Distributed Mutual Exclusion

Mutual exclusion

- Concurrent access of processes to a shared resource or data is executed in mutually exclusive manner.
- Only one process is allowed to execute the critical section (CS) at any given time.
- In a distributed system, shared variables (semaphores) or a local kernel cannot be used to implement mutual exclusion.
- Message passing is the sole means for implementing distributed mutual exclusion.

Distributed Mutual Exclusion Algorithms

- These algorithms must deal with unpredictable message delays and incomplete knowledge of the system state.
- Two basic approaches for distributed mutual exclusion:
  1. Non-token based approach
  2. Token based approach

Non-token based approach

- Two or more successive rounds of messages are exchanged among the sites to determine which site will enter the CS next.
- A site/process can enter a critical section when an assertion (condition) becomes true.
- Algorithm should ensure that the assertion will be true in only one site/process.

Token based approach

- A unique token (a known, unique message) is shared among the sites.
- A site is allowed to enter its CS if it possesses the token.
- Mutual exclusion is ensured because the token is unique.
- Need to take care of conditions such as loss of token, crash of token holder, possibility of multiple tokens, etc.

System Model

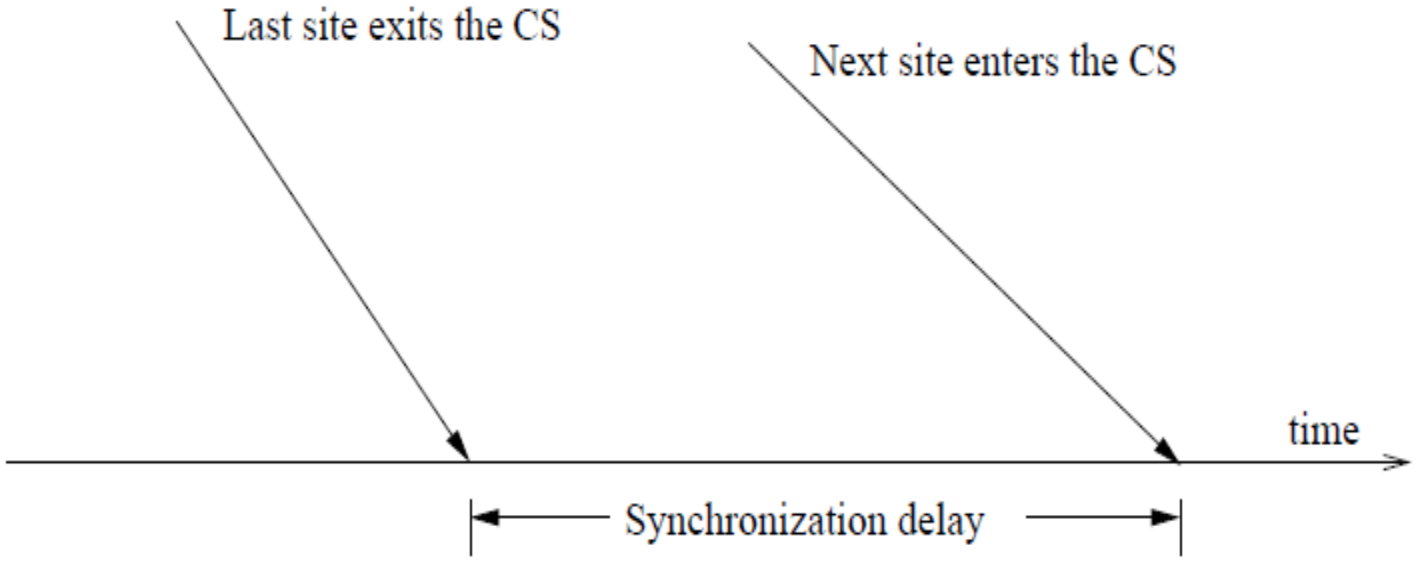
- The system consists of N sites, S_1, S_2, ..., S_N. It is assumed that a single process is running on each site. The process at site S_i is denoted by p_i .
- A site can be in one of the following three states:
  - Requesting CS: blocked until granted access, cannot make additional requests for CS.
  - Executing CS: using the CS.
  - Idle: action is outside the site. In token-based approaches, *idle* site can have the token.
- In token-based algorithms, a site can also be in a state where a site holding the token is executing outside the CS (called 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.

Mutual Exclusion : Requirements

- **Freedom from deadlocks**: two or more sites should not endlessly wait on conditions/messages that never become true/arrive.
- **Freedom from starvation**: No indefinite waiting.
- **Fairness**: Order of execution of CS follows the order of the requests for CS. (equal priority).
- **Fault tolerance**: recognize “faults”, reorganize, continue. (e.g., loss of token).

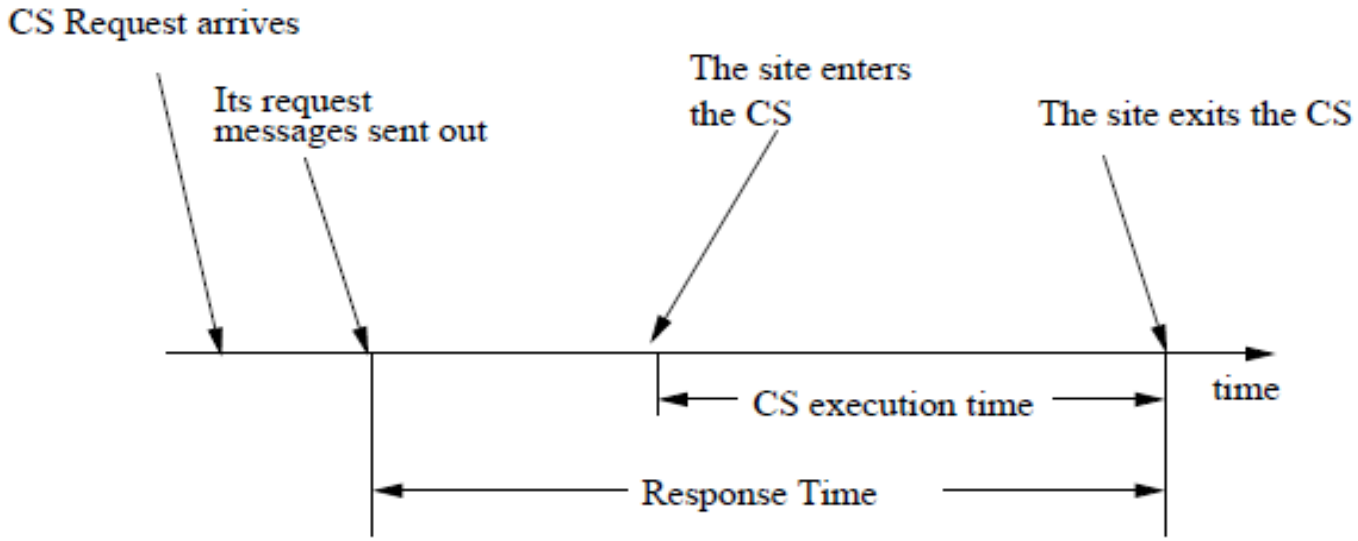
Performance Metrics

- The performance is generally measured by the following four metrics:
  1. **Message complexity**: The number of messages required per CS execution by a site.
  2. **Synchronization delay**: After a site leaves the CS, it is the time required and before the next site enters the CS



Performance Metrics

3. **Response time**: The time interval a request waits for its CS execution to be over after its request messages have been sent out
4. **System throughput**: The rate at which the system executes requests for the CS.
   - system throughput=1/(SD+E)
   - where SD is the synchronization delay and E is the average critical section execution time.



Performance Metrics

Low and High Load Performance:

- Performance of mutual exclusion algorithms are often studied under two special loading conditions, viz., “low load” and “high load”.
- The load is determined by the arrival rate of CS execution requests.
- 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.

Simple Solution to DME

- Control site: grants permission for CS execution.
- A site sends REQUEST message to control site.
- Controller grants access one by one.
- Synchronization delay (2T) : A site release CS by sending message to controller and controller sends permission to another site.
- System throughput: 1/(2T + E). If synchronization delay is reduced to T, throughput doubles.
- Controller becomes a bottleneck, congestion can occur.

Non-token Based Algorithms

Notations:

- S_i: Site I
- R_i: Request set, containing the ids of all S_i 's from which permission must be received before accessing CS.
- Non-token based approaches use time stamps to order requests for CS.
- Logical clocks are maintained and updated as per Lamport's scheme
- Smaller time stamps get priority over larger ones.

Lamport's Algorithm

- Requests for CS are executed in the increasing order of timestamps and time is determined by logical clocks.
- Every site Si keeps a queue, $request\_queue_i$, which contains mutual exclusion requests ordered by their timestamps.
- $R_i = \{S_1, S_2, \ldots, S_n\}$, i.e., all sites.
- This algorithm requires communication channels to deliver messages the FIFO order.

The Algorithm

Requesting the critical section:

- When a site S_i wants to enter the CS, it broadcasts a REQUEST(ts_i , i) message to all other sites and places the request on $request\_queue_i$. ((ts_i , i) denotes the timestamp of the request.)
- When a site S_j receives the REQUEST(ts_i , i) message from site S_i ,places site S_i 's request on $request\_queue_j$ and it returns a time stamped REPLY message to S_i .

The Algorithm

Executing the critical section:

- Site Si enters the CS when the following two conditions hold:
  - **L1**: S_i has received a message with timestamp larger than (ts_i , i) from all other sites.
  - **L2**: S_i ’s request is at the top of $request\_queue_i$

The Algorithm

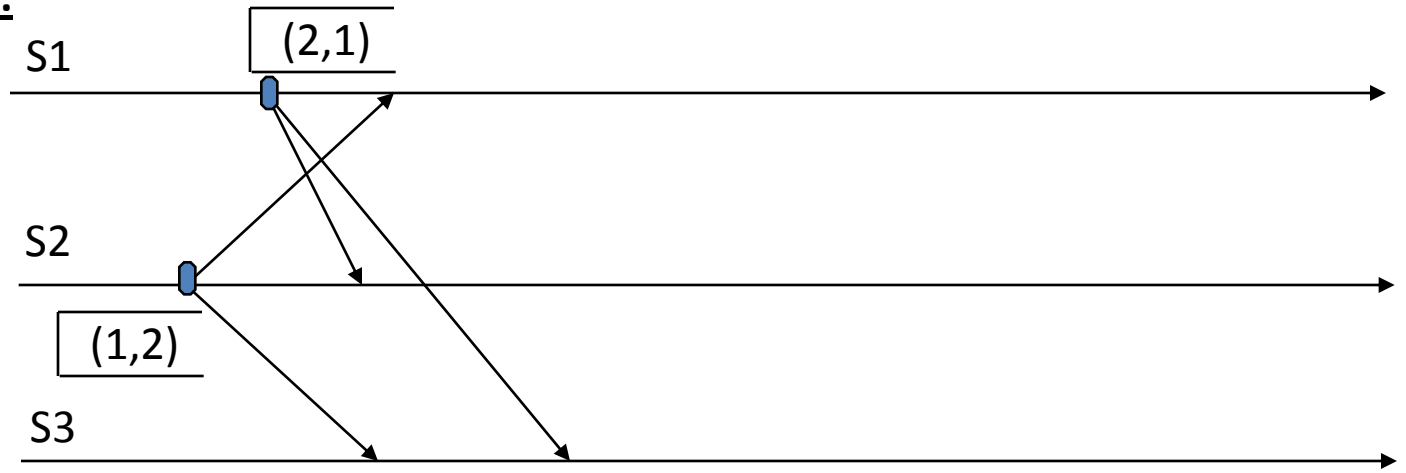
Releasing the critical section:

- Site S_i , upon exiting the CS, removes its request from the top of its request queue and broadcasts a time stamped RELEASE message to all other sites.
- When a site S_j receives a RELEASE message from site S_i, it removes S_i 's request from its request queue.

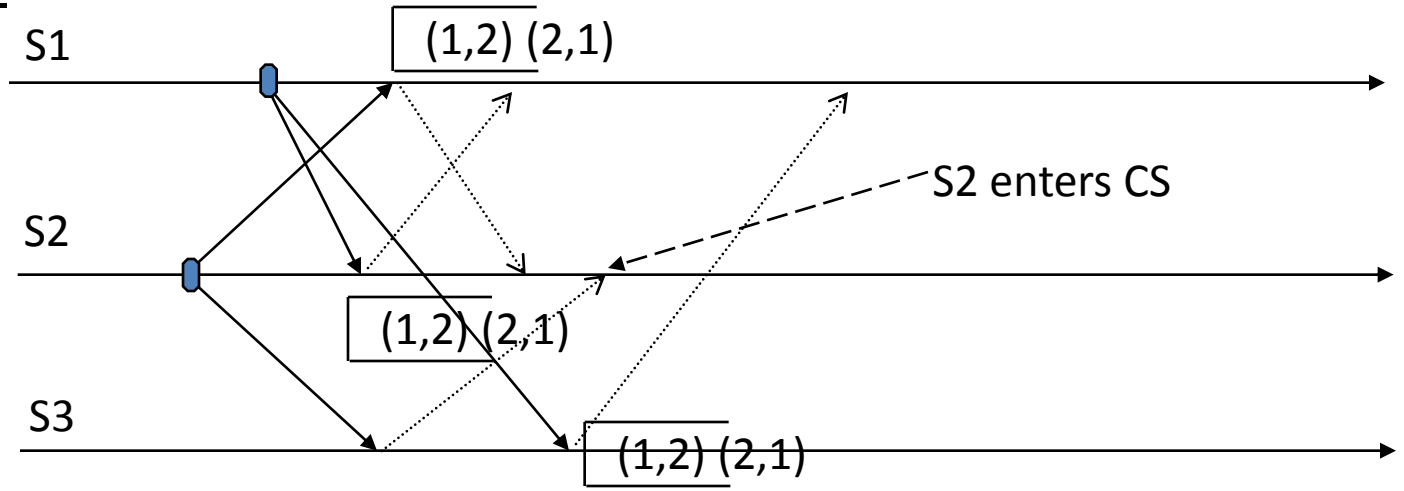
- When a site removes a request from its request queue, its own request may come at the top of the queue, enabling it to enter the CS.

An Example

Step 1:

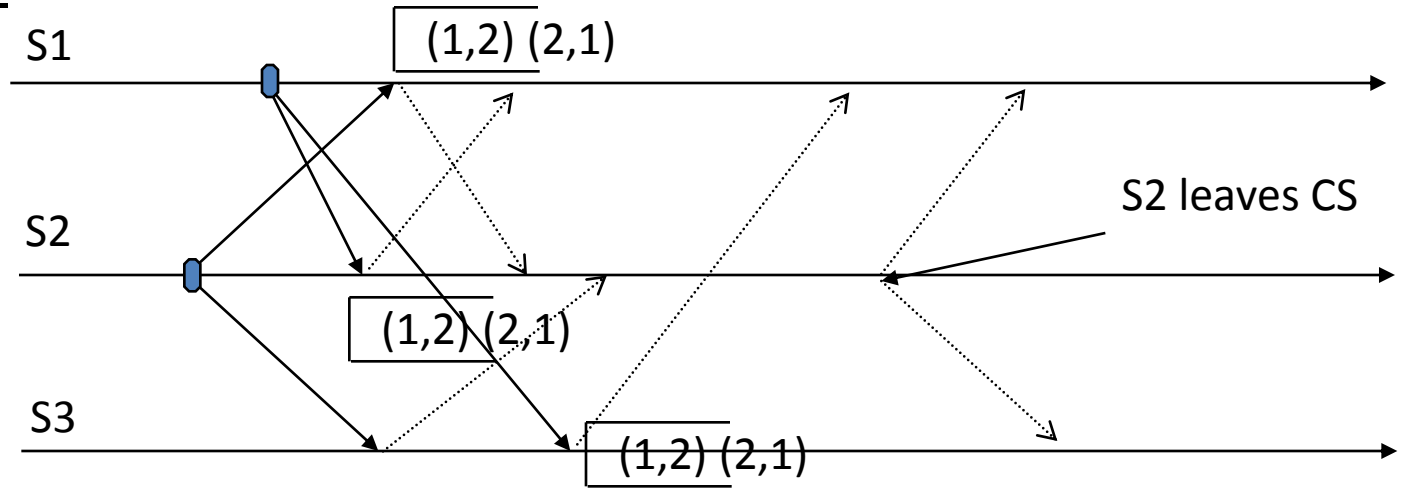


Step 2:

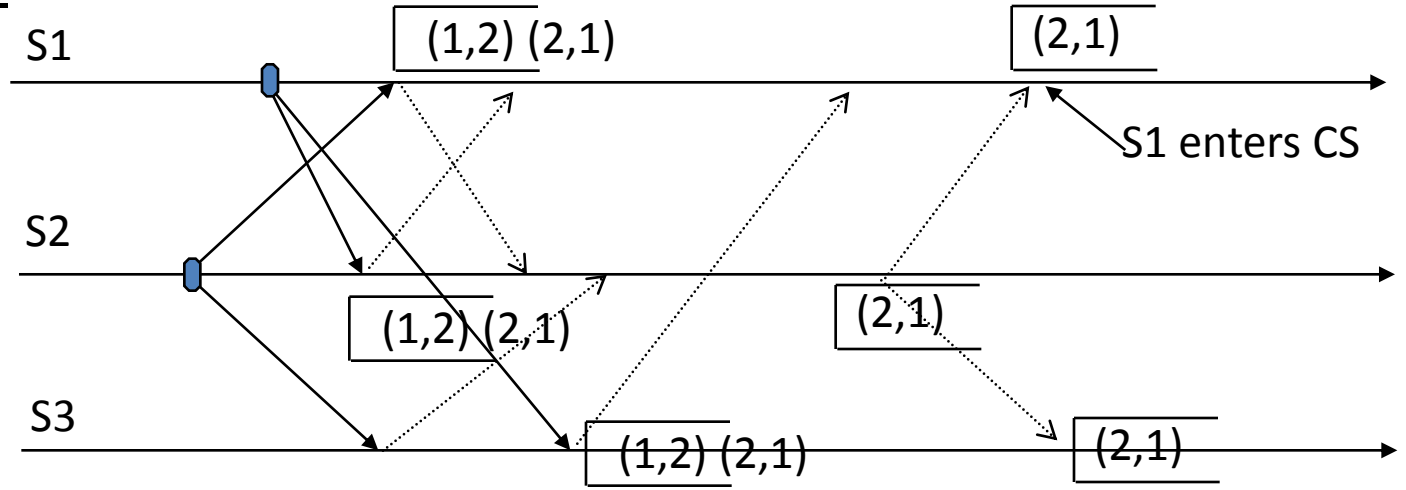


An Example (Cont...)

Step 3:



Step 4:



Performance

- For each CS execution, Lamport's algorithm requires (N − 1) REQUEST messages, (N − 1) REPLY messages, and (N − 1) RELEASE messages.
- Thus, Lamport's algorithm requires 3(N − 1) messages per CS invocation.
- Synchronization delay in the algorithm is T.

Optimization

- In Lamport's algorithm, REPLY messages can be omitted in certain situations.
- For example, if site S_j receives a REQUEST message from site S_i after it has sent its own REQUEST message with timestamp higher than the timestamp of site S_i 's request, then site S_j need not send a REPLY message to site S_i .
- This is because when site S_i receives site S_j 's request with timestamp higher than its own, it can conclude that site S_j does not have any smaller timestamp request which is still pending.
- With this optimization, Lamport's algorithm requires between 3(N - 1) and 2(N - 1) messages per CS execution.

Correctness

Theorem: Lamport's algorithm achieves mutual exclusion.

Proof:

- Proof is by contradiction. Suppose two sites S_i and S_j are executing the CS concurrently. For this to happen conditions L1 and L2 must hold at both the sites concurrently.
- This implies that at some instant in time, say ***t***, both S_i and S_j have their own requests at the top of their request queues and condition L1 holds at them. Without loss of generality, assume that S_i 's request has smaller timestamp than the request of S_j .
- From condition L1 and FIFO property of the communication channels, it is clear that at instant ***t*** the request of S_i must be present in $request\_queue_j$ when S_j was executing its CS. This implies that S_j 's own request is at the top of its own request queue when a smaller timestamp request, S_i 's request, is present in the $request\_queue_j$ – a contradiction!

Correctness

Theorem: Lamport's algorithm is fair.

Proof:

- The proof is by contradiction. Suppose a site S_i 's request has a smaller timestamp than the request of another site S_j and S_j is able to execute the CS before S_i .
- For S_j to execute the CS, it has to satisfy the conditions L1 and L2. This implies that at some instant in time say ***t***, S_j has its own request at the top of its queue and it has also received a message with timestamp larger than the timestamp of its request from all other sites.
- But request queue at a site is ordered by timestamp, and according to our assumption S_i has lower timestamp. So S_i's request must be placed ahead of the S_j 's request in the $request\_queue_j$. This is a contradiction!

Ricart-Agrawala Algorithm

- The Ricart-Agrawala algorithm assumes the communication channels are FIFO.
- The algorithm uses two types of messages: REQUEST and REPLY.
- A process sends a REQUEST message to all other processes to request their permission to enter the critical section. A process sends a REPLY message to a process to give its permission to that process.
- Processes use Lamport-style logical clocks to assign a timestamp to critical section requests and timestamps are used to decide the priority of requests.
- Each process p_i maintains the Request-Deferred array, RD_i , the size of which is the same as the number of processes in the system.
- Initially, $\forall i \; \forall j : RD_i[j]=0$. Whenever p_i defer the request sent by p_j , it sets $RD_i[j]=1$ and after it has sent a REPLY message to p_j , it sets $RD_i[j]=0$.

The Algorithm

Requesting the critical section:

- When a site S_i wants to enter the CS, it broadcasts a time stamped REQUEST message to all other sites.
- When site S_j receives a REQUEST message from site S_i , it sends a REPLY message to site S_i , if
  - Site S_j is neither requesting nor executing the CS
  - The site S_j is requesting and S_i 's request's timestamp is smaller than site S_j 's own request's timestamp.
  - Otherwise, the reply is deferred and S_j sets $RD_j[i]=1$

The Algorithm

Executing the critical section:

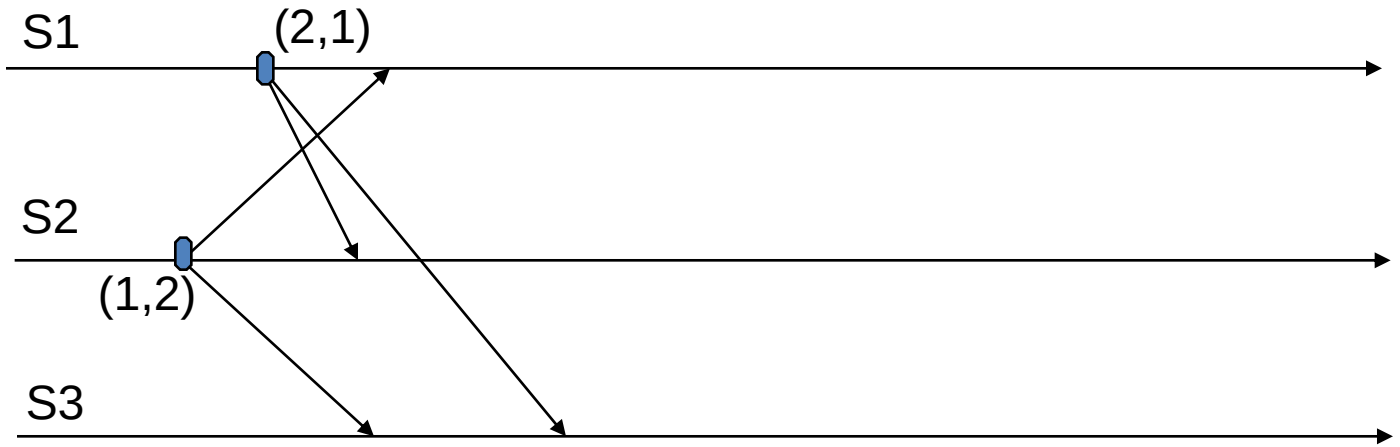
- Site S_i enters the CS after it has received a REPLY message from every site, it has sent a REQUEST message to.

Releasing the critical section:

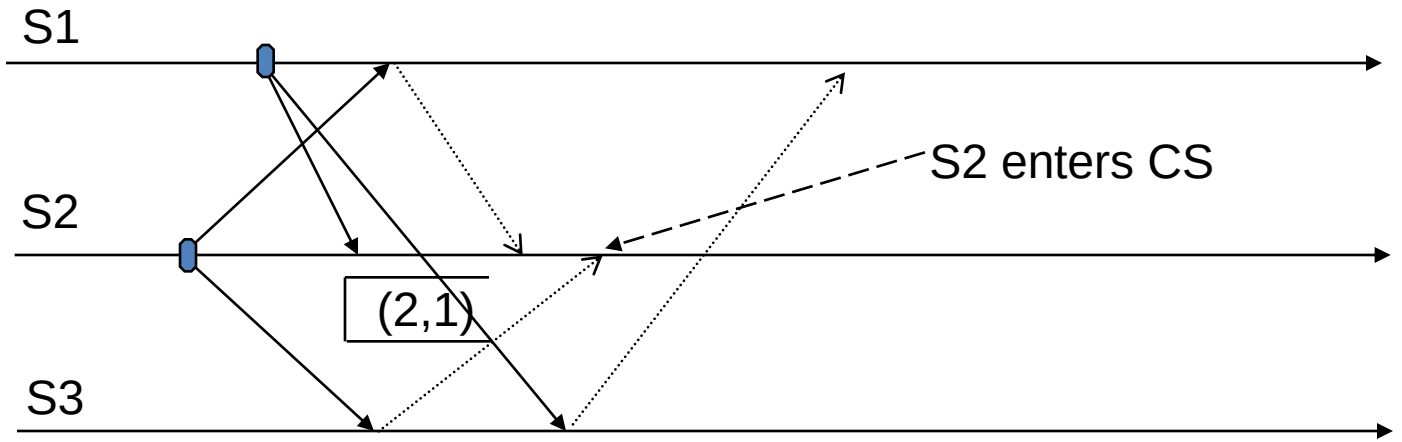
- When site S_i exits the CS, it sends all the deferred REPLY messages: $\forall j$ if $RD_i[j]=1$, then send a REPLY message to S_j and set $RD_i[j]=0$

An Example

Step 1:

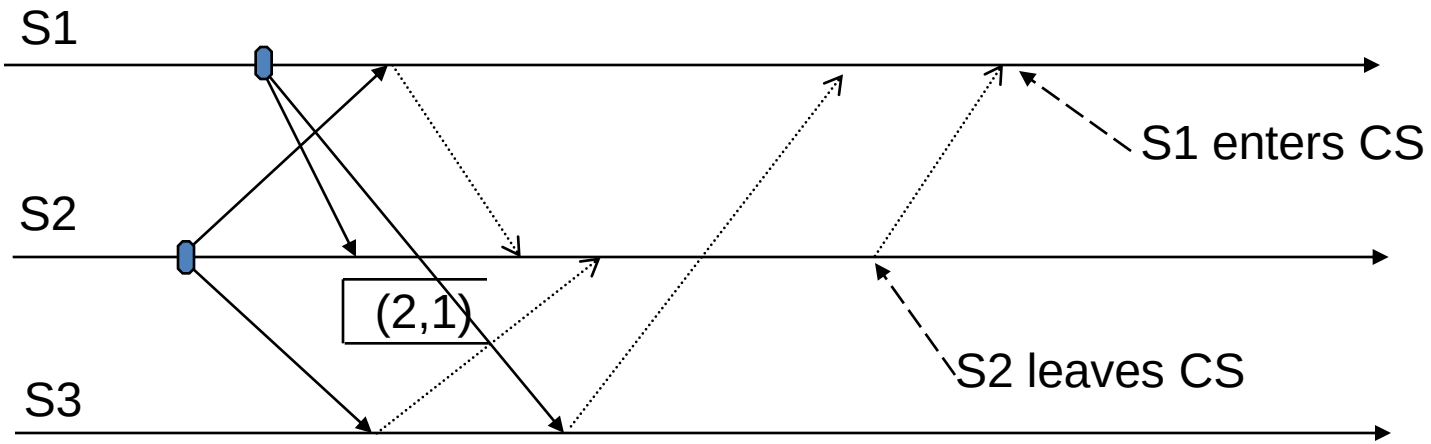


Step 2:



An Example (Cont…)

Step 3:



Performance

- For each CS execution, Ricart-Agrawala algorithm requires (N − 1) REQUEST messages and (N − 1) REPLY messages.
- Thus, it requires 2(N − 1) messages per CS execution.
- Synchronization delay in the algorithm is T.

Optimization

When S_i receives REPLY message from S_j -> authorization to access CS till

- S_j sends a REQUEST message and S_i sends a REPLY message.
- Access CS repeatedly till then.
- A site requests permission from dynamically varying set of sites: 0 to 2(N-1) messages.

Correctness

Theorem: Ricart-Agarwala 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_i received S_j ’s request after it has made its own request.
- Thus, S_j can concurrently execute the CS with S_i only if S_i returns a REPLY to S_j (in response to S_j ’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.

$\sqrt{N}$

Maekawa's Algorithm

- A site requests permission only from a subset of sites.
- Request set of sites s_i & s_j: R_i, R_j such that R_i and R_j will have at least one common site (S_k). S_k mediates conflicts between R_i and R_j.
- A site can send only one REPLY message at a time, i.e., a site can send a REPLY message only after receiving a RELEASE message for the previous REPLY message.

Maekawa's Algorithm

- The request sets for sites in Maekawa's algorithm are constructed to satisfy the following conditions:
  - M1: $(\forall i\ \forall j : i \neq j , 1 \leq i , j \leq N :: R_i \cap R_j \neq \phi)$
  - M2: $(\forall i : 1 \leq i \leq N :: S_i \in R_i)$
  - M3: $(\forall i : 1 \leq i \leq N :: |R_i| = K)$
  - M4: Any site S_j is contained in K number of R_i s, $1 \leq i , j \leq N$.
- Maekawa established the following relation
- $N = K(K - 1) + 1 \Rightarrow |R_i| = \sqrt{N}$

Maekawa's Algorithm

- Conditions M1 and M2 are necessary for correctness; whereas conditions M3 and M4 provide other desirable features to the algorithm.
- Condition M3 states that the size of the requests sets of all sites must be equal implying that all sites should have to do equal amount of work to invoke mutual exclusion.
- Condition M4 enforces that exactly the same number of sites should request permission from any site implying that all sites have "equal responsibility" in granting permission to other sites.

Request Subsets

- Example 1.
  - If k = 2; N = 3 then
  - R1 = {1, 2}; R3 = {1, 3}; R2 = {2, 3}
- Example 2.
  - If k = 3; N = 7 then
  - R1 = {1, 2, 3}; R4 = {1, 4, 5}; R6 = {1, 6, 7};
  - R2 = {2, 4, 6}; R5 = {2, 5, 7}; R7 = {3, 4, 7};
  - R3 = {3, 5, 6}

The Algorithm

- Requesting the critical section
  a) A site S_i requests access to the CS by sending REQUEST(i) messages to all sites in its request set R_i .
  b) When a site S_j receives the REQUEST(i) message, it sends a REPLY(j) message to S_i provided it hasn't sent a REPLY message to a site since its receipt of the last RELEASE message. Otherwise, it queues up the REQUEST(i) for later consideration.

The Algorithm

- Executing the critical section
  c) Site S_i executes the CS only after it has received a REPLY message from every site in R_i.

The Algorithm

- Releasing the critical section
  d) After the execution of the CS is over, site S_i sends a RELEASE(i) message to every site in R_i.
  e) When a site S_j receives a RELEASE(i) message from site S_i, it sends a REPLY message to the next site waiting in the queue and deletes that entry from the queue. If the queue is empty, then the site updates its state to reflect that it has not sent out any REPLY message since the receipt of the last RELEASE message.

Performance

- Since the size of a request set is √N, an execution of the CS requires √N REQUEST, √N REPLY, and √N RELEASE messages, resulting in 3√N messages per CS execution.
- Synchronization delay in this algorithm is 2T. This is because after a site S_i exits the CS, it first releases all the sites in R_i and then one of those sites sends a REPLY message to the next site that executes the CS.

Correctness

- **Theorem:** Maekawa's algorithm achieves mutual exclusion.
- **Proof:**
- Proof is by contradiction. Suppose two sites S_i and S_j are concurrently executing the CS.
- This means site S_i received a REPLY message from all sites in R_i and concurrently site S_j was able to receive a REPLY message from all sites in R_j .
- If $R_i \cap R_j = \{S_k\}$, then site S_k must have sent REPLY messages to both S_i and S_j concurrently, which is a contradiction.

Problem of Deadlock

- Maekawa's algorithm can deadlock because a site is exclusively locked by other sites and requests are not prioritized by their timestamps.
- Assume three sites S_i , S_j , and S_k simultaneously invoke mutual exclusion.
- Suppose $R_i \cap R_j = \{S_{ij}\}$, $R_j \cap R_k = \{S_{jk}\}$, and $R_k \cap R_i = \{S_{ki}\}$.
- Consider the following scenario:
  - S_{ij} has been locked by S_i (forcing S_j to wait at S_{ij}).
  - S_{jk} has been locked by S_j (forcing S_k to wait at S_{jk}).
  - S_{ki} has been locked by S_k (forcing S_i to wait at S_{ki}).
- This state represents a deadlock involving sites S_i , S_j , and S_k.

Handling Deadlocks

- Maekawa's algorithm handles deadlocks by requiring a site to yield a lock if the timestamp of its request is larger than the timestamp of some other request waiting for the same lock.

- A site suspects a deadlock (and initiates message exchanges to resolve it) whenever a higher priority request arrives and waits at a site because the site has sent a REPLY message to a lower priority request.

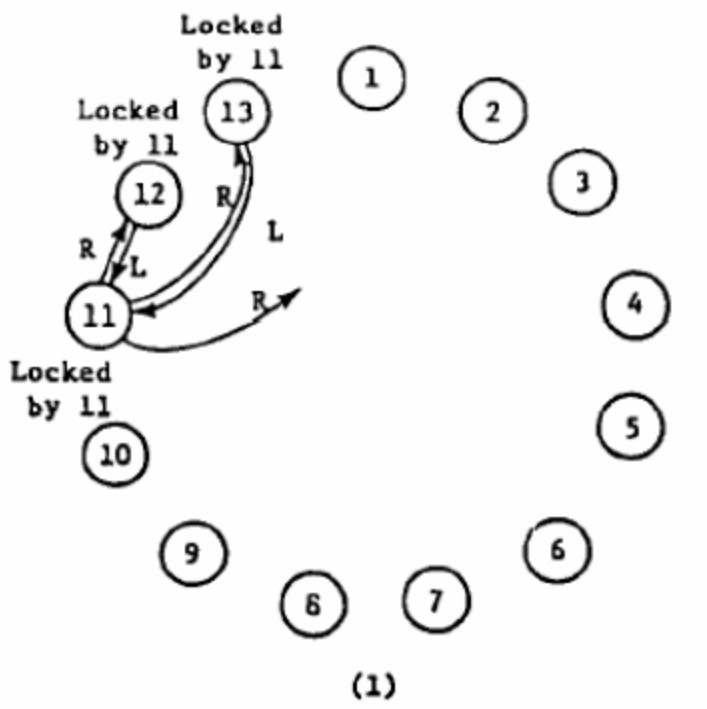
Handling Deadlocks

- Deadlock handling requires three types of messages:
  - **FAILED:** A FAILED message from site S_i to site S_j indicates that S_i can not grant S_j 's request because it has currently granted permission to a site with a higher priority request.
  - **INQUIRE:** An INQUIRE message from S_i to S_j indicates that S_i would like to find out from S_j if it has succeeded in locking all the sites in its request set.
  - **YIELD:** A YIELD message from site S_i to S_j indicates that S_i is returning the permission to S_j (to yield to a higher priority request at S_j).

Handling Deadlocks

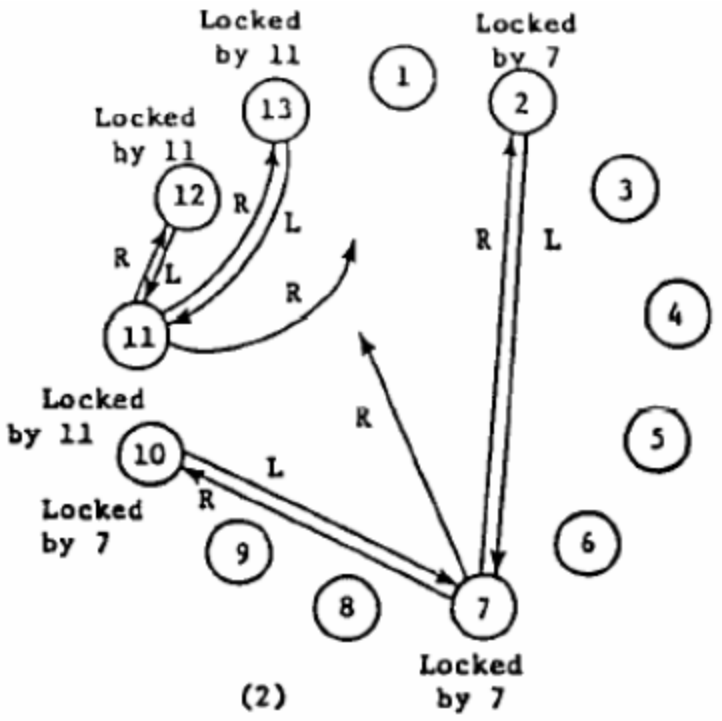
- Maekawa's algorithm handles deadlocks as follows:
  - When a REQUEST(ts, i) from site S_i blocks at site S_j because S_j has currently granted permission to site S_k , then S_j sends a FAILED(j) message to S_i if S_i 's request has lower priority. Otherwise, S_j sends an INQUIRE(j) message to site S_k .
  - In response to an INQUIRE(j) message from site S_j , site S_k sends a YIELD(k) message to S_j provided S_k has received a FAILED message from a site in its request set or if it sent a YIELD to any of these sites, but has not received a new GRANT from it.
  - In response to a YIELD(k) message from site S_k , site S_j assumes as if it has been released by S_k , places the request of S_k at appropriate location in the request queue, and sends a GRANT(j) to the top request's site in the queue.
  - Maekawa's algorithm requires extra messages to handle deadlocks.
  - Maximum number of messages required per CS execution in this case is 5√N.

Example : Deadlock Handling



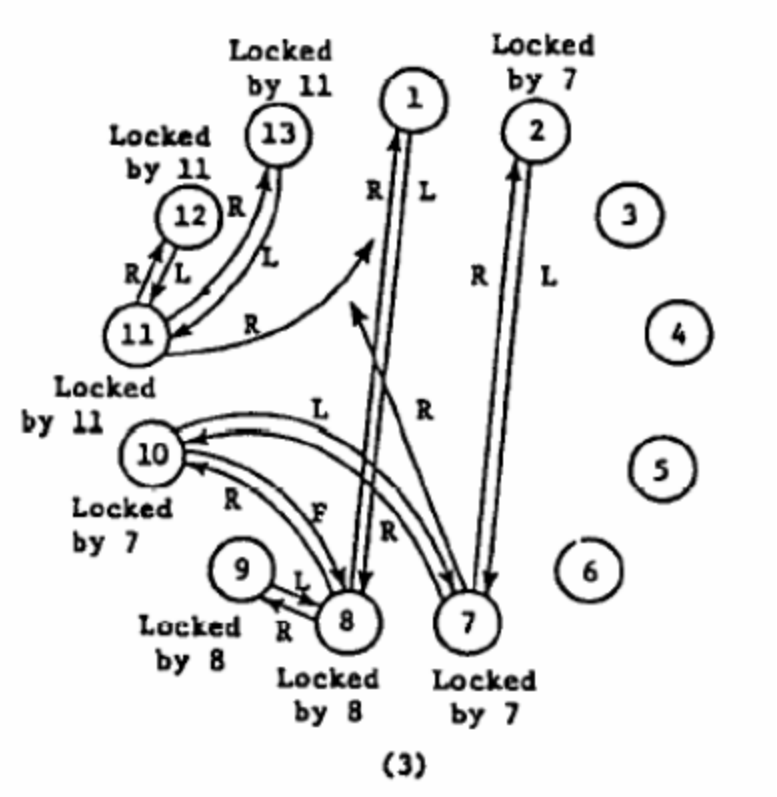
$$
\begin{aligned}
S_1 &= \{1, 2, 3, 4\} \\
S_5 &= \{1, 5, 6, 7\} \\
S_8 &= \{1, 8, 9, 10\} \\
S_{11} &= \{1, 11, 12, 13\} \\
S_2 &= \{2, 5, 8, 11\} \\
S_6 &= \{2, 6, 9, 12\} \\
S_7 &= \{2, 7, 10, 13\} \\
S_{10} &= \{3, 5, 10, 12\} \\
S_3 &= \{3, 6, 8, 13\} \\
S_9 &= \{3, 7, 9, 11\} \\
S_{13} &= \{4, 5, 9, 13\} \\
S_4 &= \{4, 6, 10, 11\} \\
S_{12} &= \{4, 7, 8, 12\}
\end{aligned}
$$

Example : Deadlock Handling



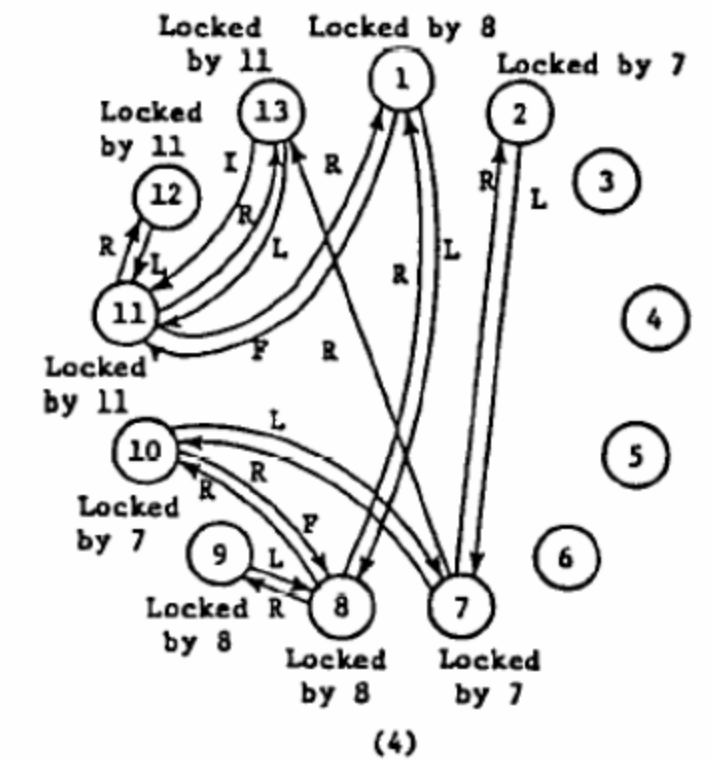
$$
\begin{aligned}
S_1 &= \{1, 2, 3, 4\} \\
S_5 &= \{1, 5, 6, 7\} \\
S_8 &= \{1, 8, 9, 10\} \\
S_{11} &= \{1, 11, 12, 13\} \\
S_2 &= \{2, 5, 8, 11\} \\
S_6 &= \{2, 6, 9, 12\} \\
S_7 &= \{2, 7, 10, 13\} \\
S_{10} &= \{3, 5, 10, 12\} \\
S_3 &= \{3, 6, 8, 13\} \\
S_9 &= \{3, 7, 9, 11\} \\
S_{13} &= \{4, 5, 9, 13\} \\
S_4 &= \{4, 6, 10, 11\} \\
S_{12} &= \{4, 7, 8, 12\}
\end{aligned}
$$

Example : Deadlock Handling



$$
\begin{aligned}
S_1 &= \{1, 2, 3, 4\} \\
S_5 &= \{1, 5, 6, 7\} \\
S_8 &= \{1, 8, 9, 10\} \\
S_{11} &= \{1, 11, 12, 13\} \\
S_2 &= \{2, 5, 8, 11\} \\
S_6 &= \{2, 6, 9, 12\} \\
S_7 &= \{2, 7, 10, 13\} \\
S_{10} &= \{3, 5, 10, 12\} \\
S_3 &= \{3, 6, 8, 13\} \\
S_9 &= \{3, 7, 9, 11\} \\
S_{13} &= \{4, 5, 9, 13\} \\
S_4 &= \{4, 6, 10, 11\} \\
S_{12} &= \{4, 7, 8, 12\}
\end{aligned}
$$

Example : Deadlock Handling



$S_1 = \{1, 2, 3, 4\}$

$S_5 = \{1, 5, 6, 7\}$

$S_8 = \{1, 8, 9, 10\}$

$S_{11} = \{1, 11, 12, 13\}$

$S_2 = \{2, 5, 8, 11\}$

$S_6 = \{2, 6, 9, 12\}$

$S_7 = \{2, 7, 10, 13\}$

$S_{10} = \{3, 5, 10, 12\}$

$S_3 = \{3, 6, 8, 13\}$

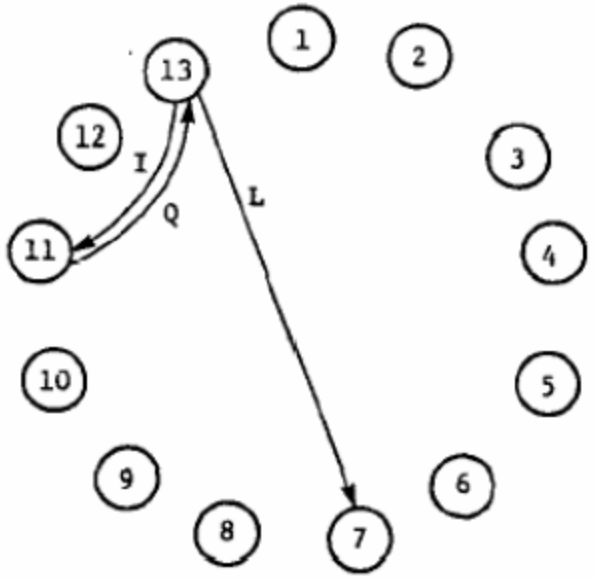
$S_9 = \{3, 7, 9, 11\}$

$S_{13} = \{4, 5, 9, 13\}$

$S_4 = \{4, 6, 10, 11\}$

$S_{12} = \{4, 7, 8, 12\}$

Example : Deadlock Handling



$S_1 = \{1, 2, 3, 4\}$
$S_5 = \{1, 5, 6, 7\}$
$S_8 = \{1, 8, 9, 10\}$
$S_{11} = \{1, 11, 12, 13\}$
$S_2 = \{2, 5, 8, 11\}$
$S_6 = \{2, 6, 9, 12\}$
$S_7 = \{2, 7, 10, 13\}$
$S_{10} = \{3, 5, 10, 12\}$
$S_3 = \{3, 6, 8, 13\}$
$S_9 = \{3, 7, 9, 11\}$
$S_{13} = \{4, 5, 9, 13\}$
$S_4 = \{4, 6, 10, 11\}$
$S_{12} = \{4, 7, 8, 12\}$

Token-based Algorithms

- Unique token circulates among the participating sites.
- A site can enter CS if it has the token.
- Token-based approaches use sequence numbers instead of time stamps (Used to distinguish between old and current requests).
  - Request for a token contains a sequence number.
  - Sequence number of sites advance independently.
- Correctness issue is trivial since only one token is present , only one site can enter CS.
- Deadlock and starvation issues to be addressed.

Suzuki-Kasami's Broadcast Algorithm

- If a site wants to enter the CS and it does not have the token, it broadcasts a REQUEST message for the token to all other sites.
- A site which possesses the token sends it to the requesting site upon the receipt of its REQUEST message.
- If a site receives a REQUEST message when it is executing the CS, it sends the token only after it has completed the execution of the CS.

Suzuki-Kasami's Broadcast Algorithm

This algorithm must efficiently address the following two design issues:

1. How to distinguish an outdated REQUEST message from a current REQUEST message:

2. How to determine which site has an outstanding request for the CS:
   - After a site has finished the execution of the CS, it must determine what sites have an outstanding request for the CS so that the token can be dispatched to one of them.

Suzuki-Kasami's Broadcast Algorithm

- The first issue is addressed in the following manner:
- A REQUEST message of site S_j has the form REQUEST(j, n) where n (n=1,2, ...) is a sequence number which indicates that site S_j is requesting its n^{th} CS execution.
- A site S_i keeps an array of integers $RN_i[1..N]$ where $RN_i[j]$ denotes the largest sequence number received in a REQUEST message so far from site S_j .
- When site S_i receives a REQUEST(j, n) message, it sets $Rn_i[j] := \max(RN_i[j], n)$.
- When a site S_i receives a REQUEST(j, n) message, the request is outdated if $RN_i[j] > n$.

Suzuki-Kasami's Broadcast Algorithm

- The second issue is addressed in the following manner:
- The token consists of a queue of requesting sites, Q, and an array of integers LN[1..N], where LN[j] is the sequence number of the request which site S_j executed most recently.
- After executing its CS, a site S_i updates $LN[i]:=RN_i[i]$ to indicate that its request corresponding to sequence number $RN_i[i]$ has been executed.
- At site S_i if $RN_i[j]=LN[j]+1$, then site S_j is currently requesting token.

The Algorithm

Requesting the critical section

a. If requesting site S_i does not have the token, then it increments its sequence number, $RN_i[i]$, and sends a REQUEST(i, sn) message to all other sites. ('sn' is the updated value of $RN_i[i]$).

b. When a site S_j receives this message, it sets $RN_j[i]$ to $\max(RN_j[i], sn)$. If S_j has the idle token, then it sends the token to S_i if $RN_j[i]=LN[i]+1$.

Executing the critical section

c. Site S_i executes the CS after it has received the token.

The Algorithm

Releasing the critical section

Having finished the execution of the CS, site S_i takes the following actions:

d. It sets LN[i] element of the token array equal to $RN_i[i]$.

e. For every site S_j whose id is not in the token queue, it appends its id to the token queue if $RN_i[j]=LN[j]+1$.

f. If the token queue is nonempty after the above update, S_i deletes the top site id from the token queue and sends the token to the site indicated by the id.

An Example

Step 1: S1 has token, S3 is in queue

Site	Seq. Vector RN	Token Vect. LN	Token Queue
S1	10, 15, 9	10, 15, 8	3
S2	10, 16, 9		
S3	10, 15, 9		

Step 2: S3 gets token, S2 in queue

Site	Seq. Vector RN	Token Vect. LN	Token Queue
S1	10, 16, 9		
S2	10, 16, 9		
S3	10, 16, 9	10, 15, 9	2

Step 3: S2 gets token, queue empty

Site	Seq. Vector RN	Token Vect. LN	Token Queue
S1	10, 16, 9		
S2	10, 16, 9	10, 16, 9	<empty>
S3	10, 16, 9		

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.

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.

Raymond's Tree-Based Algorithm

- This algorithm uses a spanning tree to reduce the number of messages exchanged per critical section execution.
- The network is viewed as a graph, a spanning tree of a network is a tree that contains all the N nodes.
- The algorithm assumes that the underlying network guarantees message delivery. All nodes of the network are completely reliable.

Raymond's Tree-Based Algorithm

- Sites are arranged in a logical directed tree.
  - Root: token holder.
  - Edges: directed towards root.

- Every site has a variable holder that points to an immediate neighbor node, on the directed path towards root (Root's holder point to itself).

The Algorithm

Requesting CS

- If S_i does not hold token and request CS, sends REQUEST upwards provided its request_q is empty. It then adds its request to request_q.
- **Non-empty request_q ->** REQUEST message for top entry in q (if not done before).
- **Site on path to root receiving REQUEST ->** propagate it up, if its request_q is empty. Add request to request_q.
- **Root on receiving REQUEST ->** send token to the site that forwarded the message. Set holder to that forwarding site.
- **Any Si receiving token ->** delete top entry from request_q, send token to that site, set holder to point to it. If request_q is non-empty now, send REQUEST message to the holder site.

The Algorithm

Executing CS:

- getting token with the site at the top of request_q.
- Delete top of request_q, enter CS.

Releasing CS:

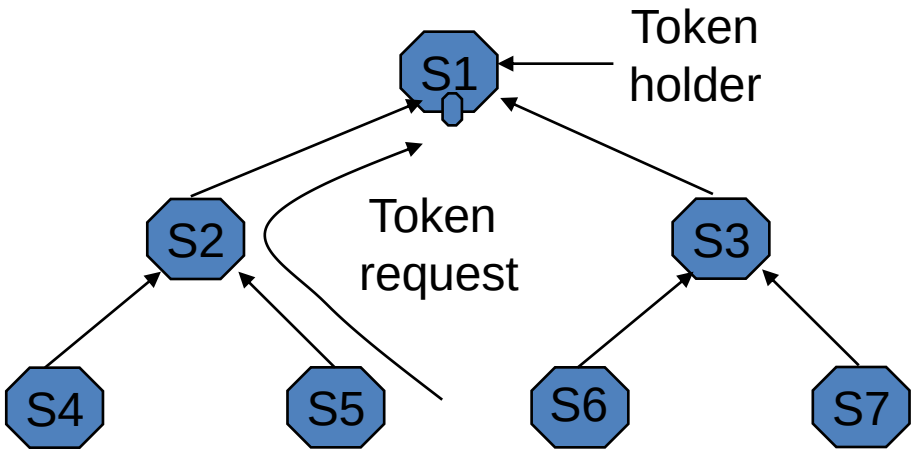
- If request_q is non-empty, delete top entry from q, send token to that site, set holder to that site.
- If request_q is non-empty now, send REQUEST message to the holder site.

Performance

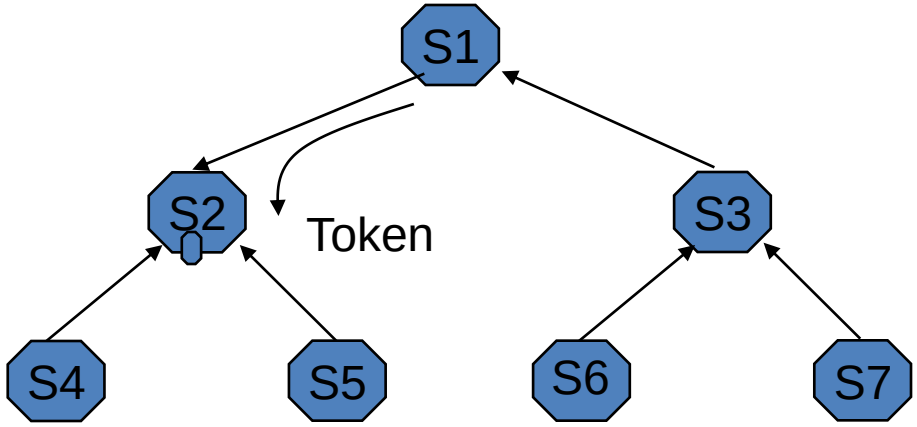
- Average messages: O(log N) as average distance between 2 nodes in the tree is O(log N).
- Synchronization delay: (T log N) / 2, as average distance between 2 sites to successively execute CS is (log N) / 2.
- Greedy approach: Intermediate site getting the token may enter CS instead of forwarding it down. Affects fairness, may cause starvation.

An Example

Step 1:

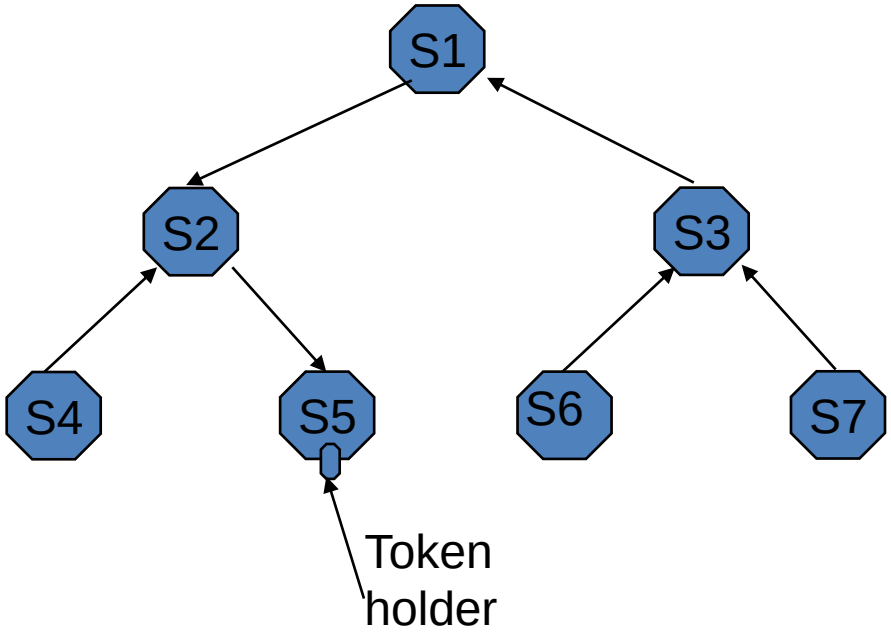


Step 2:



An Example…

Step 3:



Singhal's Heuristic Algorithm

- Instead of broadcast, each site maintains information of other sites, guess the sites likely to have the token.
- Data Structures:
  - S_i maintains $SV_i[1..M]$ (state) and $SN_i[1..M]$ (highest sequence number) for storing information about other sites.
  - Token contains 2 arrays: TSV[1..M] and TSN[1..M].
  - States of a site
    - R : requesting CS
    - E : executing CS
    - H : Holding token, idle
    - N : None of the above
  - Initialization:
    - $SV_i[j] := N$, for j = M .. i; $SV_i[j] := R$, for j = i-1 .. 1; $SN_i[j] := 0$, j = 1..M. S1 (Site 1) is in state H.
    - Token: TSV[j] := N and TSN[j] := 0, j = 1 .. M.

The Algorithm

Requesting CS

1. If S_i has no token and requests CS:
   - $SV_i[i] := R$.
   - $SN_i[i] := SN_i[i] + 1$.
   - Send REQUEST(i,sn) to sites S_j for which $SV_i[j] = R$. (sn: sequence number, updated value of $SN_i[i]$).
2. Receiving REQUEST(i,sn): if sn <= $SN_j[i]$, then discard the message, otherwise, update $SN_j[i]$ to sn and do:
   - $SV_j[j] = N$ -> $SV_j[i] := R$.
   - $SV_j[j] = R$ -> If $SV_j[i]$!= R, set it to R & send REQUEST(j,$SN_j[j]$) to S_i. Else do nothing.
   - $SV_j[j] = E$ -> $SV_j[i] := R$.
   - $SV_j[j] = H$ -> $SV_j[i] := R$, $TSV[i] := R$, $TSN[i] := sn$, $SV_j[j] = N$. Send token to S_i.

The Algorithm

Executing CS

3. After getting token set $SV_i[i] := E$.

Releasing CS

4. $SV_i[i] := N$, $TSV[i] := N$. Then, do:

   For all S_j, , j=1 to M do

   if $SN_i[j] > TSN[j]$

   then $TSV[j] := SV_i[j]$; $TSN[j] := SN_i[j]$

   else $SV_i[j] := TSV[j]$; $SN_i[j] := TSN[j]\}$

5. If $SV_i[j] = N$, for all j, then set $SV_i[i] := H$. Else send token to a site S_j provided $SV_i[j] = R$.

Performance and Fairness

- Performance
  - Low to moderate loads: average of N/2 messages.
  - High loads: N messages (all sites request CS).
  - Synchronization delay: T.
- Fairness of algorithm will depend on choice of S_i, since no queue is maintained in token.

An Example

Sn	R	R	R	R	N
S4	R	R	R	N	N
S3	R	R	N	N	N
S2	R	N	N	N	N
S1	H	N	N	N	N
	1	2	3	4	n

(a) Initial Pattern

Each row in the matrix has increasing number of Rs.

Sn	R	R	R	R	N
S4	R	R	R	N	N
S2	R	N	R	N	N
S1	N	N	R	N	N
S3	N	N	H	N	N
	1	2	3	4	n

(b) Pattern after S3 gets the token from S1.

Stair case is pattern can be identified by noting that S1 has 1 R and S2 has 2 Rs and so on. Order of occurrence of R in a row does not matter.

An Example

Assume there are 3 sites in the system.
Initially:

Site 1: $SV_1[1] = H$, $SV_1[2] = N$, $SV_1[3] = N$.
$SN_1[1]$, $SN_1[2]$, $SN_1[3]$ are 0.

Site 2: $SV_2[1] = R$, $SV_2[2] = N$, $SV_2[3] = N$.
SNs are 0.

Site 3: $SV_3[1] = R$, $SV_3[2] = R$, $SV_3[3] = N$.
SNs are 0.

Token: TSVs are N.
TSNs are 0.

An Example

- Assume site 2 is requesting token.
  - S2 sets $SV_2[2] = R$, $SN_2[2] = 1$.
  - S2 sends REQUEST(2,1) to S1 (since only S1 is set to R in SV[2])
- S1 receives the REQUEST. Accepts the REQUEST since $SN_1[2]$ is smaller than the message sequence number.
  - Since $SV_1[1]$ is H: $SV_1[2] = R$, $TSV[2] = R$, $TSN[2] = 1$, $SV_1[1] = N$.
  - Send token to S2
- S2 receives the token. $SV_2[2] = E$. After exiting the CS, $SV_2[2] = TSV[2] = N$. Updates SN, SV, TSN, TSV. Since nobody is REQUESTing, $SV_2[2] = H$.
- Assume S3 makes a REQUEST now. It will be sent to both S1 and S2. Only S2 responds since only $SV_2[2]$ is H ($SV_1[1]$ is N now).

Comparison

Non-Token	**Resp. Time(ll)**	**Sync. Delay**	**Messages(ll)**	**Messages(hl)**
Lamport	2T+E	T	3(N-1)	3(N-1)
Ricart-Agrawala	2T+E	T	2(N-1)	2(N-1)
Maekawa	2T+E	2T	3*sq.rt(N)	5*sq.rt(N)
Token	**Resp. Time(ll)**	**Sync. Delay**	**Messages(ll)**	**Messages(hl)**
Suzuki-Kasami	2T+E	T	N	N
Singhal	2T+E	T	N/2	N
Raymond	T(log N)+E	Tlog(N)/2	log(N)	4