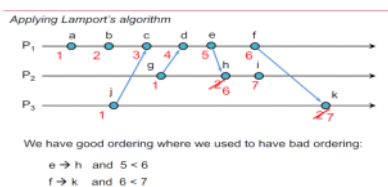
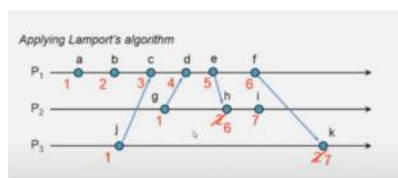


Casual relation yeh kehta hai ki agr A mein kuch change ho raha hai to uska asar B mein bhi padega

### Lamport's Algorithm

If  $a$  and  $b$  occur on different processes that do not exchange messages, then neither  $a \rightarrow b$  nor  $b \rightarrow a$  are true

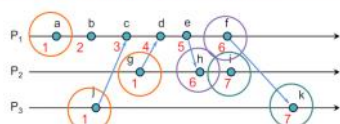
- These events are **concurrent**
- Otherwise, they are **causal**



### When a message arrives:

if receiver's clock  $<$  message\_timestamp  
set system clock to (message\_timestamp + 1)  
else do nothing

### Problem: Identical timestamps



$a \rightarrow b, b \rightarrow c, \dots$ : local events sequenced

$i \rightarrow c, f \rightarrow d, d \rightarrow g, \dots$ : Lamport imposes a send-receive relationship

Concurrent events (e.g.,  $b$  &  $g$ ;  $i$  &  $k$ ) may have the same timestamp ... or not

### Unique timestamps (total ordering)

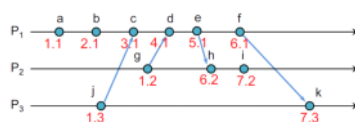
We can force each timestamp to be unique

- Define **global logical timestamp** ( $T_i, i$ )
  - $T_i$  represents local Lamport timestamp
  - $i$  represents process number (globally unique)
    - e.g., (host address, process ID)
- Compare timestamps:
  - $(T_i, i) < (T_j, j)$  if and only if
    - $T_i < T_j$  or
    - $T_i = T_j$  and  $i < j$

Does not necessarily relate to actual event ordering

Solution

### Unique (totally ordered) timestamps



### Problem: Detecting causal relations

If  $L(e) < L(e')$

- We cannot conclude that  $e \rightarrow e'$

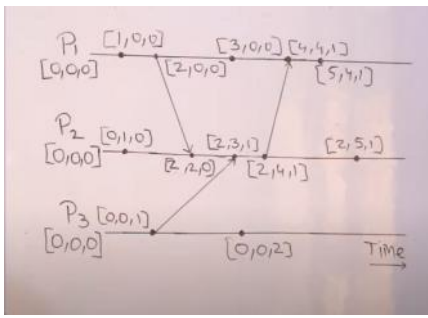
By looking at Lamport timestamps

- We cannot conclude which events are causally related

Solution: use a **vector clock**

Vector clocks are a way to prove the sequence of events by keeping version history based on each process that made changes to an object

Vector Clock



Vector initialized to 0 for each process  
 $V_i[j] = 0$  for  $i, j = 1, 2, 3 \dots N$

Increment vector before timestamp increment  
 $V_i[i] = V_i[i] + 1$

message is sent from  $P_i$  with  $V_i$  attached to it

When  $P_j$  receives message  
 $V_j[i] = \max(V_j[i], V_i[i])$

## Distributed Mutual Exclusion Algorithms

### Lamport's Distributed Mutual Exclusion Algorithm (Non-token)

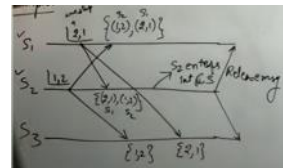
a request with smaller timestamp will be given permission to execute critical section first than a request with larger timestamp.

Three type of messages (REQUEST, REPLY and RELEASE)

- Every site  $S_i$  keeps a queue to store critical section requests ordered by their timestamps. **request\_queue** 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.

Algorithm:

- To enter Critical section:**
  - When a site  $S_i$  wants to enter the critical section, it sends a request message **Request( $ts_i, i$ )** 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( $ts_i, i$ )** 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 ( $ts_i, i$ ) 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

- $(N-1)$  request messages
- $(N-1)$  reply messages
- $(N-1)$  release messages

**Drawbacks of Lamport's Algorithm:**

- Unreliable approach:** failure of any one of the processes will halt the progress of entire system.
- High message complexity:** Algorithm requires  $3(N-1)$  messages per critical section invocation.

**Performance:**

- Synchronization delay is equal to maximum message transmission time
- It requires  $3(N-1)$  messages per CS execution.
- Algorithm can be optimized to  $2(N-1)$  messages by omitting the REPLY message in some situations.

An optimization in performance in  $3(N-1)$

In Lamport's algorithm, when one site asks for permission (let's call it  $S_i$ ), and another site ( $S_j$ ) already asked earlier with a higher timestamp,  $S_j$  doesn't need to reply to  $S_i$ .  $S_i$  can figure out that  $S_j$  doesn't have any earlier requests pending.

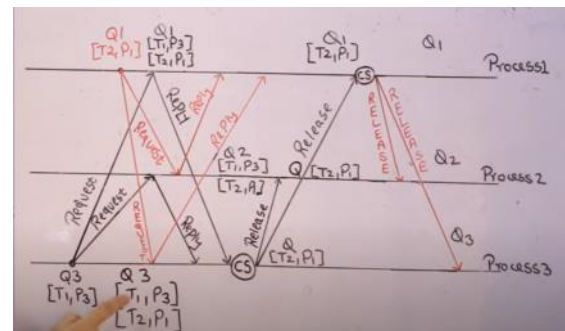
This optimization makes Lamport's algorithm work more efficiently, needing fewer messages (between  $3(N-1)$  and  $2(N-1)$  messages) each time a site wants to do something important, like entering a critical section.

**Theorem: Lamport's algorithm achieves mutual exclusion.**

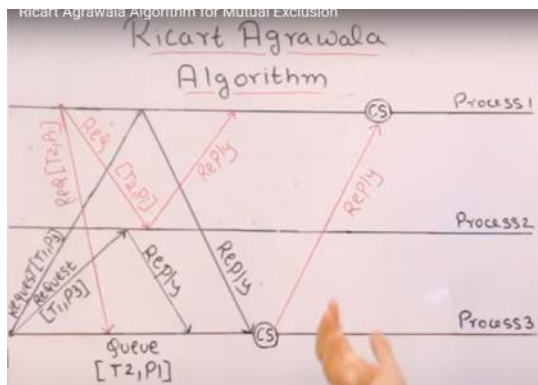
**Proof:**

Certainly! In Lamport's algorithm for achieving mutual exclusion:

- Suppose two sites,  $S_i$  and  $S_j$ , are trying to enter the critical section at the same time.
- The proof shows that if both  $S_i$  and  $S_j$  are in the critical section concurrently, it leads to a contradiction.
- The contradiction arises from the fact that  $S_i$ 's request with a smaller timestamp is at the top of both  $S_i$  and  $S_j$ 's request queues simultaneously.
- The contradiction demonstrates that Lamport's algorithm effectively prevents two sites from being in the critical section at the same time, ensuring mutual exclusion in a distributed system.



## Ricart Agrawala Algorithm



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.

**Theorem:** Ricart-Agrawala algorithm achieves mutual exclusion.  
**Proof:**

- Proof is by contradiction. Suppose two sites  $S_i$  and  $S_j$  are executing the CS concurrently and  $S_i$ 's request has higher priority than the request of  $S_j$ . Clearly,  $S_j$  received  $S_i$ 's request after it has made its own request.
- Thus,  $S_j$  can concurrently execute the CS with  $S_i$  only if  $S_j$  returns a **REPLY** to  $S_i$  (in response to  $S_i$ 's request) before  $S_i$  exits the CS.
- However, this is impossible because  $S_j$ 's request has lower priority. Therefore, Ricart-Agrawala algorithm achieves mutual exclusion.

**Message Complexity:** Ricart-Agrawala algorithm requires invocation of  $2(N - 1)$  messages per critical section execution. These  $2(N - 1)$  messages involves

Q4. [CO2 2+ 2 marks] a) In a system, each process may use critical section many times before another process requires it. Explain why Ricart-Agrawala ME algorithm is inefficient for this case and describe how its performance be improved.

b) A DS may have multiple independent critical regions (CR). Imagine Process 0 wants to enter CR A and process 1 wants to enter CR B. Can Ricart-Agrawala ME algorithm lead to deadlocks?

a) Ricart-Agrawala can be inefficient when a process uses the critical section many times due to frequent message exchanges. To improve:

1. **Batching Requests:**

- Processes can batch critical section requests to reduce message frequency.

2. **Local Permission Mechanism:**

- Allow processes to locally grant permission to enter the critical section without having to obtain global agreement from all other processes.

Processes can maintain a counter to keep track of the number of times they have entered the critical section and release the resources accordingly.

b) Yes, the Ricart-Agrawala algorithm can lead to deadlocks when multiple processes are trying to enter different critical regions simultaneously. If two processes request permission for different critical regions concurrently, a circular wait condition may occur, resulting in a deadlock. Consider modifying the algorithm or exploring alternatives for scenarios with multiple independent critical regions.

Q5 [CO2 2 Marks] Raymond's tree based algorithm does not order CS requests based on time. Explain whether the algorithm is fair or not

Q5 [CO2 2 Marks] Raymond's tree based algorithm does not order CS requests based on time. Explain whether the algorithm is fair or not

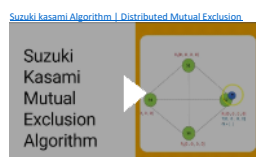
Q5 [CO2 2 Marks] Raymond's tree based algorithm does not order CS requests based on time. Explain whether the algorithm is fair or not

Q1 [CO1, Marks 3] What problem of Lamport's clocks do vector clocks solve? Give 2 examples of events concurrent with the vector timestamp (2, 8, 4).

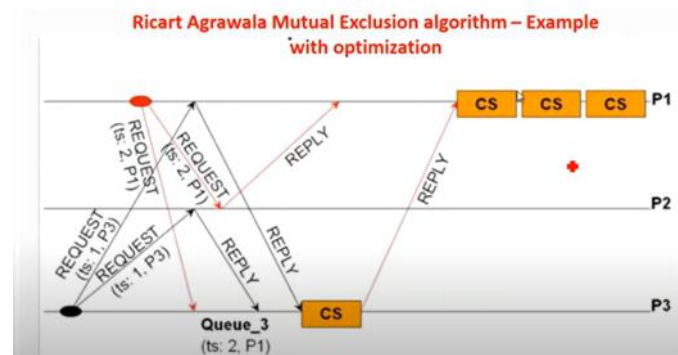
Q2 [CO2, Marks 3+3] a) You want to select a distributed mutual exclusion algorithm for the scenario: You have 20 low-cost, unreliable computers with varying speeds connected to the same Ethernet switch. When a request for processing a job arrives, it should be assigned to one of these computers, which then processes the job. Exactly ONE computer should handle each request. Which distributed mutual exclusion algorithm would you use? Justify.  
b) Compare Lamport's distributed mutual exclusion algorithm with Ricart Agrawala algorithm in terms of correctness, fairness, deadlock freedom and message complexity.

Vector clocks address the lack of causality information and false concurrency in Lamport's clocks. Two events concurrent with the vector timestamp (2, 8, 4) could be (2, 8, 7) and (2, 8, 2), where the first two entries match, indicating events in processes 1 and 2 are in the same state.

For 20 low-cost, unreliable computers connected to the same Ethernet switch, the **Token Ring Algorithm** is suitable. It provides sequential access to the critical section through a circulating token, ensuring exactly one computer processes each job request. The algorithm is simple, has low overhead, and is well-suited for scenarios with such computer characteristics.



Compare Lamport's distributed mutual exclusion algorithm with Ricart Agrawala algorithm in terms of correctness, fairness, deadlock freedom and message complexity.



Q4. [CO2 2+ 2 marks] a) In a system, each process may use critical section many times before another process requires it. Explain why Ricart-Agrawala ME algorithm is inefficient for this case and describe how its performance be improved

### Suzuki Kasami Algorithm

- If a site attempting to enter the Critical Section (CS) but does not have the token, it broadcasts a REQUEST message for the token to all other sites.
- On receipt of request message, the site that possess the token
  - If the site is not executing the CS, then it sends the token to the requesting site.
  - Otherwise, sends once it exits CS.
- A site holding the token can enter into the CS repeatedly until it sends the token to some other site.

05:09

### Data structures used in SK algorithm

- $R[1..N]$  – request queue maintained at each site  $S_i$  of size  $n$  each index corresponds to every other site of DS.
- $T[n]$  – token array of size  $n$  to maintain the number of times the particular site requested the token.
- $Q$  – Token request queue consists of the site IDs of simultaneous requests from different sites.
- Messages :
  - REQUEST( $S_i, n$ ) – site  $S_i$  request for token for  $n^{\text{th}}$  time.
  - PREVIOUS( $T$ ) – granting token message

08:42

### Distinguishing outdated and current REQUEST messages

- A site  $S_i$  keeps an array of integers  $R_i[1..N]$  where  $R_i[j]$  is the largest sequence number received so far in REQUEST message of site  $S_j$ .

A REQUEST message of site  $S_i$  has the form :

REQUEST( $S_i, n$ )

- where  $n$  is the sequence number indicating that site  $S_i$  is requesting its  $n^{\text{th}}$  CS execution.

A REQUEST( $S_i, n$ ) message by site  $S_i$  is considered

$R_i[S_i] < n$

09:45

### Determining outstanding requests for CS

- The token consists of a queue of requesting sites  $Q$ , and an array of integers  $T[1..N]$ , where  $T[i]$  is the sequence number of the request that site  $S_i$  executed most recently.

$R_i[S_i] < T[i] + 1$  then there is no outstanding request

### Suzuki Kasami Algorithm

#### Requesting Stage:

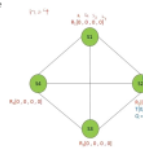
- Site  $S_i$  wants to enter into CS.
- Sends a token request message REQUEST( $S_i, n$ ) to all other sites.
- On receiving the Request message from  $S_i$ , site  $S_j$  update its request  $R_j$  with  $\max(R_j, n)$ .
- On receiving  $S_i$ 's request, the 2 conditions to be checked:
  - Check for outdated request  
if  $R_i[S_i] < n$ , then the request is not an outdated request
  - Determining outstanding request  
if  $R_i[S_i] < T[i] + 1$ , then there is no outstanding request

#### Releasing Stage:

- On Exiting CS.
- Set Token array  $T[i] = R[i]$
- If token  $Q$  is not empty, then the token is sent to the site in the top of token  $Q$ .

13:04

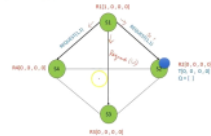
Initial State



jo request karta hai uska token increase hota hai

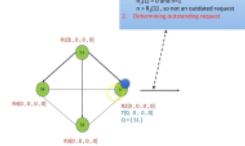
13:54

$S_1$  wants to enter CS and broadcast token request REQUEST( $S_1, 1$ )



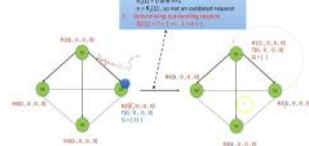
14:39

On receiving  $S_1$ 's request



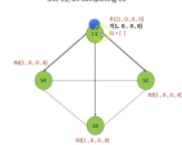
15:48

On receiving  $S_1$ 's request

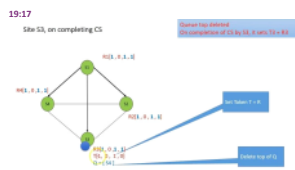
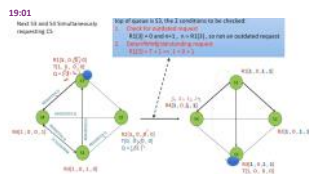
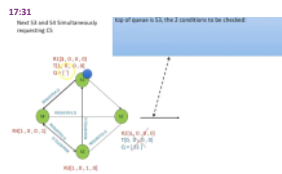
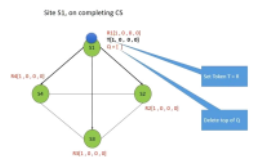


16:35

Site  $S_1$ , on completing CS



17:10



From <<https://www.youtube.com/watch?v=lythl-Gajic>>

#### Correctness

Mutual exclusion is guaranteed because there is only one token in the system and a site holds the token during the CS execution.

**Theorem:** A requesting site enters the CS in finite time.

**Proof:**

- Token request messages of a site  $S_i$  reach other sites in finite time.
- Since one of these sites will have token in finite time, site  $S_i$ 's request will be placed in the token queue in finite time.
- Since there can be at most  $N - 1$  requests in front of this request in the token queue, site  $S_i$  will get the token and execute the CS in finite time.

#### Performance

- No message is needed and the synchronization delay is zero if a site holds the idle token at the time of its request.
- If a site does not hold the token when it makes a request, the algorithm requires  $N$  messages to obtain the token. Synchronization delay in this algorithm is 0 or  $T$ .

00:52

#### Raymond Tree Based Algorithm

- Token-based
- Sites are logically arranged as a directed tree such that the edges of the tree are having direction toward the site (Tree Root) that has the token.
- Every site has a local variable called holder that points to an immediate neighbour node on a directed path to the root node. Holder of root is self pointed.
- Every site keeps a FIFO queue called request Q that stores the requests sent by neighbouring site to this site.

Tree data structure is used

03:25

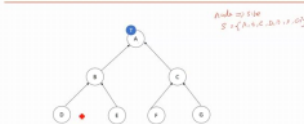
#### The Algorithm : Requesting the Critical Section

When a site wants to enter CS,

- it sends REQUEST to its immediate parent, it then update its request Q.
- When the parent receives this REQUEST message, it update its Request Q and forwarded the REQUEST message to its parent.
- This directed REQUEST forwarding continues till root reached.
- On receiving the REQUEST message, the root node sends the token to the site from which it received REQUEST with PRIVILEGE message and sets its holder value to the requesting site. Hence direction changed.
- When a site receives the token, it deletes the top entry from request Q, sends the token to the indicated site and sets its holder variable to point the target site.

04:07

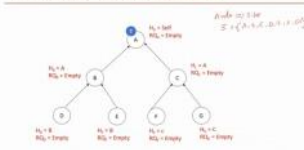
#### Raymond-tree Example:



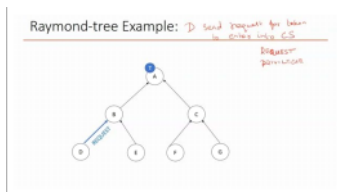
B and C is directed toward A also B and C is neighbour of A

06:20

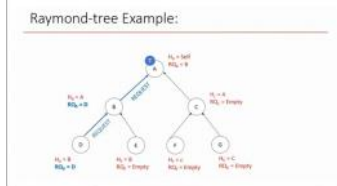
#### Raymond-tree Example:



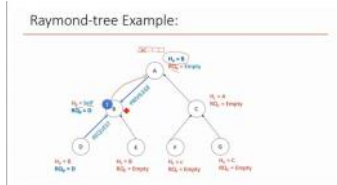
07:52



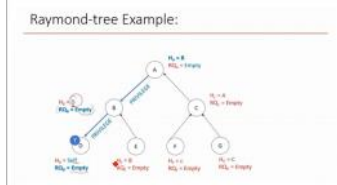
09:15



11:13



12:38



12:44

**The Algorithm : Requesting the Critical Section**

When a site wants to enter CS,

1. It sends REQUEST to its immediate parent, It then update its request Q.
2. When the parent receives the REQUEST message, it update its Request Q and forwarded the REQUEST message to its parent.
3. This directed REQUEST forwarding continues till Root reached.
4. On receiving the REQUEST message, the root node sends the token to the site from which it received REQUEST with PRIVILEGE message and sets its holder value to the requesting site. Hence direction changed.
5. When a site receives the token, it deletes the top entry from request Q, sends the token to the indicated site and sets its holder variable to point the target site.

14:58

**Executing the CS**

6. A site enters CS when it receives the token and its own entry is at the top of the request Q, then deletes the top entry from its own request Q and enters the CS.

15:06

**Releasing the CS**

After executing CS,

7. If its request Q is non-empty, then deletes the top entry from queue and sends the token to that site, and sets the holder value to the target node.
8. If the request Q is empty, then the site sends a REQUEST message to the site which is pointed at by the holder variable.

From <<https://www.youtube.com/watch?v=1PFu6VFXk>>

29:08

**Singhal heuristic algorithm**

A heuristically added algorithm to achieve mutual exclusion in distributed systems is presented which has better performance characteristics than previously proposed algorithms. The algorithm makes use of state information, which is defined as the set of states of mutual exclusion processes in the system. Each site maintains information about the state of other sites and uses it to deduce a subset of sites likely to have the token. Consequently, the number of messages exchanged for a critical section invocation is a random variable between 0 and n (n is the number of sites in the system). It is shown that the Algorithm achieves mutual exclusion and is free from deadlock and starvation.

Instead of Broadcast, each site maintains information about the state of next site.

State of Site:

- R: Requesting the CS
- E: Executing the CS
- H: Holding the CS, site
- N: Normal, None of the above.

29:53

P1 is holding the token so, currently empty and other states are in request mode.

32:55

In this diagram the P1 is requesting for token from P2 because P1 checks that it is holding the token and the chances are more to get the token. The states of other processes are idle.

It will check for the conditions like outstand request or number and state of the process. If all conditions satisfy then it will provide the token to P1.

From <<https://www.youtube.com/watch?v=afK460F184>>

11:11

From <<https://www.youtube.com/watch?v=5Q9-n5PMIM>>