any errors in the implementation can lead to incorrect results.
- It may not be scalable to large distributed systems with a large number of processes and resources. As the size of the system grows, the overhead and complexity of the algorithm also increase.

\*\*\*\*\*

## **10. CHANDY-MISRA-HAAS ALGORITHMS**

### **10.1 CHANDY-MISRA-HAAS ALGORITHM FOR THE AND MODEL**

- Chandy-Misra-Haas's distributed deadlock detection algorithm for AND model that is based on edge-chasing.
- The algorithm uses a special message called *probe*, which is a triplet (i, j, k),

denoting that it belongs to a deadlock detection initiated for process  $P_i$  and it is being sent by the home site of process  $P_j$  to the home site of process  $P_k$ .

- A probe message travels along the edges of the global WFG graph, and a deadlock is detected when a probe message returns to the process that initiated it.
- A process  $P_j$  is said to be *dependent* on another process  $P_k$  if there exists a sequence of processes  $P_j, P_{i1}, P_{i2}, \dots, P_{im}, P_k$  such that each process except  $P_k$  in the sequence is blocked and each process, except the  $P_j$ , holds a resource for which the previous process in the sequence is waiting.
- Process  $P_j$  is said to be *locally dependent* upon process  $P_k$  if  $P_j$  is dependent upon  $P_k$  and both the processes are on the same site.

### **Data Structures**

- Each process  $P_i$  maintains a boolean array, dependent  $i$ , where dependent  $i(j)$  is true only if  $P_i$  knows that  $P_j$  is dependent on it. Initially, dependent  $i(j)$  is false for all  $i$  and  $j$ .

## The Algorithm

The following algorithm is executed to determine if a blocked process is deadlocked:

```
if  $P_i$  is locally dependent on itself
  then declare a deadlock
  else for all  $P_j$  and  $P_k$  such that
    (a)  $P_i$  is locally dependent upon  $P_j$ , and
    (b)  $P_j$  is waiting on  $P_k$ , and
    (c)  $P_j$  and  $P_k$  are on different sites,
    send a probe (i, j, k) to the home site of  $P_k$ 
```

On the receipt of a probe (i, j, k), the site takes the following actions:

```
if
  (d)  $P_k$  is blocked, and
  (e)  $dependent_k(i)$  is false, and
  (f)  $P_k$  has not replied to all requests  $P_j$ ,
  then
    begin
      ...
       $dependent_k(i) = \text{true};$ 
      if  $k=i$ 
        then declare that  $P_i$  is deadlocked
      else for all  $P_m$  and  $P_n$  such that
        (a')  $P_k$  is locally dependent upon  $P_m$ , and
        (b')  $P_m$  is waiting on  $P_n$ , and
        (c')  $P_m$  and  $P_n$  are on different sites,
        send a probe (i, m, n) to the home site of  $P_n$ 
      end.
```

Therefore, a probe message is continuously circulated along the edges of the global WFG graph and a deadlock is detected when a probe message returns to its initiating process.

## Performance Analysis

In the algorithm, one probe message (per deadlock detection initiation) is sent on every edge of the WFG which has two sites. Thus, the algorithm exchanges at most  $m(n - 1)/2$  messages to detect a deadlock that involves  $m$  processes and that spans over  $n$  sites. The size of messages is fixed and is very small (only 3 integer words). Delay in

detecting a deadlock is  $O(n)$ .

## **10.2 CHANDY-MISRA-HAAS ALGORITHM FOR THE OR MODEL**

We now discuss Chandy-Misra-Haas distributed deadlock detection algorithm for OR model that is based on the approach of diffusion-computation.

A blocked process determines if it is deadlocked by initiating a diffusion computation.

Two types of messages are used in a diffusion computation:

query( $i, j, k$ ) and reply( $i, j, k$ ), denoting that they belong to a diffusion computation initiated by a process  $P_i$  and are being sent from process  $P_j$  to process  $P_k$ .

### **Basic Idea**

A blocked process initiates deadlock detection by sending query messages to all processes in its dependent set (i.e., processes from which it is waiting to receive a message).

If an active process receives a query or reply message, it discards it.

### **When a blocked process $P_k$ receives a query( $i, j, k$ ) message, it takes the following actions:**

- If this is the first query message received by  $P_k$  for the deadlock detection initiated by  $P_i$  (called the engaging query), then it propagates the query to all the processes in its dependent set and sets a local variable  $num_k(i)$  to the number of query messages sent.
- If this is not the engaging query, then  $P_k$  returns a reply message to it immediately provided  $P_k$  has been continuously blocked since it received the corresponding engaging query.
- Otherwise, it discards the query. Process  $P_k$  maintains a boolean variable  $wait_k(i)$  that denotes the fact that it has been continuously blocked since it received the last engaging query from process  $P_i$ .
- When a blocked process  $P_k$  receives a reply( $i, j, k$ ) message, it decrements  $num_k(i)$  only

if  $wait_k(i)$  holds. A process sends a reply message in response to an engaging query only after it has received a reply to every query message it had sent out for this engaging query. The initiator process detects a deadlock when it receives reply messages to all the query messages it had sent out.

### **The Algorithm**

The algorithm works as follows:

**Initiate a diffusion computation for a blocked process  $P_i$ :**

send query( $i, i, j$ ) to all processes  $P_j$  in the dependent set  $DS_i$  of  $P_i$ ;  
 $num_i(i) := |DS_i|$ ;  $wait_i(i) := \text{true}$ ;

**When a blocked process  $P_k$  receives a query( $i, j, k$ ):**

if this is the engaging query for process  $P_i$   
 then send query( $i, k, m$ ) to all  $P_m$  in its dependent set  $DS_k$ ;  
 $num_k(i) := |DS_k|$ ;  $wait_k(i) := \text{true}$   
 else if  $wait_k(i)$  then send a *reply*( $i, k, j$ ) to  $P_j$ .

**When a process  $P_k$  receives a reply( $i, j, k$ ):**

if  $wait_k(i)$   
 then begin  
 $num_k(i) := num_k(i) - 1$ ;  
 if  $num_k(i) = 0$   
 then if  $i=k$  then declare a deadlock  
 else send reply( $i, k, m$ ) to the process  $P_m$   
 which sent the engaging query.

For ease of presentation, we assumed that only one diffusion computation is initiated for a process. In practice, several diffusion computations may be initiated for a process (A diffusion computation is initiated every time the process gets blocked), but, at any time only one diffusion computation is current for any process. However, messages for outdated diffusion computations may still be in transit. The current diffusion computation can be distinguished from outdated ones by using sequence numbers.

### **Performance Analysis**

For every deadlock detection, the algorithm exchanges  $e$  query messages and  $e$  reply messages, where  $e=n(n-1)$  is the number of edges.

\*\*\*\*\*