Distributed Mutual Exclusion

Mutual exclusion in single-computer system vs. distributed system

In a single-computer system, the status of a shared resource and the status of users is readily available in the shared memory and solutions to the mutual exclusion problem can be easily implemented using shared variables(e.g. semaphore).

However in distributed system both the shared resources and the users may be distributed. So, approaches based on shared variables are not applicable to distributed systems and approaches based on message passing must be used.

The problem of mutual exclusion becomes much more complex in distributed systems due to lack of shared memory and common physical clock and because of unpredictable message delays.

Contd.

Classification of Mutual Exclusion Algo

These algo can be grouped into two classes

- Nontoken-based: require two or more successive rounds of message exchanges among the sites. These algorithms are assertion based because a site can enter its critical section(CS) when assertion defined on its local variables becomes true. Mutual exclusion is enforced because the assertion becomes true only at one site at any given time.

Contd.

- Token-based: a unique token (also known as the PRIVILEGE message) is shared among the sites. A site is allowed to enter its CS if it possesses the token and it continues to hold the token until the execution of the CS is over.

System Model:

At any instant a site may have several requests for CS. A site queues up these requests and serves them one at a time. A site can be in one of the following three states: requesting CS, executing CS or neither requesting nor executing CS (i.e. idle). In the requesting CS state, the site is blocked and can not make further requests for CS. In the idle state the site is executing outside its CS. In the token-based algo a site can also be in a state where a site holding the token is executing outside the CS. Such a state is referred to as an idle token state.

Contd.

Requirements of Mutual Exclusion Algo

Primary objective is to maintain mutual exclusion, i.e. to guarantee that only one request accesses the CS at a time. In addition the following characteristics are considered important in a mutual exclusion algo.

- Freedom from deadlocks

two or more sites should not endlessly wait for messages that will never arrive.

Contd.

- Freedom from starvation

a site should not be forced to wait indefinitely to execute CS while other sites are repeatedly executing CS. That is every requesting site should get an opportunity to execute CS in finite time.

- Fairness

it dictates that requests must be executed in the order they are made. Since a physical global clock does not exist, time is determined by logical clocks.

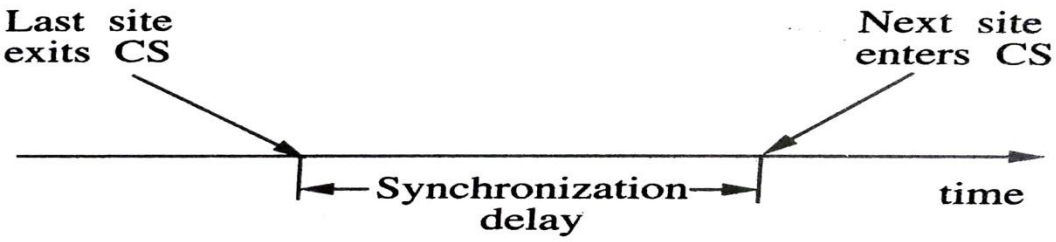
- Fault tolerance

a mutual exclusion algo is fault-tolerant if in the wake of a failure it can recognize itself so that it continues to function without any disruptions.

How to measure the performance?

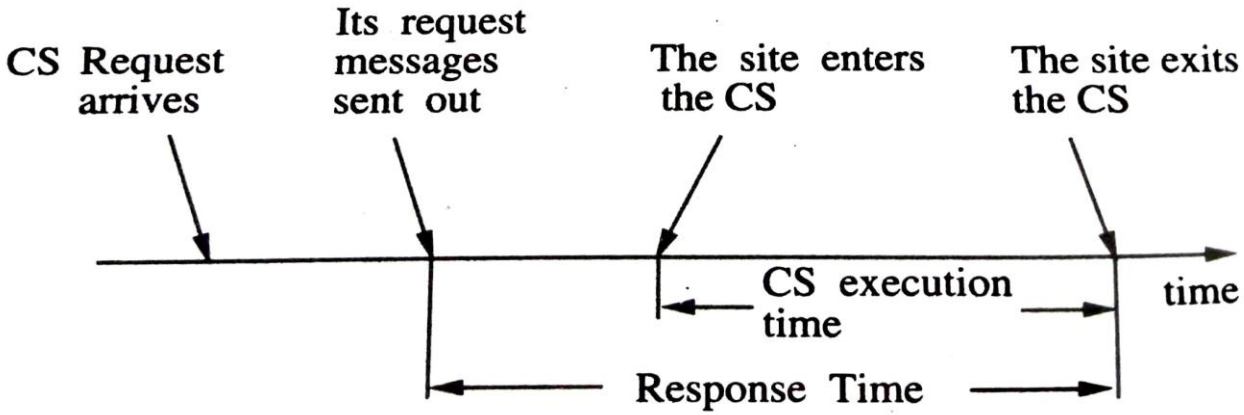
The performance of mutual exclusion algorithm is generally measured by the following four metrics:

- number of messages necessary per CS invocation
- synchronization delay(which is the time required after a site leaves the CS and before the next site enters the CS)



Contd.

- Response time:- is the time interval a request wait for its CS execution to be over after its request message have been sent out. Thus response time does not include the time a request waits at a site before its request messages have been sent out.



Contd.

- System throughput:- is the rate at which the system executes requests for the CS. If sd is the synchronization delay and E is the avg. CS execution time, then throughput is given by the following equation:

  system throughput = 1/(sd+E)

Best case Performance:

In most mutual exclusion algo the best value of the response time is a round trip message delay plus CS execution time, 2T+E (where T is the avg. message delay and E is the avg. CS execution time).

A simple solution to distributed mutual exclusion

A site called the control site is assigned the task of granting permission for the CS execution. To request the CS, a site sends a REQUEST message to the control site. The control site queues up the requests for the CS and grants them permission one by one.

Drawbacks:

- Single point of failure, the control site.
- The control site likely to be swamped with extra work.
- The communication link near the control site are likely to be congested and become a bottleneck.
- The synchronization delay of this algo is 2T because a site should first release permission to the control site and then the control site should grant permission to the next site to execute the CS. This has serious implications for the system throughput, which is = 1/(2T+E) in this algo.

Non-token based algo

These algo use timestamps to order requests for the CS and to resolve conflicts between simultaneous requests for the CS. In all these algo logical clocks are maintained and updated according to Lamport's scheme. Each request for the CS gets a timestamp and smaller timestamp requests have priority over larger timestamp requests.

Lamport's Algorithm

Lamport was the first to give a distributed mutual exclusion algo as an illustration of his clock synchronization scheme. In Lamport's algo for all i : $1<=i<=N::R_i=\{S_1,S_2,\ldots\ldots,S_n\}$. Every site S_i keeps a queue , reqest-queue, which contains mutual exclusion requests ordered by their timestamps. The algo requires messages to be delivered in the FIFO order between every pair of sites.

Contd.

Requesting the CS

1. When a site S_i wants to enter the CS, it sends REQUEST(ts_i,i) message to all the sites in its request set R_i and places the request on $request\_queue_i$. ((ts_i, i) is the timestamp of the request.)
2. When a site S_j receives the REQUEST(ts_i, i) message from site S_i, it returns a timestamped REPLY message to S_i and places site S_i's request on $request\_queue_j$.

Contd.

Executing the CS:

Site S_i enters the CS when the two following 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 $request\_queue_i$.

Releasing the CS:

3. Site S_i, upon exiting the CS removes its request from the top of its request queue and sends a timestamped RELEASE message to all the sites in its request set.

4. 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. The algo executess CS requests in the increasing order of timestamps.

Contd.

Correctness

Theorem: Lamport's Algo achieves mutual exclusion

Proof: The 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 a loss of generality, assume that S_i's request has a smaller timestamp than the request of S_j. Due to condition L1 and the FIFO property of the communication channels, it is clear that at instant t, the request of Si 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! Hence Lamport's algo achieves mutual exclusion.

Contd.

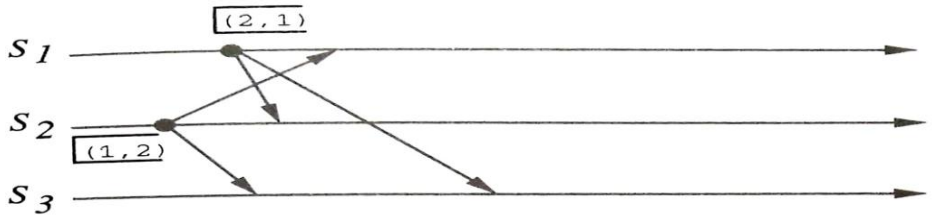


FIGURE 6.3
Sites S_1 and S_2 are making requests for the CS.

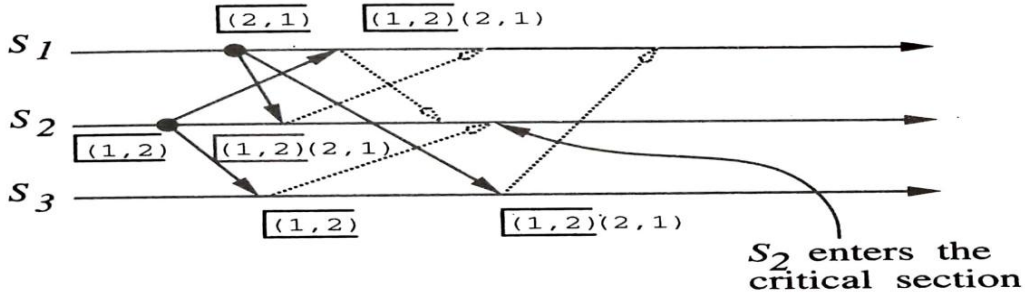


FIGURE 6.4
Site S_2 enters the CS.

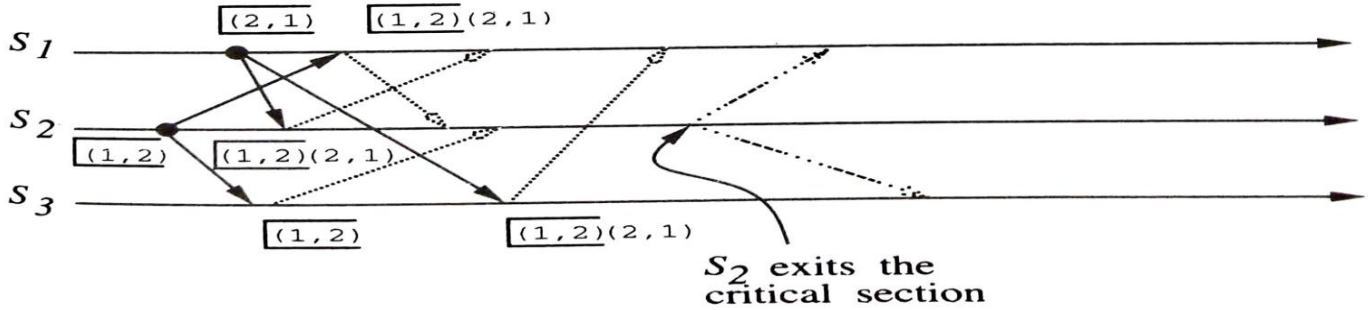


FIGURE 6.5
Site S_2 exits the CS and sends RELEASE messages.

Contd.

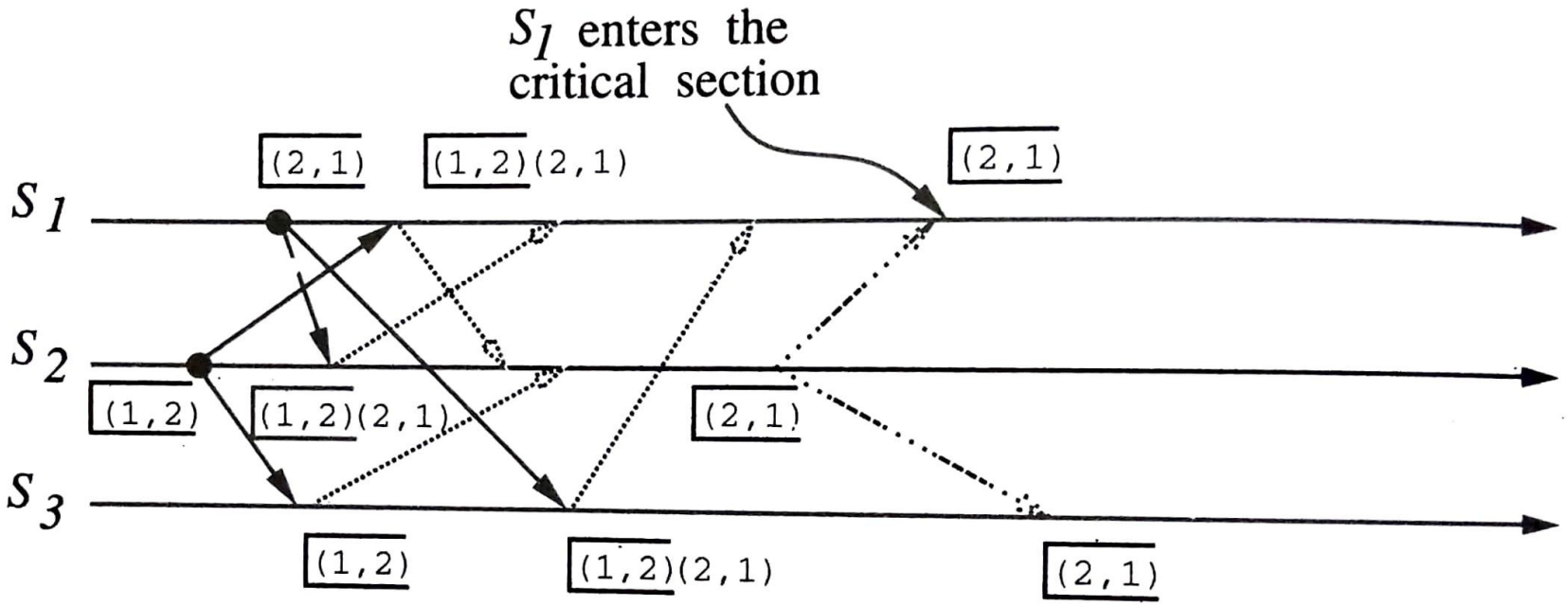


FIGURE 6.6
Site S_1 enters the CS.

Contd.

In previous Fig. 6.3 to 6.6 illustrate the operation of Lamport's algo. In Fig. 6.3 sites S_1 and S_2 are making requests for the CS and send out REQUEST messages to other sites. The timestamps of the requests are (2,1) and (1,2) respectively. In Fig. 6.4 S_2 has received REPLY messages from all the other sites and its request is at the top of its *request_queue*. Consequently, it enters the CS. In Fig. 6.5 S_2 exits and sends RELEASE messages to all other sites. In Fig. 6.6, site S_1 has received REPLY messages from all other sites and its request is at the top of its *request_queue*. Consequently, it enters the CS next.

Contd.

Performance:

Lamport's algo requires 3(N-1) messages per CS invocation. (N-1) REQUEST, (N-1) REPLY and (N-1) RELEASE messages. Synchronization delay in the algo is T.

An Optimization:

Lamport's algo can be optimized to require between 3(N-1) and 2(N-1) messages per CS execution by suppressing REPLY messages in certain situations. For ex. Suppose 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. In this case 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 a timestamp higher than its own, it can conclude that site S_j does not have any smaller timestamp request that is still pending.

The Ricart-Agrawala Algorithm

This algo is an optimization of Lamport's algo that dispenses with RELEASE messages by cleverly merging them with REPLY messages. In this algo also for all i : 1<=i<=N :: $R_i=\{S_1,S_2,.....,S_n\}$

The Algo

Requesting the CS:

1. When a site S_i wants to enter the CS, it sends a timestamped REQUEST message to all the sites in its request set.

Contd.

2. 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 or if site S_j is requesting and S_i's request's timestamp is smaller than site S_j's own request's timestamp. The request is deferred otherwise.

Executing the CS:

3. Site S_i enters the CS after it has received REPLY messages from all the sites in its request set.

Contd.

Releasing the CS:

4. When site S_i exits the CS, it sends REPLY messages to all the deferred requests.

A site's REPLY messages are blocked only by sites that are requesting the CS with higher priority(i.e. smaller timestamp). Thus when a site sends out REPLY messages to all the deferred requests, the site with the next highest priority request receives the last needed REPLY message and enters the CS.

Contd.

Correctness

Theorem: The Ricart-Agrawala algo achieves mutual exclusion.

Proof: By contradiction. Suppose two sites S_i and S_j are executing the CS concurrently and S_i's request has a higher priority than the request of S_j. Clearly, S_i received S_j's request after it had made its own request. (otherwise, S_i's request would have lower priority.) 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 the Ricart-Agrawala algo achieves mutual exclusion.

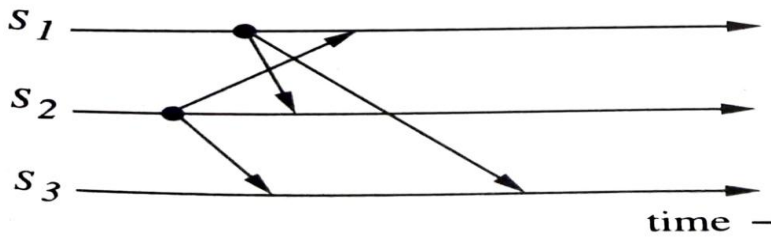


FIGURE 6.7
Sites S_1 and S_2 are making requests for the CS.

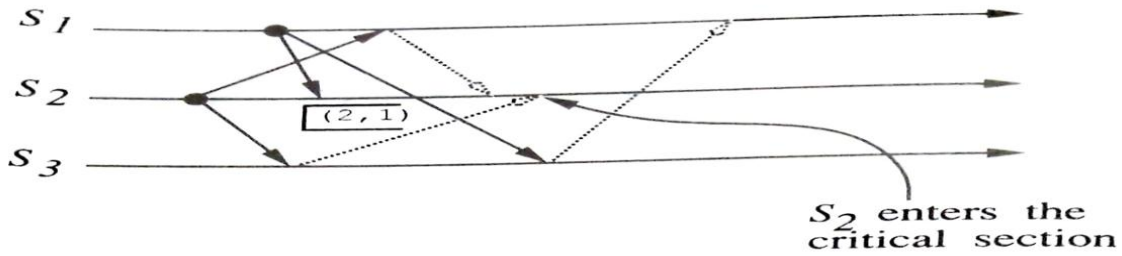


FIGURE 6.8
Site S_2 enter the CS.

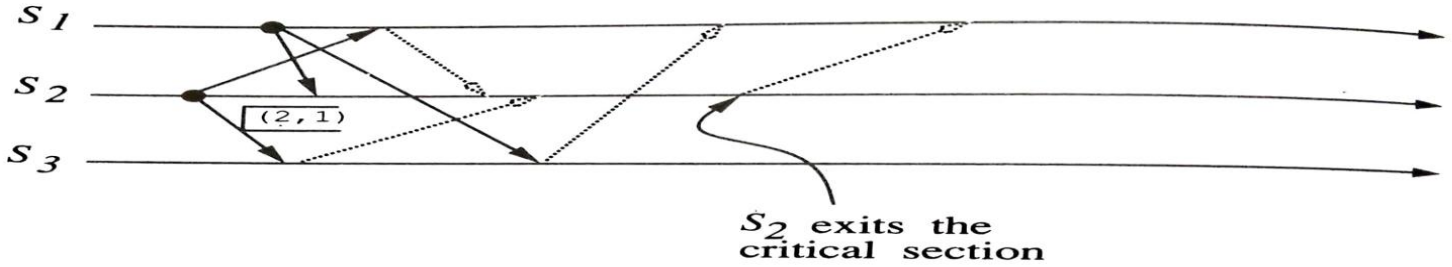


FIGURE 6.9
Site S_2 exits the CS and sends a REPLY message to S_1.

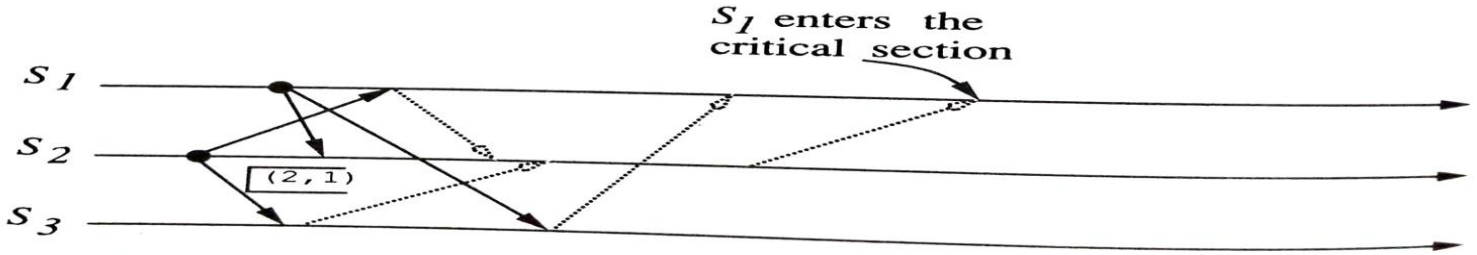


FIGURE 6.10
Site S_1 enters the CS.

Contd.

Fig. 6.7 through 6.10 illustrate the operation of this algo. In Fig. 6.7 sites S1 and S2 are making requests for the CS, sending out REQUEST messages to other sites. The timestamp of the requests are (2,1) and (1,2) respectively. In Fig. 6.8 S2 has received REPLY messages from all other sites and consequently, it enters the CS. In Fig. 6.9, S2 exits the CS and sends a REPLY message to site S1. In Fig. 6.10 site S1 has received REPLY messages from all other sites and enters the CS next.

Contd.

Performance:

This algo requires 2(N-1) messages per CS execution : (N-1) REQUEST and (N-1) REPLY messages. Synchronization delay in the algo is T.

An Optimization:

Once a site S_i has received a REPLY message from a site S_j, the authorization implicit in this message remains valid until S_i sends a REPLY message to S_j(which happens only after the reception of a REQUEST message from S_j). Therefore, after site S_i has received a REPLY message from site S_j, site S_i can enter its CS any number of times without requesting permission from site S_j until S_i sends a REPLY message to S_j. With this change, a site in this algo requests permission from a dynamically varying set of sites and requires 0 to 2(N-1) messages per CS execution.

Token-based Algorithm

In this algo sequence no. is used instead of timestamps. Every request for the token contains a sequence no. and the sequence no. of sites advance independently. A site increments its sequence no. counter every time it makes a request for the token. A primary function of the sequence no. is to distinguish between old and current requests.

Suzuki-Kasami's Broadcast Algorithm

In this algo. if a site attempting to enter the CS does not have the token, it broadcasts a REQUEST message for the token to all the other sites. A site that possesses the token sends it to the requesting site upon receiving its REQUEST message. If a site receives a REQUEST message when it is executing the CS, it sends the token only after it has exited the CS. A site holding the token can enter its CS repeatedly until it sends the token to some other site.

The main design issues in this algo are:

a) distinguishing outdated REQUEST messages from current REQUEST messages

b) determining which site has an outstanding request for the CS

Contd.

Outdated REQUEST messages are distinguished from current REQUEST messages 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 no. that 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 $RN_i[j]$ is the largest sequence no. received so far in a REQUEST message from site S_j. A REQUEST(j,n) message received by site S_i is outdated if $RN_i[j] > n$. When site S_i receives a REQUEST(j,n) message, it sets $RN_i[j]:=max(RN_i[j],n)$.

Contd.

Sites with outstanding requests for the CS are determined 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 no. of the request that 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 no. $R_i[i]$ has been executed. The token array LN[1…N] permits a site to determine if some other site has an outstanding request for the CS. Note that at site S_i if $RN_i[j]=LN[j]+1$, then site S_j is currently requesting the token . After having executed the CS, a site checks this condition for all the j's to determine all the sites that are requesting the token and places their id's in queue, Q , if not already present in this queue, Q. Then the site sends the token to the site at the head of the queue, Q.

The Algorithm(Suzuki-Kasami)

Requesting the CS:

1. If the requesting site S_i does not have the token, then it increments its sequence no. $RN_i[i]$ and sends a REQUEST(i,sn) message to all other sites.(sn is the updated value of $RN_i[i]$)
2. 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 $RN_j[i]=LN[i]+1$.

Contd.

Executing the CS:

3. Site S_i executes the CS when it has received the token.

Releasing the CS:

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

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

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

6. If token queue is non empty after the above update, then it deletes the top site ID from the queue and sends the token to the site indicated by the ID.

Contd.

Correctness:

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 the 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 execute the CS in finite time.

Contd.

Performance:

The beauty of the Suzuki-Kasami algo lies in its simplicity and efficiency. The algo requires 0 or N messages per CS invocation. Synchronization delay in this algo is 0 to T. No message is needed and the sd = 0 if a site holds the idle token at the time of its request.

Raymond's Tree-based Algorithm

Here sites are logically arranged as a directed tree such that the edges of the tree are assigned directions toward the site(root of the tree) that has the token. Every site has a local variable *holder* that points to an immediate neighbor node on a directed path to the root node. Thus, holder variables at the sites define logical tree structure among the sites. If we follow *holder* variables at sites, every site has a directed path leading to the site holding the token. At root site, *holder* points to itself.

Contd.

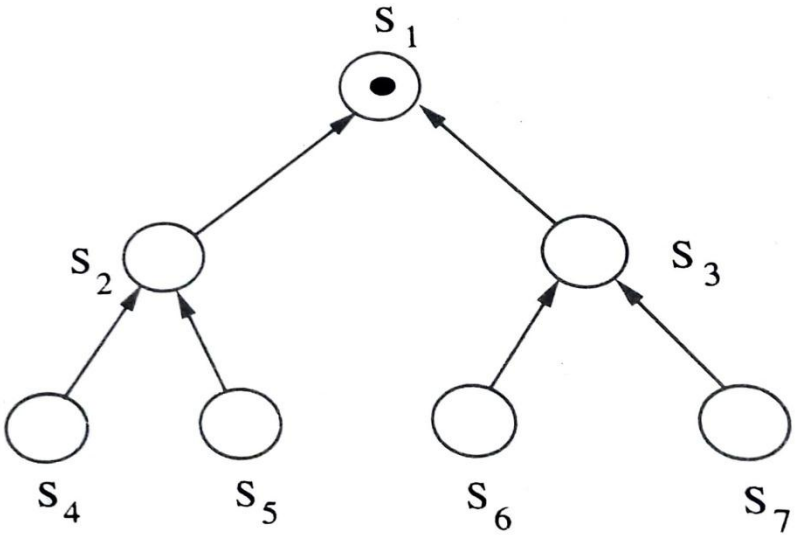


Fig. Sites arranged in a tree configuration

Every site keeps a FIFO queue, called *request_q*, which stores the requests of those neighboring sites that have sent a request to this site, but have not yet been sent the token.

Contd.

The Algorithm

Requesting the CS

1. When a site wants to enter the CS, it sends a REQUEST message to the node along the directed path to the root, provided it does not hold the token and its *request_q* is empty. It then adds its request to its *request_q*. (note that a nonempty *request_q* at a site indicates that the site has sent a REQUEST message to the root node for the top entry in its *request_q*).

Contd.

2. When a site on the path receives this message, it places the REQUEST in its *request_q* and sends a REQUEST message along the directed path to the root provided it has not sent out a REQUEST message on its outgoing edge(for a previously received REQUEST on its *request_q*).

3. When the root site receives a REQUEST message, it sends the token to the site from which it received the REQUEST message and sets its holder variable to point at that site.

Contd.

4. When a site receives the token, it deletes the top entry from its *request_q*, sends the token to the site indicated in this entry, and sets its *holder* variable to point at that site. If the *request_q* is nonempty at this point, then the site sends a REQUEST message to the site which is pointed at by *holder* variable.

Executing the CS

5. A site enters the CS when it receives the token and its own entry is at the top of its *request_q*. in this case, the site deletes the top entry from its *request_q* and enters the CS.

Contd.

Releasing the CS

After a site has finished execution of the CS, it takes the following actions:

6. If its *request_q* is nonempty, then it deletes the top entry from its *request_q*, sends the token to that site, and sets its *holder* variable to point at that site.

7. If the *request_q* is empty at this point, then the site sends a REQUEST message to the site which is pointed at by the *holder* variable.

Contd.

Correctness: The algorithm is free from deadlocks because the acyclic nature of tree configuration eliminates the possibility of circular wait among requesting sites.

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

Proof: A formal correctness proof is long and complex. Thus an informal correctness proof is provided.

The essence of proof is based on the following two facts:

i) a site serves requests in its *request_q* in the FCFS order

ii) every site has a path leading to the site that has the token

Contd.

Due to the latter fact and Step2 of the algorithm, when a site S_i is making a request, there exists a chain of requests from site S_i to site S_h, which holds the token. Let the chain be denoted by S_i, S_{i1}, S_{i2},……,S_{ik-1}, S_{ik}, S_h. When S_h receives a REQUEST message from S_{ik}, it sends the token to S_{jk}. There are two possibilities: S_{ik-1}'s request is at the top of S_{ik}'s *request_q* or it is not at the top. In the first case, S_{ik} sends the token to site S_{ik-1}. In the second case, S_{ik} sends the token to the site, say S_j, at the top of its *request_q* and also sends it a REQUEST message. This extends the chain of requests to S_i, S_{i1}, S_{i2},…..,S_{ik-1}, S_{ik}, S_j,….,S_l, where site S_l executes the CS next. Note that due to fact(1) all the sites in the chain S_j,…..,S_l will execute the CS at most once before the token is returned to site S_{ik}. Thus, site S_{ik} sends the token to S_{ik-1} in finite time. Likewise, S_{ik-1} sends the token to S_{ik-2} in finite time and so on. Eventually, S_{i1} sends the token to S_i. Consequently, a requesting site eventually receives the token.

Contd.

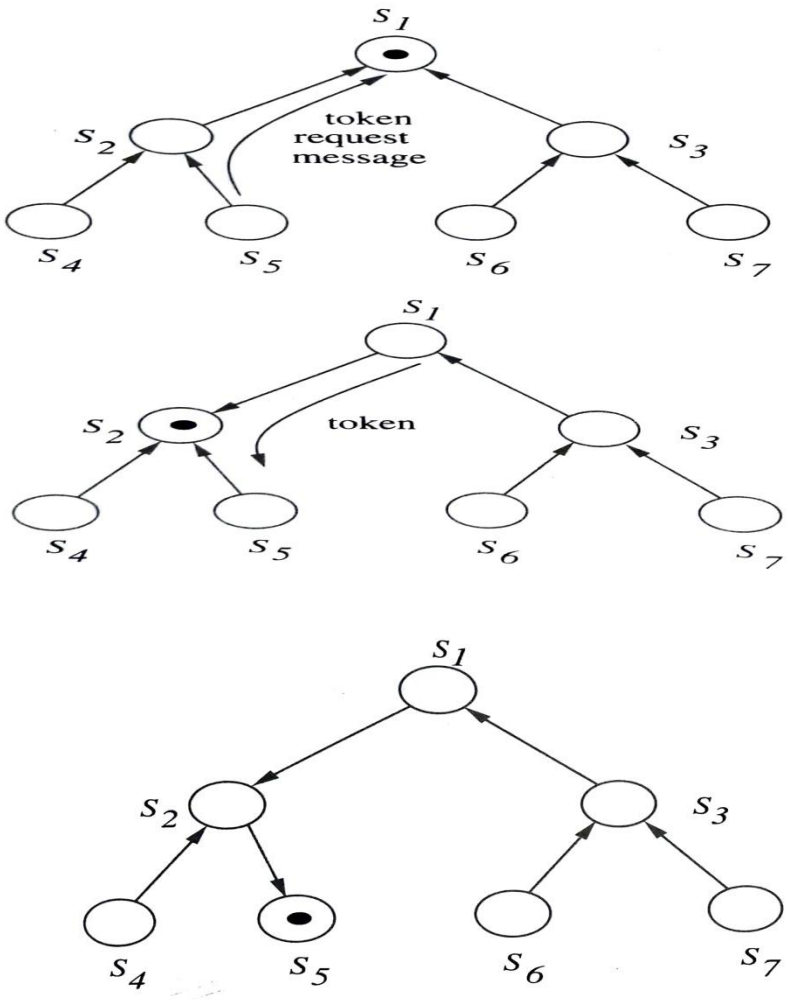


Fig1. Site S5 is requesting the token
Fig2. The token is in transit to S5
Fig3. State after S5 has received the token

Contd.

Performance:

The average message complexity of Raymond's algorithm is O(logN) because the average distance between any two nodes in a tree with N nodes is O(logN). Synchronization delay in this algorithm is (T logN)/2.