

Lecture: 09

Token Based Distributed Mutual Exclusion Algorithms



Rajiv Misra

Dept. of Computer Science & Engineering
Indian Institute of Technology Patna
rajivm@iitp.ac.in

Preface

Recap of Previous Lecture:

- In the previous lecture, we have discussed about quorum based approaches i.e. **Maekawa's Algorithm and Agarwal-El Abbadi Quorum-Based Algorithm.**

Content of this Lecture:

- In this lecture, we will discuss about token based approaches i.e. **Suzuki-Kasami's Broadcast Algorithm** and **Raymond's Tree-Based Algorithm** .

(iii) Token Based Approaches

Introduction

- In token-based algorithms, a **unique token is shared** among the sites. A site is allowed to enter its CS if it possesses the token.
- Token-based algorithms **use sequence numbers** instead of timestamps. (Used to distinguish between old and current requests.)
- The correctness proof of token-based algorithms, that they enforce mutual exclusion, is trivial because an **algorithm guarantees mutual exclusion so long as a site holds the token** during the execution of the CS.
- In this lecture, we will discuss about two token based approaches i.e. (i) Suzuki-Kasami's Broadcast Algorithm and (ii) Raymond's Tree-Based Algorithm .

(i) Suzuki-Kasami's Broadcast Algorithm

- In Suzuki–Kasami's 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.

Contd...

This algorithm must efficiently address the following two design issues:

(1) How to distinguish an outdated REQUEST message from a current REQUEST message:

- Due to variable message delays, a site may receive a token request message after the corresponding request has been satisfied.
- If a site cannot determine if the request corresponding to a token request has been satisfied, it may dispatch the token to a site that does not need it.
- This will not violate the correctness, however, this may seriously degrade the performance.

(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.

Contd...

The first issue is addressed in the following manner:

- A **REQUEST** message of site S_j has the form **REQUEST(j, n)** where n ($n=1, 2, \dots$) 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 $RN_i[j] > n$.

Contd...

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.

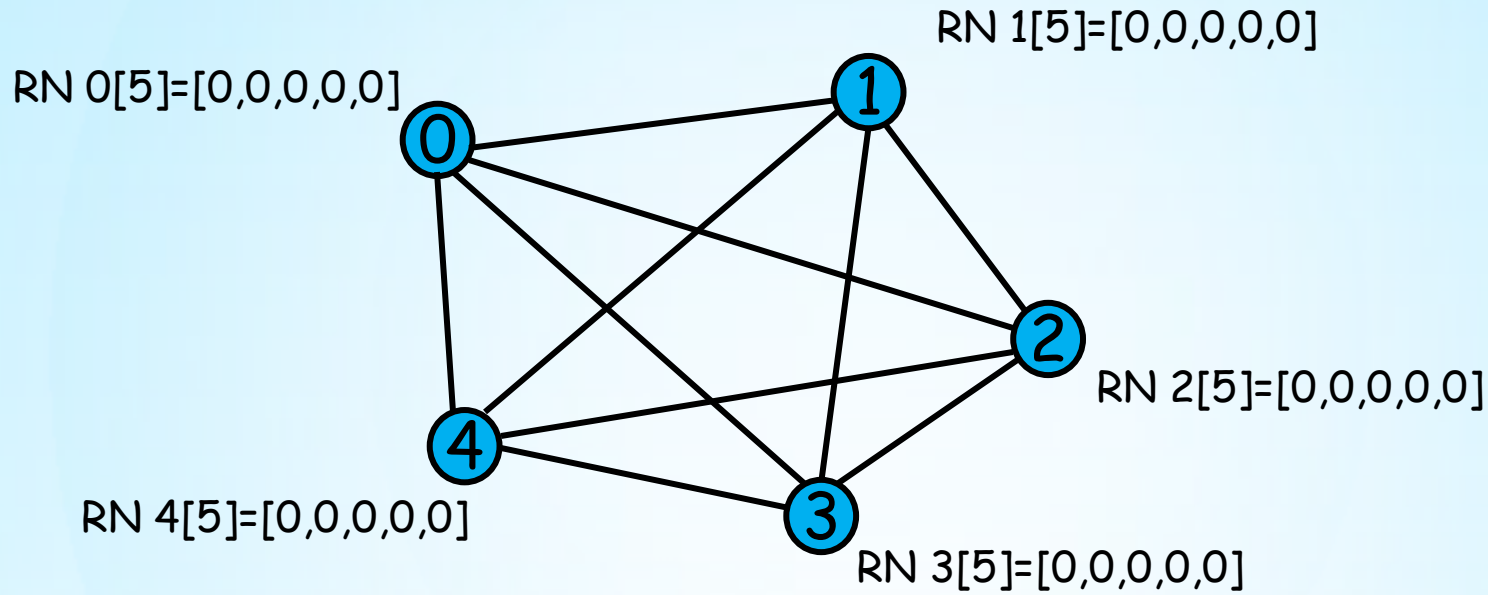
The Algorithm

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.

Suzuki-Kasami's Broadcast Algorithm: Example

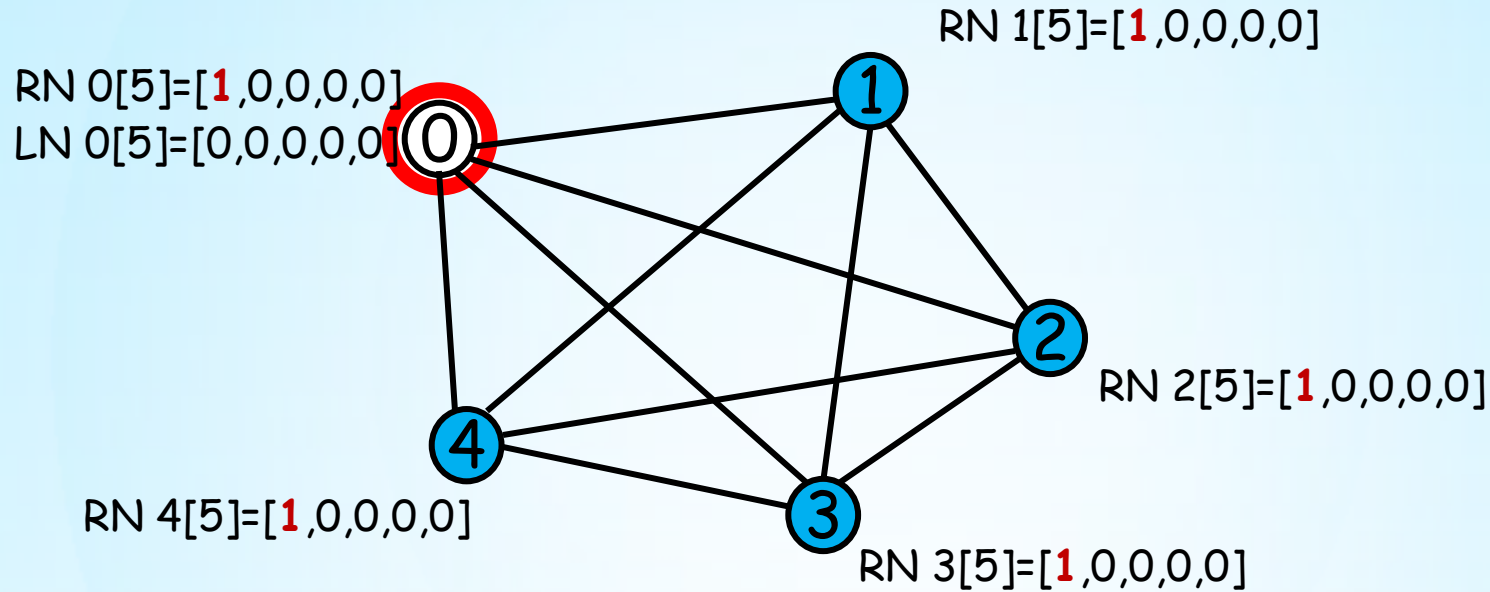


There are 5 sites initially and non are requesting

$RN[j]$ denotes the sequence no of the *latest request* from process j

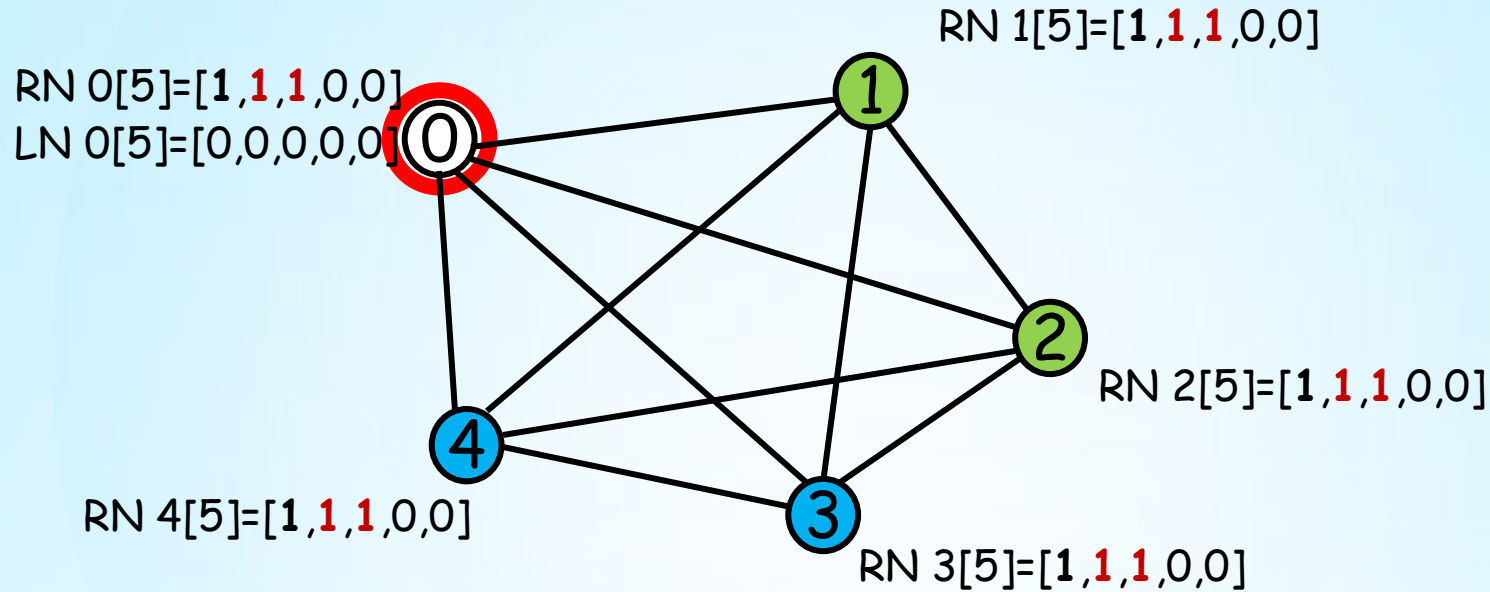
$LN[j]$ denotes the sequence number of *the latest visit to CS* for process j .

Suzuki-Kasami's Broadcast Algorithm: Example



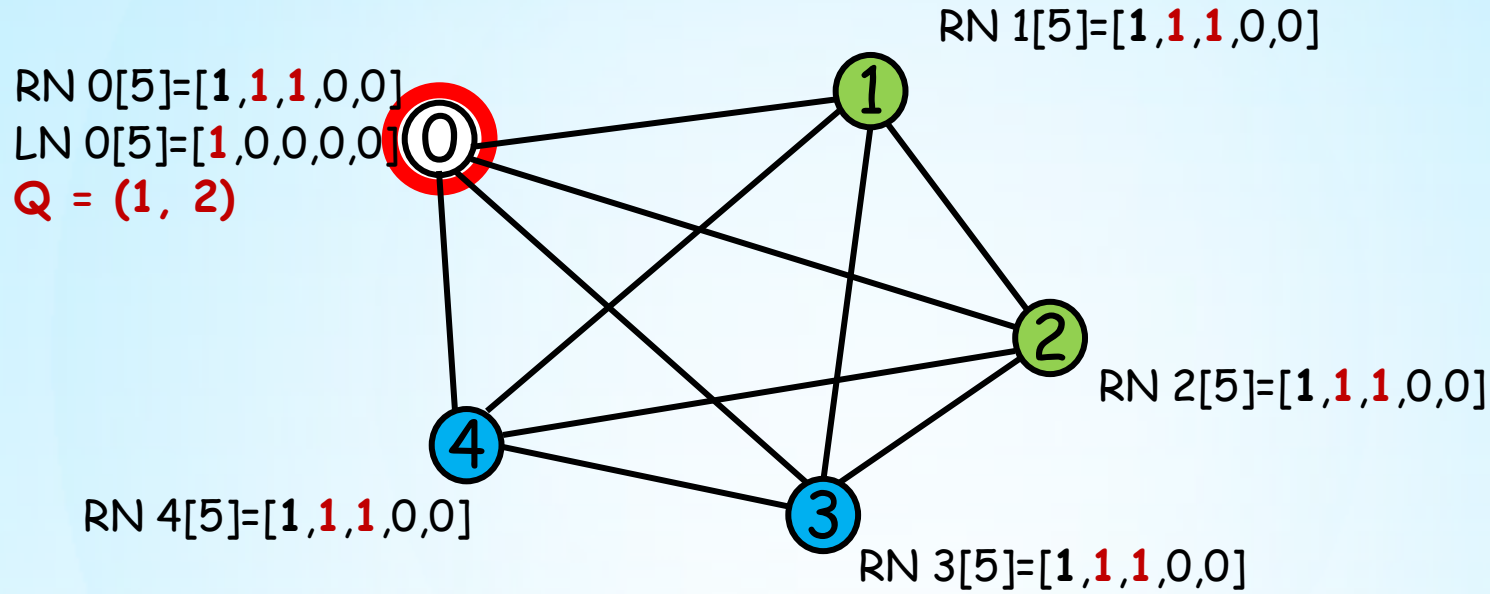
Initial state: Site 0 has sent a request to all, and grabbed the token

Suzuki-Kasami's Broadcast Algorithm: Example



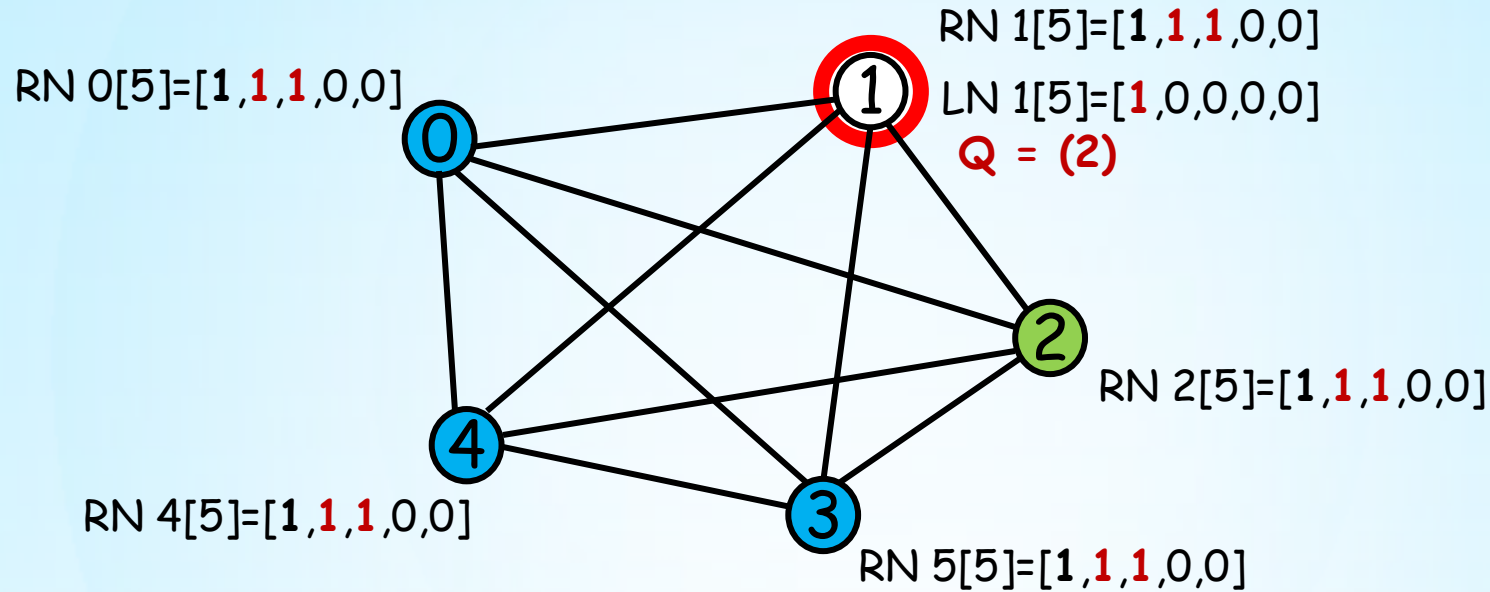
Site 1 & 2 send requests to enter CS

Suzuki-Kasami's Broadcast Algorithm: Example



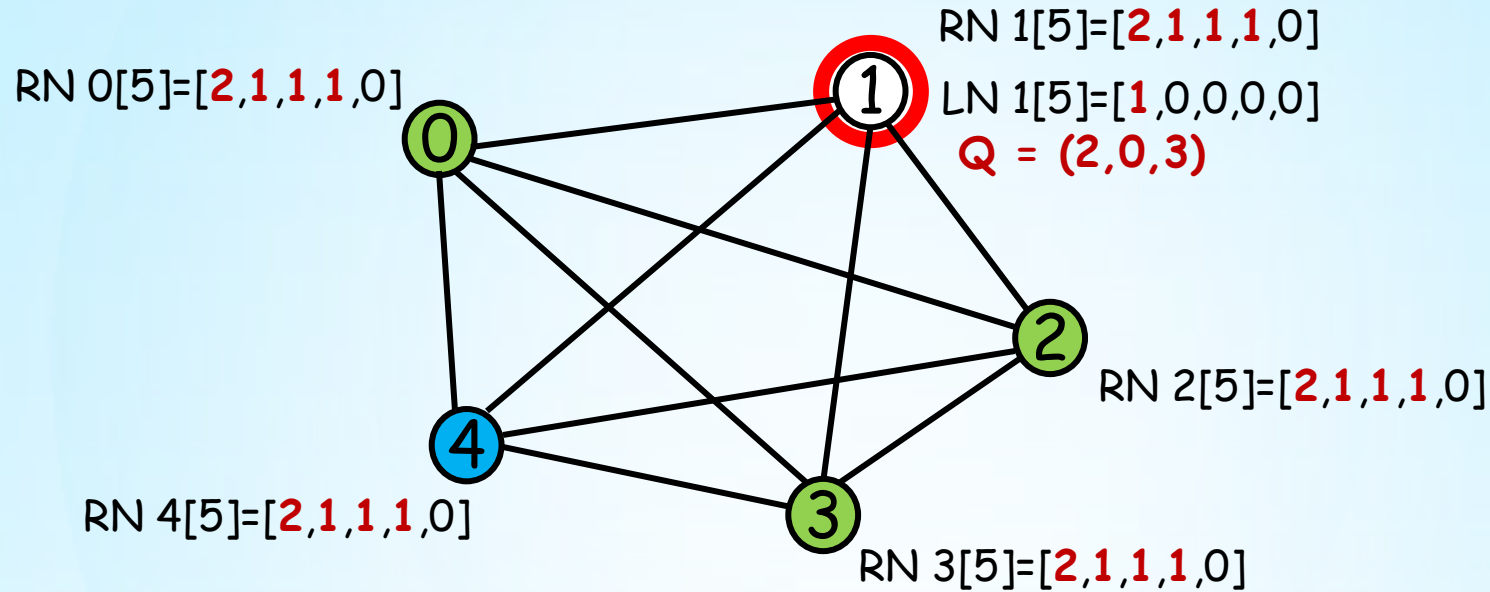
Site 0 prepares to exit CS

Suzuki-Kasami's Broadcast Algorithm: Example



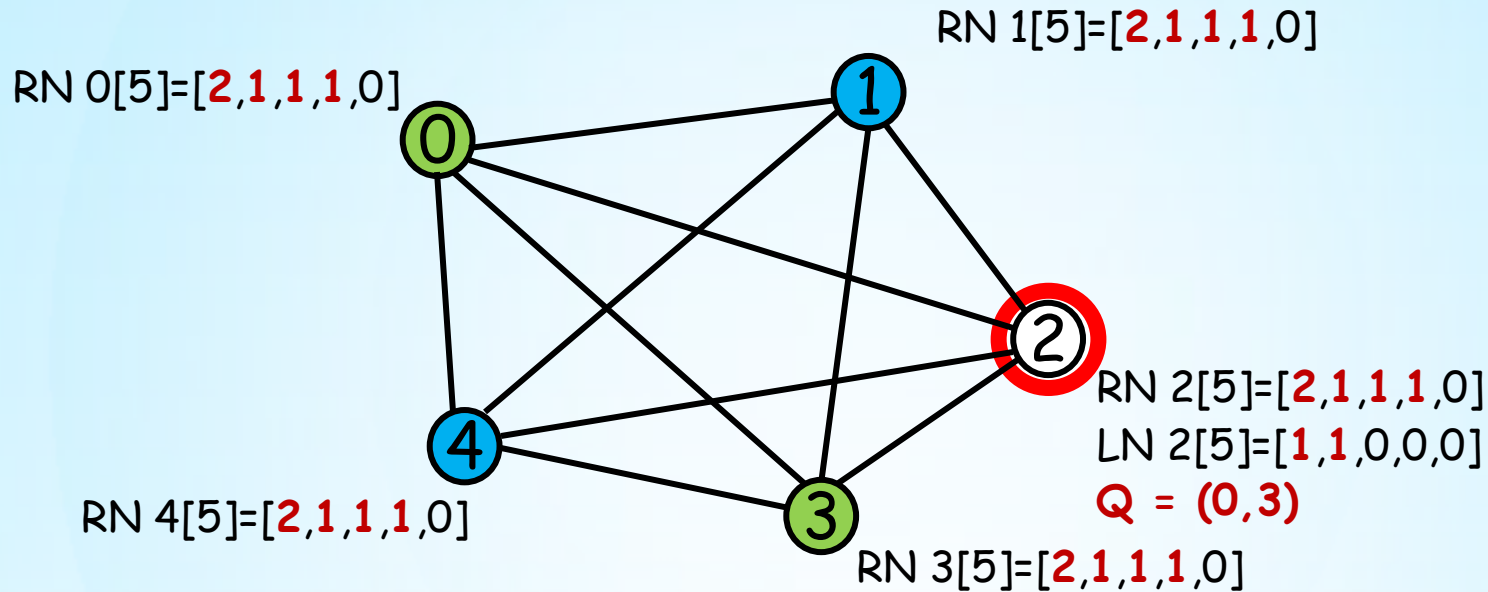
Site 0 passes token (Q and LN) to Site 1

Suzuki-Kasami's Broadcast Algorithm: Example



Site 0 and Site 3 send requests

Suzuki-Kasami's Broadcast Algorithm: Example



Site 1 sends token to Site 2

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 .**

(ii) 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**'.

Contd...

- The algorithm operates on a **minimal spanning tree** of the network topology or a logical structure imposed on the network.
- The algorithm assumes the network nodes to be arranged in an **unrooted tree structure**.
- Figure 9.1 shows a spanning tree of seven nodes A, B, C, D, E, F, and G.
- Messages between nodes traverse along the **undirected edges** of the tree.

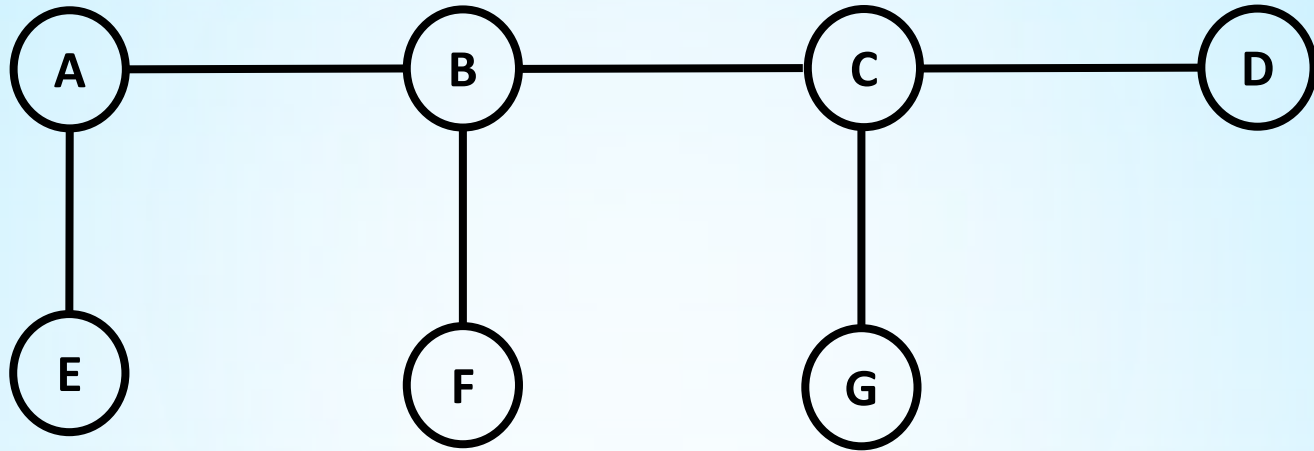


Figure 9.1: A tree of 15 sites

Contd...

- A node needs to hold information about and communicate only to its **immediate-neighboring nodes**.
- Similar to the concept of tokens used in token-based algorithms, this algorithm uses a concept of privilege.
- Only one node can be in possession of the privilege (called the **privileged node**) at any time, except when the privilege is in transit from one node to another in the form of a **PRIVILEGE message**.
- When there are no nodes requesting for the privilege, it remains in possession of the node that last used it.

The HOLDER Variables

- Each node maintains a **HOLDER** variable that provides information about the placement of the privilege in relation to the node itself.
- A node stores in its **HOLDER** variable the identity of a node that it thinks has the privilege or leads to the node having the privilege.
- For two nodes X and Y, if **HOLDER_x = Y**, we could redraw the undirected edge between the nodes X and Y as a directed edge from X to Y.
- For instance, if node G holds the privilege, Figure 9.2 can be redrawn with logically directed edges as shown in the Figure 9.3.

The HOLDER Variables

- The **shaded node in Figure 9.2 represents the privileged node.**
- The following will be the values of the **HOLDER** variables of various nodes:

$\text{HOLDER}_A = B$

$\text{HOLDER}_B = C$

$\text{HOLDER}_C = G$

$\text{HOLDER}_D = C$

$\text{HOLDER}_E = A$

$\text{HOLDER}_F = B$

$\text{HOLDER}_G = \text{self}$

Raymond's Algorithm: Example (1)

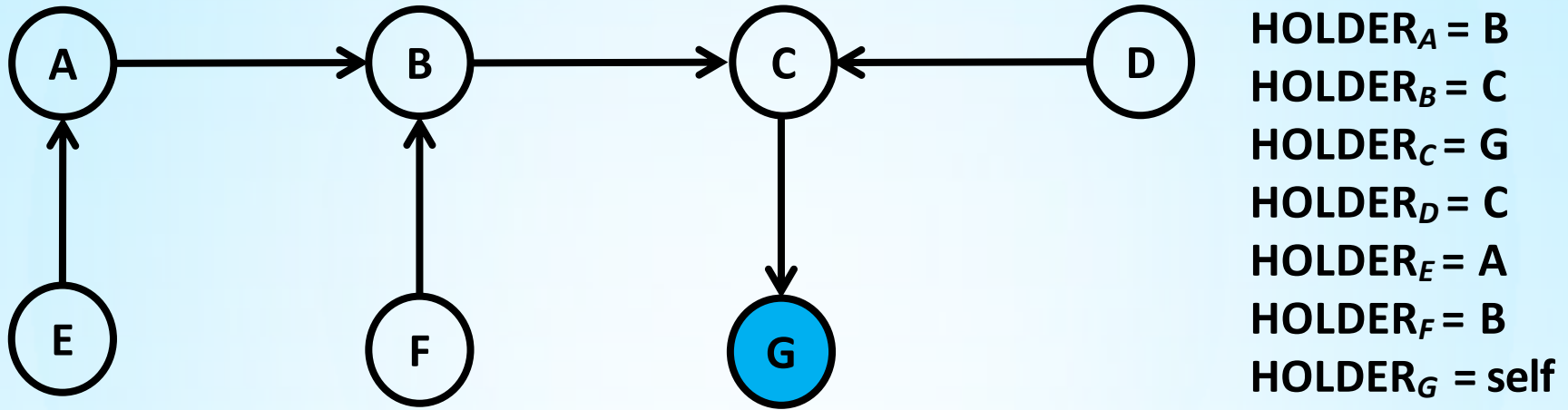


Figure 9.2: Tree with logically directed edges, all pointing in a direction towards node G - the privileged node.

Raymond's Algorithm: Example (1)

- Now suppose node B that does not hold the privilege wants to execute the critical section.
- B sends a **REQUEST** message to **HOLDER_B**, i.e., C, which in turn forwards the **REQUEST** message to **HOLDER_C**, i.e., G.
- The privileged node G, if it no longer needs the privilege, sends the **PRIVILEGE** message to its neighbor C, which made a request for the privilege, and resets **HOLDER_G** to C.
- Node C, in turn, forwards the **PRIVILEGE** to node B, since it had requested the privilege on behalf of B. Node C also resets **HOLDER_C** to B.
- The tree in Figure 9.2 will now look as in Figure 9.3.

Raymond's Algorithm: Example (1)

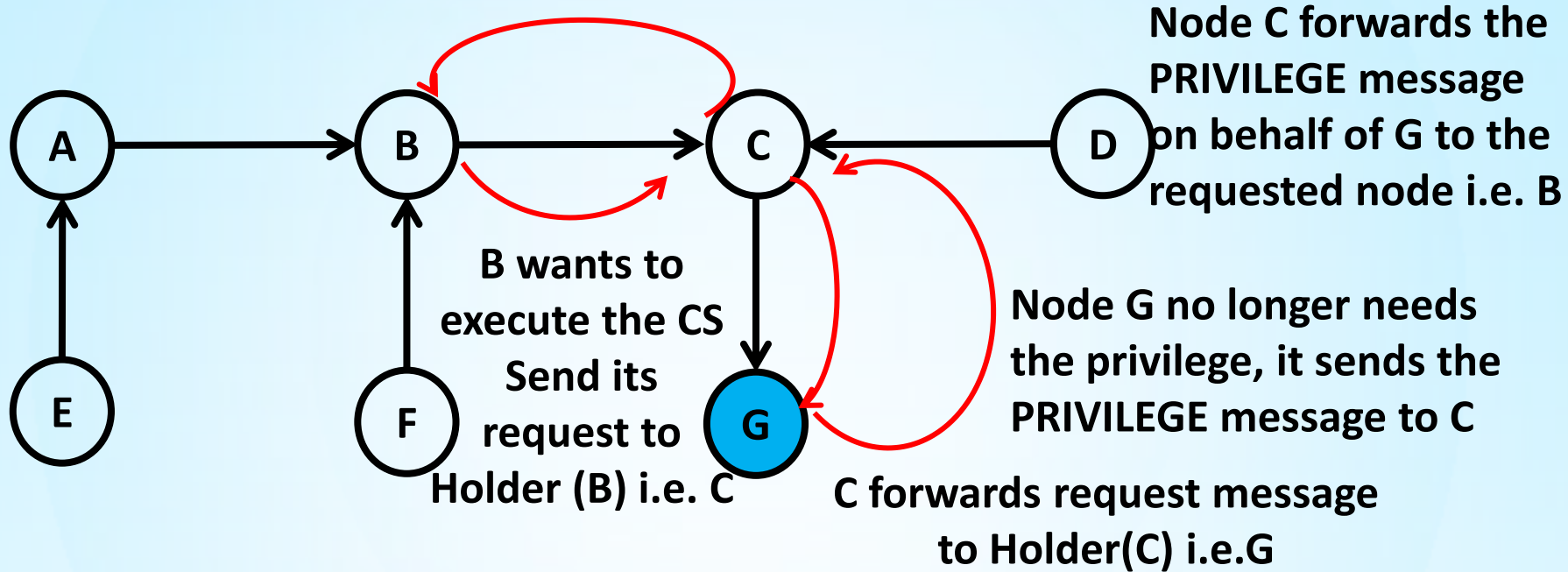


Figure 9.3: Tree with logically directed edges, all pointing in a direction towards node G - the privileged node.

Raymond's Algorithm: Example (1)

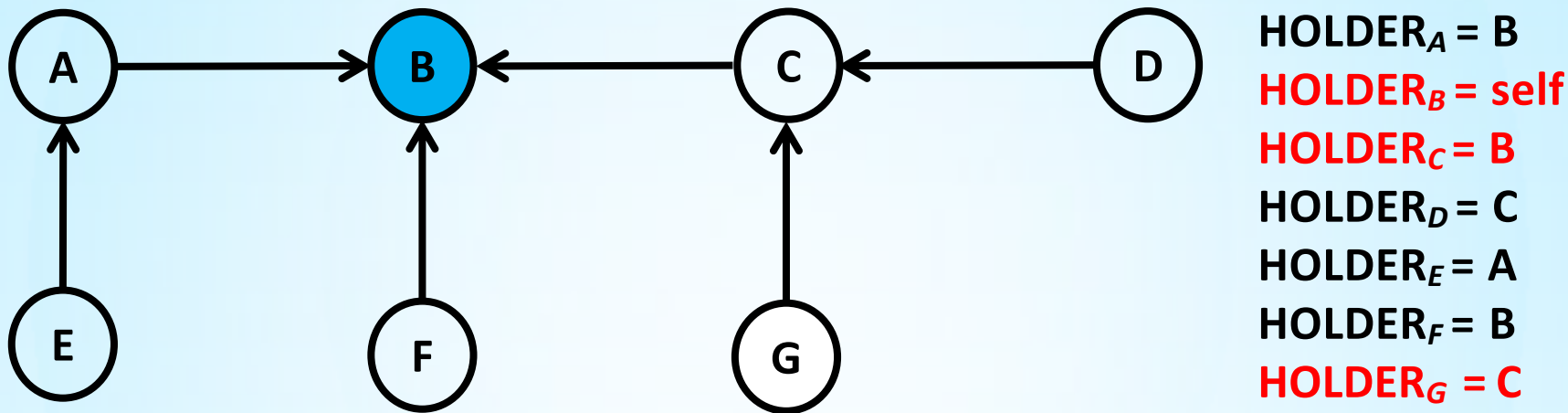
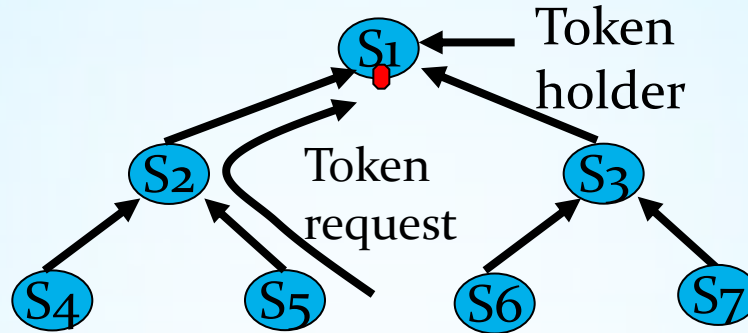


Figure 9.3 Tree with logically directed edges, all pointing in a direction towards node B - the new privileged node.

Raymond's Algorithm: Example (2)

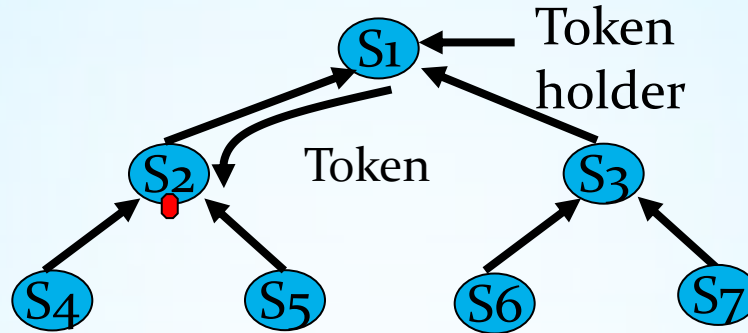
Step 1:



Site S5 sends token request

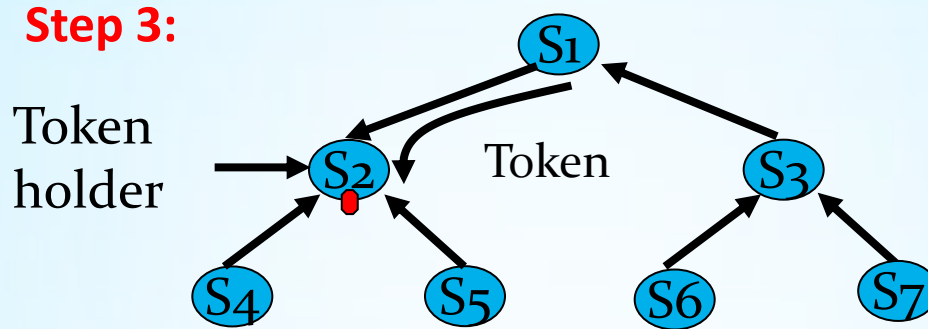
Raymond's Algorithm: Example (2)

Step 2:



Site S1 sends token to S2

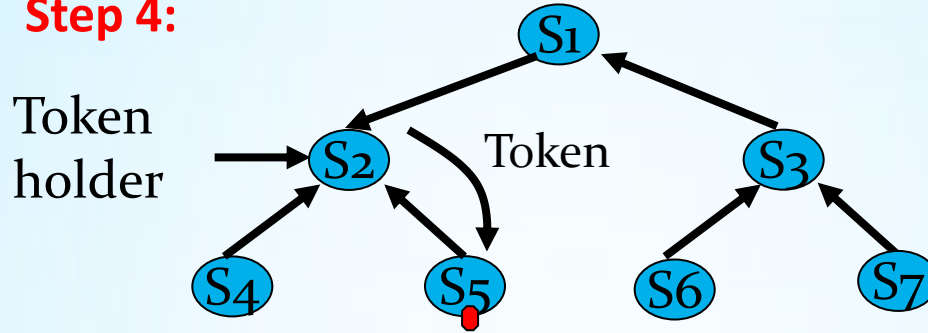
Raymond's Algorithm: Example (2)



Holder of S1 is changed to S2

Raymond's Algorithm: Example (2)

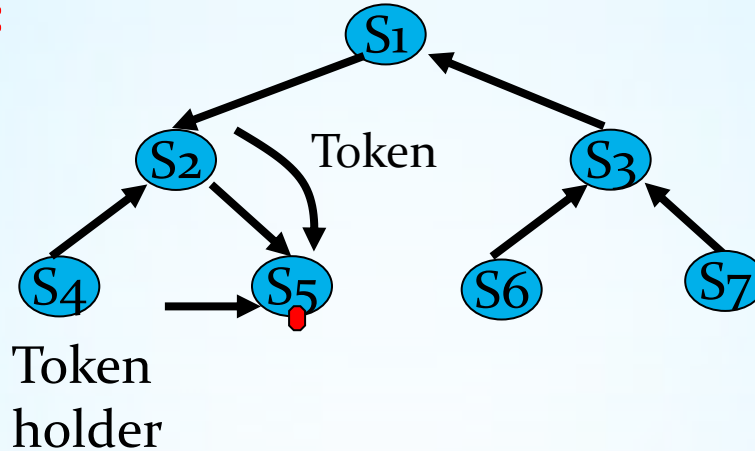
Step 4:



S2 sends token to S5

Raymond's Algorithm: Example (2)

Step 5:



**Holder of S2 is changed to S5 and
S5 becomes the new token holder**

Data Structures

- Each node to maintains the following variables:

Variable Name	Possible Values	Comments
HOLDER	“self” or the identity of one of the immediate neighbours.	Indicates the location of the privileged node in relation to the current node.
USING	True or false.	Indicates if the current node is executing the critical section.
REQUEST_Q	A FIFO queue that could contain “self” or the identities of immediate neighbors as elements.	The REQUEST_Q of a node consists of the identities of those immediate neighbors that have requested for privilege but have not yet been sent the privilege .
ASKED	True or false.	Indicates if node has sent a request for the privilege.

Data Structures

- The value “**self**” is placed in **REQUEST_Q** if the node makes a request for the privilege for its own use.
- The **maximum size** of **REQUEST_Q** of a node is the number of **immediate neighbors + 1** (for “self”).
- **ASKED** prevents the sending of duplicate requests for privilege, and also makes sure that the **REQUEST_Qs** of the various nodes do not contain any duplicate elements.

The Algorithm

- The algorithm consists of the following routines:

(i) **ASSIGN_PRIVILEGE** (ii) **MAKE_REQUEST**

(i) **ASSIGN_PRIVILEGE**: This is a routine sends a **PRIVILEGE** message. A privileged node sends a **PRIVILEGE** message if

- it holds the privilege but is not using it,
- its **REQUEST_Q** is not empty, and
- the element at the head of its **REQUEST_Q** is not “self.”

(i) ASSIGN_PRIVILEGE

- A situation where “self” is at the head of **REQUEST_Q** may occur immediately after a node receives a **PRIVILEGE** message.
- The node will enter into the critical section after removing “self” from the head of **REQUEST_Q**. If the id of another node is at the head of **REQUEST_Q**, then it is removed from the queue and a **PRIVILEGE** message is sent to that node.
- Also, the variable **ASKED** is set to false since the currently privileged node will not have sent a request for the **PRIVILEGE** message.

(ii) MAKE_REQUEST

This is a routine sends a **REQUEST** message. An unprivileged node sends a **REQUEST** message if

- it does not hold the privilege,
- its **REQUEST_Q** is not empty, i.e., it requires the privilege for itself, or on behalf of one of its immediate neighboring nodes, and
- it has not sent a **REQUEST** message already.

(ii) MAKE_REQUEST

- The variable **ASKED** is set to true to reflect the sending of the **REQUEST** message. The **MAKE_REQUEST** routine makes no change to any other variables.
- The variable **ASKED** will be true at a node when it has sent **REQUEST** message to an immediate neighbor and has not received a response.
- A node does not send any **REQUEST** messages, if **ASKED** is true at that node. Thus the variable **ASKED** makes sure that unnecessary **REQUEST** messages are not sent from the unprivileged node.
- This makes the **REQUEST_Q** of any node bounded, even when operating under heavy load.

Events

- Below we show four events that constitute the algorithm.

Event	Algorithm Functionality
A node wishes to execute critical section.	Enqueue(REQUEST_Q, self); ASSIGN_PRIVILEGE; MAKE_REQUEST
A node receives a REQUEST message from one of its immediate neighbors X.	Enqueue(REQUEST_Q, X); ASSIGN_PRIVILEGE; MAKE_REQUEST
A node receives a PRIVILEGE message.	HOLDER := self; ASSIGN_PRIVILEGE; MAKE_REQUEST
A node exits the critical section.	USING := false; ASSIGN_PRIVILEGE; MAKE_REQUEST

Events

A node wishes critical section entry:

- If it is the privileged node, the node could enter the critical section using the **ASSIGN_PRIVILEGE** routine. Otherwise, it sends a **REQUEST** message using the **MAKE_REQUEST** routine.

A node receives a REQUEST message from one of its immediate neighbors:

- If this node is the current **HOLDER**, it may send the **PRIVILEGE** to a requesting node using the **ASSIGN_PRIVILEGE** routine. Otherwise, it forwards the request using the **MAKE_REQUEST** routine.

Events

A node receives a PRIVILEGE message:

- The **ASSIGN_PRIVILEGE** routine could result in the execution of the critical section at the node, or may forward the privilege to another node. After the
- privilege is forwarded, the **MAKE_REQUEST** routine could send a **REQUEST** message to reacquire the privilege, for a pending request at this node.

A node exits the critical section:

- On exit from the critical section, this node may pass the privilege on to a requesting node using the **ASSIGN_PRIVILEGE** routine. It may then use the
- **MAKE_REQUEST** routine to get back the privilege, for a pending request at this node.

Events

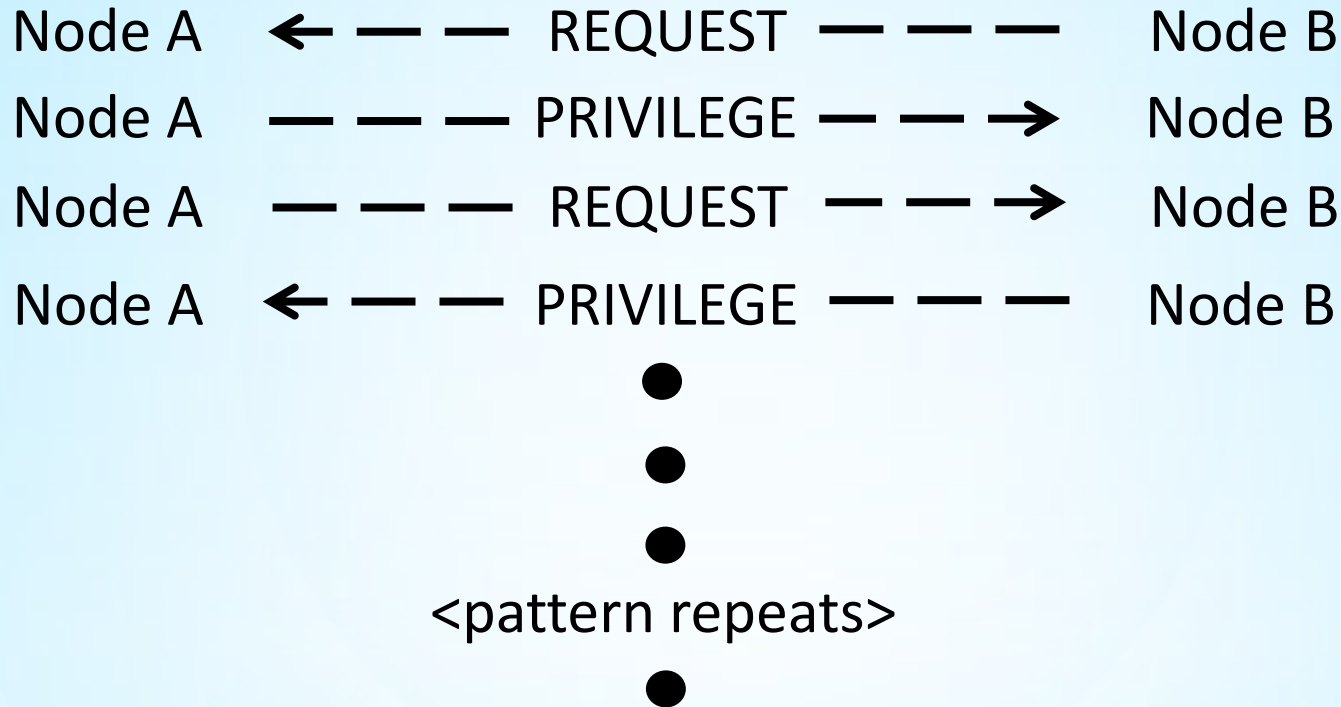


Figure 9.4: Logical pattern of message flow between neighboring nodes A and B.

Message Overtaking

- This algorithm does away with the use of sequence numbers. The algorithm works such that message flow between any two neighboring nodes sticks to a logical pattern as shown in the Figure 9.4.
- If at all message overtaking occurs between the nodes A and B, it can occur when a **PRIVILEGE** message is sent from node A to node B, which is then very closely followed by a **REQUEST** message from node A to node B.
- Such a message overtaking will not affect the operation of the algorithm.
- If node B receives the **REQUEST** message from node A before receiving the **PRIVILEGE** message from node A, A's request will be queued in **REQUEST_Q_B**. Since B is not a privileged node, it will not be able to send a privilege to node A in reply.

Message Overtaking

- When node B receives the **PRIVILEGE** message from A after receiving the **REQUEST** message, it could enter the critical section or could send a **PRIVILEGE** message to an immediate neighbor at the head of **REQUEST_** Q_B , which need not be node A. So message overtaking does not affect the algorithm.

Correctness

The algorithm provides the following guarantees:

- Mutual exclusion is guaranteed
- Deadlock is impossible
- Starvation is impossible

Mutual Exclusion

- The algorithm ensures that at any instant of time, not more than one node holds the privilege.
- Whenever a node receives a **PRIVILEGE** message, it becomes privileged. Similarly, whenever a node sends a **PRIVILEGE** message, it becomes unprivileged.
- Between the instants one node becomes unprivileged and another node becomes privileged, there is no privileged node. Thus, there is at most one privileged node at any point of time in the network.

Deadlock is Impossible

When the critical section is free, and one or more nodes want to enter the critical section but are not able to do so, a deadlock may occur. This could happen due to any of the following scenarios:

- The privilege cannot be transferred to a node because no node holds the privilege.
- The node in possession of the privilege is unaware that there are other nodes requiring the privilege.
- The **PRIVILEGE** message does not reach the requesting unprivileged node.

Deadlock is Impossible

- The **scenario 1** can never occur in this algorithm because nodes do not fail and messages are not lost.
- The logical pattern established using **HOLDER** variables ensures that a node that needs the privilege sends a **REQUEST** message either to a node holding the privilege or to a node that has a path to a node holding the privilege. Thus **scenario 2** can never occur.
- The series of **REQUEST** messages are enqueued in the **REQUEST_Qs** of various nodes such that the **REQUEST_Qs** of those nodes collectively provide a logical path for the transfer of the PRIVILEGE message from the privileged node to the requesting unprivileged nodes. So scenario 3 can never occur.

Starvation is Impossible

- When a node A holds the privilege, and another node B requests for the privilege, the identity of B or the id's of proxy nodes for node B will be present in the **REQUEST_Qs** of various nodes in the path connecting the requesting node to the currently privileged node.
- So depending upon the position of the id of node B in those **REQUEST_Qs**, node B will sooner or later receive the privilege.
- Thus once node B's **REQUEST** message reaches the privileged node A, node B, is sure to receive the privilege.

Cost and Performance Analysis

- In the **worst-case**, the algorithm requires **$(2 * \text{longest path length of the tree})$ messages** per critical section entry.
- This happens when the privilege is to be passed between nodes at either ends of the longest path of the minimal spanning tree.
- The **worst possible** network topology for this algorithm is **where all nodes are arranged in a straight line** and the longest path length will be $N - 1$, and thus the algorithm will exchange **$2 * (N - 1)$ messages** per CS execution.
- However, if all nodes generate **equal number of REQUEST messages** for the privilege, the **average number of messages** needed per critical section entry will be **approximately $2N/3$** because the **average distance between a requesting node and a privileged node** is **$(N + 1)/3$** .

Cost and Performance Analysis

- The **best topology** for the algorithm is the **radiating star topology**. The **worst case cost** of this algorithm for this topology is $O(\log_{K-1} N)$.
- Trees with higher fan-outs are preferred over radiating star topologies. The **longest path length** of such trees is typically $O(\log N)$. Thus, on an average, this algorithm involves the exchange of **$O(\log N)$ messages** per critical section execution.
- **Under heavy load**, the algorithm exhibits an interesting property: “As the number of nodes requesting for the privilege increases, the number of messages exchanged per critical section entry decreases.”
- **In heavy load**, the **algorithm requires** exchange of **only four messages** per CS execution.

Comparison of Distributed Mutual Exclusion Algorithms

Non-Token	Response Time	Synchronization Delay	Messages (Low Load)	Messages (High Load)	Concept
Lamport	$2T+E$	T	$3(N-1)$	$3(N-1)$	Give Priority with Time Stamp
Ricart-Agrawala	$2T+E$	T	$2(N-1)$	$2(N-1)$	Uses Implicit Release Message strategy.
Quorum	Response Time	Synchronization Delay	Messages (Low Load)	Messages (High Load)	Concept
Maekawa	$2T+E$	$2T$	$3 \cdot \sqrt{N}$	$5 \cdot \sqrt{N}$	Uses Theory of projective planes for quorums.
Agarwal-El Abbadi	$2T+E$	$2T$	$\log N$	$\log N$	Uses Tree structured quorums.
Token	Response Time	Synchronization Delay	Messages (Low Load)	Messages (High Load)	Concept
Suzuki-Kasami	$2T+E$	T	N	N	Token as a privilege message.
Raymond Tree	$T(\log N)+E$	$T(\log N) / 2$	$\log N$	4	Minimal Spanning Tree and unrooted tree structure

Conclusion

- **Mutual exclusion** is a fundamental problem in distributed computing systems, where concurrent access to a shared resource or data is serialized.
- In **distributed mutual exclusion algorithms**, we have discussed three types:
 - (i) **Non-Token-based approach,**
 - (ii) **Quorum based approach and**
 - (iii) **Token-based approach**
- In upcoming lecture, we will discuss about the 'consensus and agreement algorithms'.