Chapter 9: Distributed Mutual Exclusion Algorithms

Ajay Kshemkalyani and Mukesh Singhal

Distributed Computing: Principles, Algorithms, and Systems

Cambridge University Press

Introduction

- 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.

Introduction

- Distributed mutual exclusion algorithms must deal with unpredictable message delays and incomplete knowledge of the system state.
- Three basic approaches for distributed mutual exclusion:
  1. Token based approach
  2. Non-token based approach
  3. Quorum based approach
- Token-based approach:
  - A unique token 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.

Introduction

- 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.
- Quorum based approach:
  - Each site requests permission to execute the CS from a subset of sites (called a quorum).
  - Any two quorums contain a common site.
  - This common site is responsible to make sure that only one request executes the CS at any time.

Preliminaries

System Model

- The system consists of N sites, S_1, S_2, ..., S_N.
- We assume 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 the CS, executing the CS, or neither requesting nor executing the CS (i.e., idle).
- In the ‘requesting the CS’ state, the site is blocked and can not make further requests for the CS. In the ‘idle’ state, the site is executing outside the CS.
- In 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.

Requirements

Requirements of Mutual Exclusion Algorithms

1. **Safety Property:** At any instant, only one process can execute the critical section.
2. **Liveness Property:** This property states the absence of deadlock and starvation. Two or more sites should not endlessly wait for messages which will never arrive.
3. **Fairness:** Each process gets a fair chance to execute the CS. Fairness property generally means the CS execution requests are executed in the order of their arrival (time is determined by a logical clock) in the system.

Performance Metrics

The performance is generally measured by the following four metrics:

- **Message complexity:** The number of messages required per CS execution by a site.
- **Synchronization delay:** After a site leaves the CS, it is the time required and before the next site enters the CS (see Figure 1).

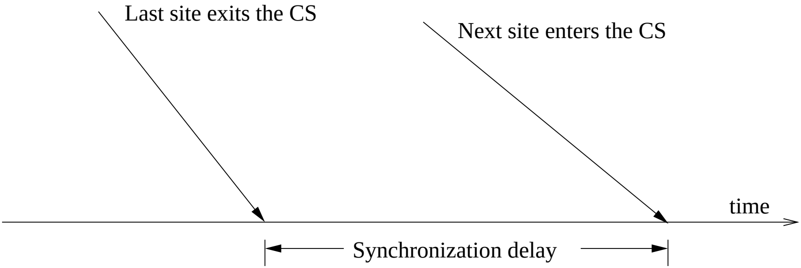


Figure 1: Synchronization Delay.

Performance Metrics

- **Response time:** The time interval a request waits for its CS execution to be over after its request messages have been sent out (see Figure 2).

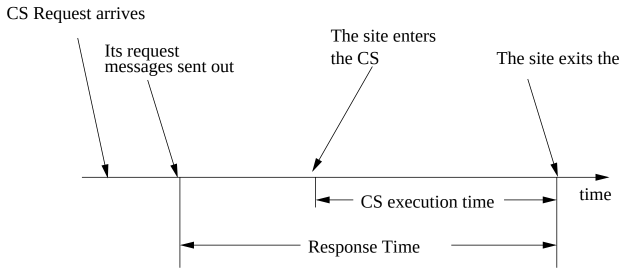


Figure 2: Response Time.

- **System throughput:** The rate at which the system executes requests for the CS.

$$\text{system throughput}=1/(SD+E)$$

where SD is the synchronization delay and E is the average critical section execution time.

Performance Metrics

Low and High Load Performance:

- We often study the performance of mutual exclusion algorithms 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.

Lamport's Algorithm

- Requests for CS are executed in the increasing order of timestamps and time is determined by logical clocks.
- Every site S_i keeps a queue, *request_queue*$_i$, which contains mutual exclusion requests ordered by their timestamps.
- 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 timestamped REPLY message to S_i.

Executing the critical section: Site S_i 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 timestamped 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.

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!

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.

An 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.

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.

Description of the Algorithm

Requesting the critical section:

(a) When a site S_i wants to enter the CS, it broadcasts a timestamped REQUEST message to all other sites.

(b) 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 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

Executing the critical section:

(c) Site S_i enters the CS after it has received a REPLY message from every site it sent a REQUEST message to.

Algorithm

Releasing the critical section:

(d) 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.

Notes:

- When a site receives a message, it updates its clock using the timestamp in the message.
- When a site takes up a request for the CS for processing, it updates its local clock and assigns a timestamp to the request.

Correctness

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_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.

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.

Singhal's Dynamic Information-Structure Algorithm

- Most mutual exclusion algorithms use a static approach to invoke mutual exclusion.
- These algorithms always take the same course of actions to invoke mutual exclusion no matter what is the state of the system.
- These algorithms lack efficiency because they fail to exploit the changing conditions in the system.
- An algorithm can exploit dynamic conditions of the system to improve the performance.

continuation..

- For example, if few sites are invoking mutual exclusion very frequently and other sites invoke mutual exclusion much less frequently, then
  - A frequently invoking site need not ask for the permission of less frequently invoking site every time it requests an access to the CS.
  - It only needs to take permission from all other frequently invoking sites.
- Singhal developed an adaptive mutual exclusion algorithm based on this observation.
- The information-structure of the algorithm evolves with time as sites learn about the state of the system through messages.

continuation..

Challenges

The design of adaptive mutual exclusion algorithms is challenging:

- How does a site efficiently know what sites are currently actively invoking mutual exclusion?
- When a less frequently invoking site needs to invoke mutual exclusion, how does it do it?
- How does a less frequently invoking site makes a transition to more frequently invoking site and vice-versa.
- How to insure that mutual exclusion is guaranteed when a site does not take the permission of every other site.
- How to insure that a dynamic mutual exclusion algorithm does not waste resources and time in collecting systems state, offsetting any gain.

System Model

- We consider a distributed system consisting of n autonomous sites, say, S_1, S_2, ..., S_n, connected by a communication network.
- We assume that the sites communicate completely by message passing.
- Message propagation delay is finite but unpredictable.
- Between any pair of sites, messages are delivered in the order they are sent.
- The underlying communication network is reliable and sites do not crash.

Data Structures

- Information-structure at a site S_i consists of two sets. The first set R_i, called *request set*, contains the sites from which S_i must acquire permission before executing CS.
- The second set I_i, called *inform set*,contains the sites to which S_i must send its permission to execute CS after executing its CS.
- Every site S_i maintains a logical clock C_i, which is updated according to Lamport's rules.
- Every site maintains three boolean variables to denote the state of the site: *Requesting, Executing, and My_priority*.
- Requesting and executing are true if and only if the site is requesting or executing CS, respectively. My_priority is true if pending request of S_i has priority over the current incoming request.

Initialization

The system starts in the following initial state:
For a site S_i (i = 1 to n),

$$R_i := \{S_1, S_2, \ldots, S_i - 1, S_i\}$$
$$I_i := \{S_i\}$$
$$C_i := 0$$
$$Requesting_i = Executing_i := \text{False}$$

continuation..

If we stagger sites S_n to S_1 from left to right, then the initial system state has the following two properties:

- For a site, only all the sites to its left will ask for its permission and it will ask for the permission of only all the sites to its right.
- The cardinality of R_i decreases in stepwise manner from left to right. Due to this reason, this configuration has been called ”staircase pattern” in topological sense .

The Algorithm

```
Step 1: (Request Critical Section)
            Requesting = true;
            $C_i = C_i + 1$;
            Send REQUEST($C_i$, i) message to all sites in $R_i$;
            Wait until $R_i = \emptyset$;
            /* Wait until all sites in $R_i$ have sent a reply to $S_i$ */
            Requesting = false;
Step 2: (Execute Critical Section)
            Executing = true;
            Execute CS;
            Executing = false;
```

The Algorithm

```
Step 3: (Release Critical Section)
        For every site $S_k$ in $I_i$ (except $S_i$) do
            Begin
                $I_i = I_i - \{S_k\}$;
                Send REPLY($C_{i,}$ i) message to $S_k$;
                $R_i = R_i + \{S_k\}$
            End
```

The Algorithm

REQUEST message handler

```
/* Site $S_i$ is handling message REQUEST(c, j) */
$C_i$ := max{$C_i$, c};
Case
Requesting = true:
Begin if My_priority then $I_i := I_i + \{_j\}$
/*My_Priority true if pending request of $S_i$ has priority over incoming request */
Else
Begin
    Send REPLY($C_i$, i) message to $S_j$;
    If not ($S_j \in R_i$) then
        Begin
            $R_i = R_i + \{S_j\}$;
            Send REQUEST($C_i$, i) message to site $S_j$;
        End;
End;
Executing = true: $I_i = I_i + \{S_j\}$;
Executing = false $\wedge$ Requesting = false:
Begin
    $R_i = R_i + \{S_j\}$;
    Send REPLY($C_i$, i) message to $S_j$;
End;
```

The Algorithm

REPLY message handler

```
/* Site $S_i$ is handling a message REPLY(c, j) */
Begin
  $C_i := \max\{C_i, c\}$;
  $R_i = R_i - \{S_j\}$;
End;
```

- Note that REQUEST and REPLY message handlers and the steps of the algorithm access shared data structures, viz., C_i, R_i, and I_i.
- To guarantee the correctness, it's important that execution of REQUEST and REPLY message handlers and all three steps of the algorithm (except "wait for $R_i = \emptyset$ to hold" in Step 1) mutually exclude each other.

An Explanation of the Algorithm

- S_i acquires permission to execute the CS from all sites in its request set R_i and it releases the CS by sending a REPLY message to all sites in its inform set I_i.
- If site S_i which itself is requesting the CS, receives a higher priority REQUEST message from a site S_j, then S_i takes the following actions:
  1. (i) S_i immediately sends a REPLY message to S_j,
  2. (ii) if S_j is not in R_i, then S_i also sends a REQUEST message to S_j, and
  3. (iii) S_i places an entry for S_j in R_i. Otherwise, S_i places an entry for S_j into I_i so that S_j can be sent a REPLY message when S_i finishes with the execution of the CS.
- If S_i receives a REQUEST message from S_j when it is executing the CS, then it simply puts S_j in I_i so that S_j can be sent a REPLY message when S_i finishes with the execution of the CS.
- If S_i receives a REQUEST message from S_j when it is neither requesting nor executing the CS, then it places an entry for S_j in R_i and sends S_j a REPLY message.

Correctness

Achieving Mutual Exclusion:

- The initial state of the information-structure satisfies the following condition: for every S_i and S_j, either $S_j \in R_i$ or $S_i \in R_j$.
- Therefore, if two sites request CS, one of them will always ask for the permission of the another.
- However, whenever there is a conflict between two sites, the sites dynamically adjust their request sets such that both request permission of each other satisfying the condition for mutual exclusion.

Freedom from Deadlocks:

- The algorithm is free from deadlocks because sites use timestamp ordering (which is unique system wide) to decide request priority and a request is blocked by only higher priority requests.

Performance Analysis

-The synchronization delay in the algorithm is T.

The message complexity:

Low load condition:

- Most of the time only one or no request for the CS will be present in the system.
- The staircase pattern will reestablish between two successive requests for CS.
- Sites will send 0, 1, 2, ..., (n-1) number of REQUEST messages with equal likelihood (assuming uniform traffic of CS requests at sites).
- Therefore, the mean number of REQUEST messages sent per CS execution for this case is $=(0+1+2+ \ldots\ldots.. +(n-1))/n = (n-1)/2$. Since a REPLY message is returned for every REQUEST message, the average number of messages exchanged per CS execution is $2^*(n-1)/2 = (n-1)$.

continuation..

Heavy load condition:

- When the rate of CS requests is high, all the sites always have a pending request for CS execution.
- In this case, a site on the average receives (n-1)/2 REQUEST messages from other sites while waiting for its REPLY messages.
- Since a site sends REQUEST messages only in response to REQUEST messages of higher priority, on the average it will send (n-1)/4 REQUEST messages while waiting for REPLY messages.
- Therefore, the average number of messages exchanged per CS execution in high demand is 2*[(n-1)/2+(n-1)/4] = 3*(n-1)/2.

Adaptivity in Heterogeneous Traffic Patterns

- The information-structure adapts itself to the environments of heterogeneous traffic of CS requests and to statistical fluctuations in traffic of CS requests to optimize the performance.
- Sites with higher traffic of CS requests will position themselves towards the right end of the staircase pattern.
- Also, at a high traffic site S_i, if $S_j \in R_i$, then S_j is also a high traffic site.
- Consequently, high traffic sites will mostly send REQUEST messages only to other high traffic sites and will seldom send REQUEST messages to sites with low traffic.
- This adaptivity results in a reduction in the number of messages as well as in delay in granting CS in environments of heterogeneous traffic.

Quorum-Based Mutual Exclusion Algorithms

Quorum-based mutual exclusion algorithms are different in the following two ways:

1. A site does not request permission from all other sites, but only from a subset of the sites. The request set of sites are chosen such that ∀i ∀j : $1 \leq i, j \leq N :: R_i \cap R_j \neq \Phi$. Consequently, every pair of sites has a site which mediates conflicts between that pair.
2. A site can send out only one REPLY message at any time. A site can send a REPLY message only after it has received a RELEASE message for the previous REPLY message.

continuation..

Since these algorithms are based on the notion of 'Coteries' and 'Quorums', we next describe the idea of coteries and quorums.
A coterie C is defined as a set of sets, where each set g $\in$ C is called a quorum.
The following properties hold for quorums in a coterie:

- **Intersection property:** For every quorum g, h $\in$ C, g $\cap$ h$\neq \emptyset$.
  For example, sets {1,2,3}, {2,5,7} and {5,7,9} cannot be quorums in a coterie because the first and third sets do not have a common element.
- **Minimality property:** There should be no quorums g, h in coterie C such that g $\supseteq$ h. For example, sets {1,2,3} and {1,3} cannot be quorums in a coterie because the first set is a superset of the second.

continuation..

Coteries and quorums can be used to develop algorithms to ensure mutual exclusion in a distributed environment. A simple protocol works as follows:

- Let ‘a’ is a site in quorum ‘A’. If ‘a’ wants to invoke mutual exclusion, it requests permission from all sites in its quorum ‘A’.
- Every site does the same to invoke mutual exclusion. Due to the Intersection Property, quorum ‘A’ contains at least one site that is common to the quorum of every other site.
- These common sites send permission to only one site at any time. Thus, mutual exclusion is guaranteed.

Note that the Minimality property ensures efficiency rather than correctness.

Maekawa's Algorithm

Maekawa's algorithm was the first quorum-based mutual exclusion algorithm. The request sets for sites (i.e., quorums) in Maekawa's algorithm are constructed to satisfy the following conditions:

M1: $(\forall i \ \forall j : i \neq j, 1 \leq i, j \leq N :: \mathrm{R}_i \cap \mathrm{R}_j \neq \phi)$

M2: $(\forall i : 1 \leq i \leq N :: S_i \in \mathrm{R}_i)$

M3: $(\forall i : 1 \leq i \leq N :: |R_i| = K)$

M4: Any site S_j is contained in K number of R_is, $1 \leq i, j \leq N$.

Maekawa used the theory of projective planes and showed that $N = K(K-1) + 1$. This relation gives $|R_i| = \sqrt{N}$.

continuation..

- 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.

The Algorithm

A site S_i executes the following steps to execute the CS.

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.

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.

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. □

Performance

- Since the size of a request set is $\sqrt{N}$, an execution of the CS requires $\sqrt{N}$ REQUEST, $\sqrt{N}$ REPLY, and $\sqrt{N}$ RELEASE messages, resulting in $3\sqrt{N}$ 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.

Problem of Deadlocks

- 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.

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 $5\sqrt{N}$.

Agarwal-El Abbadi Quorum-Based Algorithm

Agarwal-El Abbadi quorum-based algorithm uses ‘tree-structured quorums’.

- All the sites in the system are logically organized into a complete binary tree.
- For a complete binary tree with level ‘k’, we have $2^{k+1} - 1$ sites with its root at level k and leaves at level 0.
- The number of sites in a path from the root to a leaf is equal to the level of the tree k+1 which is equal to O(log n).
- A path in a binary tree is the sequence a_1, $a_2 \ldots a_i$, $a_{i+1} \ldots .$ a_k such that a_i is the parent of a_{i+1}.

Algorithm for constructing a tree-structured quorum

- The algorithm tries to construct quorums in a way that each quorum represents any path from the root to a leaf.
- If it fails to find such a path (say, because node 'x' has failed), the control goes to the ELSE block which specifies that the failed node 'x' is substituted by two paths both of which start with the left and right children of 'x' and end at leaf nodes.
- If the leaf site is down or inaccessible due to any reason, then the quorum cannot be formed and the algorithm terminates with an error condition.
- The sets that are constructed using this algorithm are termed as *tree quorums*.

```
FUNCTION GetQuorum (Tree: NetworkHierarchy): QuorumSet;
  VAR left, right : QuorumSet;
  BEGIN
  IF Empty (Tree) THEN
       RETURN ({});
  ELSE IF GrantsPermission(Tree↑.Node) THEN
                RETURN ((Tree↑.Node) $\cup$ GetQuorum (Tree↑.LeftChild));
                OR
                RETURN ((Tree↑.Node) $\cup$ GetQuorum (Tree↑.RightChild));(*line 9*)
         ELSE
                left←GetQuorum(Tree↑.left);
                right←GetQuorum(Tree↑.right);
                IF (left $= \emptyset \vee$ right $= \emptyset$) THEN
                    (* Unsuccessful in establishing a quorum *)
                     EXIT(-1);
                ELSE
                    RETURN (left $\cup$ right);
                END; (* IF *)
         END; (* IF *)
  END; (* IF *)
END GetQuorum
```

Figure: Algorithm for constructing a tree-structured quorum.

Examples of Tree-Structured Quorums

When there is no node failure, the number of quorums formed is equal to the number of leaf sites.

- Consider the tree of height 3 show in Figure 3, constructed from 15 (=2^{3+1}-1) sites.
- In this case 8 quorums are formed from 8 possible root-leaf paths: 1-2-4-8, 1-2-4-9, 1-2-5-10, 1-2-5-11, 1-3-6-12, 1-3-6-13, 1-3-7-14 and 1-3-7-15.
- If any site fails, the algorithm substitutes for that site two possible paths starting from the site's two children and ending in leaf nodes.
- For example, when node 3 fails, we consider possible paths starting from children 6 and 7 and ending at leaf nodes. The possible paths starting from child 6 are 6-12 and 6-13, and from child 7 are 7-14 and 7-15.
- So, when node 3 fails, the following eight quorums can be formed: {1,6,12,7,14}, {1,6,12,7,15}, {1,6,13,7,14}, {1,6,13,7,15}, {1,2,4,8}, {1,2,4,9},{1,2,5,10}, {1,2,5,11}.

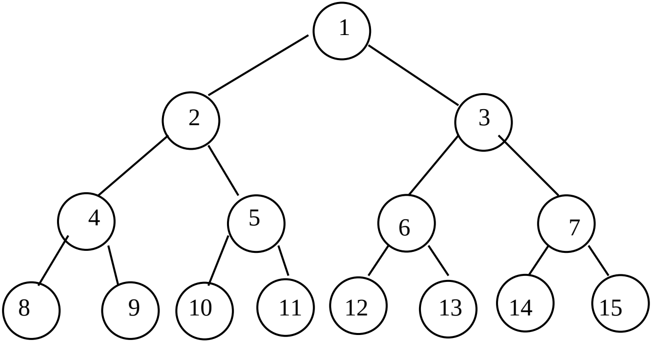


Figure 3: A tree of 15 sites.

Examples of Tree-Structured Quorums

- Since the number of nodes from root to leaf in an ‘n’ node complete tree is *log n*, the best case for quorum formation, i.e, the least number of nodes needed for a quorum is *log n*.
- When the number of node failures is greater than or equal to *log n*, the algorithm may not be able to form tree-structured quorum.
- So, as long as the number of site failures is less than log n, the tree quorum algorithm gurantees the formation of a quorum and it exhibits the property of ‘graceful degradation’.

Mutual Exclusion Algorithm

A site s enters the critical section (CS) as follows:

- Site *s* sends a 'Request' message to all other sites in the structured quorum it belongs to.
- Each site in the quorum stores incoming requests in a *request queue*, ordere by their timestamps.
- A site sends a 'Reply' message, indicating its consent to enter CS, only to the request at the head of its *request queue*, having the lowest timestamp.
- If the site *s* gets a 'Reply' message from all sites in the structured quorum it belongs to, it enters the CS.
- After exiting the CS, *s* sends a 'Relinquish' message to all sites in the structured quorum. On the receipt of the 'Relinquish' message, each site removes *s*'s request from the head of its *request queue*.

Mutual Exclusion Algorithm

- If a new request arrives with a timestamp smaller than the request at the head of the queue, an 'Inquire' message is sent to the process whose request is at the head of the queue and waits for a 'Yield' or 'Relinquish' message.
- When a site *s* receives an 'Inquire' message, it acts as follows:
  If *s* has acquired all of its necessary replies to access the CS, then it simply ignores the 'Inquire' message and proceeds normally and sends a 'Relinquish' message after exiting the CS.
  If *s* has not yet collected enough replies from its quorum, then it sends a 'Yield' message to the inquiring site.
- When a site gets the 'Yield' message, it puts the pending request (on behalf of which the 'Inquire' message was sent) at the head of the queue and sends a 'Reply' message to the requestor.

Correctness proof

Mutual exclusion is guaranteed because the set of quorums satisfy the Intersection property.

- Consider a coterie C which consists of quorums {1,2,3}, {2,4,5} and {4,1,6}.
- Suppose nodes 3, 5 and 6 want to enter CS, and they send requests to sites (1, 2), (2, 4) and (1, 4), respectively.
- Suppose site 3's request arrives at site 2 before site 5's request. In this case, site 2 will grant permission to site 3's request and reject site 5's request.
- Similarly, suppose site 3's request arrives at site 1 before site 6's request. So site 1 will grant permission to site 3's request and reject site 6's request.
- Since sites 5 and 6 did not get consent from all sites in their quorums, they do not enter the CS.
- Since site 3 alone gets consent from all sites in its quorum, it enters the CS and mutual exclusion is achieved.

Token-Based Algorithms

- 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.)

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.

continuation..

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 can not determined 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.

continuation..

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.

continuation..

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.

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.

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

- 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 4 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.

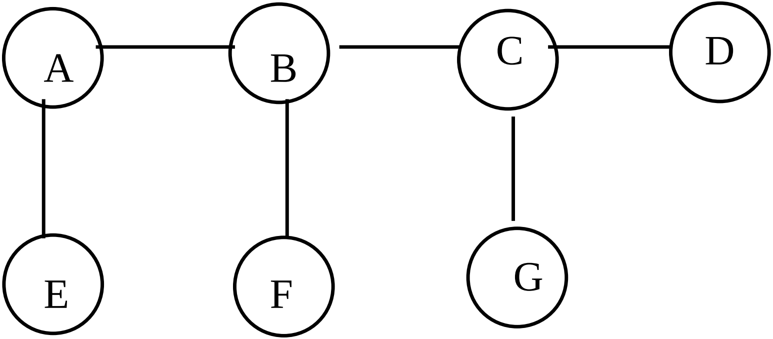


Figure 4: A tree of 15 sites.

Raymond's Tree-Based Algorithm

- 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 $\text{HOLDER}_X = \text{Y}$, 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 4 can be redrawn with logically directed edges as shown in the Figure 5.

The HOLDER Variables

- The shaded node in Figure 5 represents the privileged node.
- The following will be the values of the HOLDER variables of various nodes:
  $\text{HOLDER}_A = \text{B}$
  $\text{HOLDER}_B = \text{C}$
  $\text{HOLDER}_C = \text{G}$
  $\text{HOLDER}_D = \text{C}$
  $\text{HOLDER}_E = \text{A}$
  $\text{HOLDER}_F = \text{B}$
  $\text{HOLDER}_G = \text{self}$

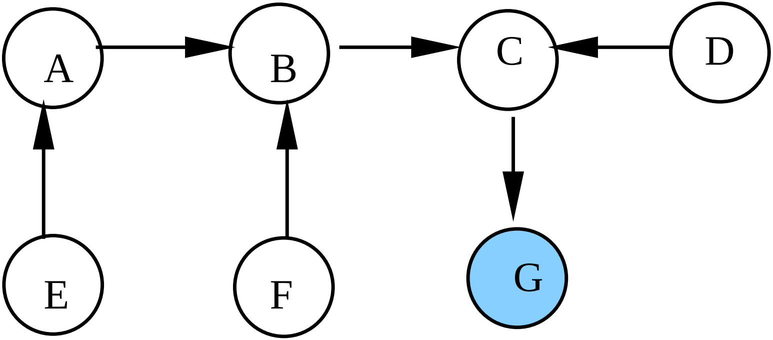


Figure 5: Tree with logically directed edges, all pointing in a direction towards node G - the privileged node.

The HOLDER Variables

- 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 5 will now look as in Figure 6.

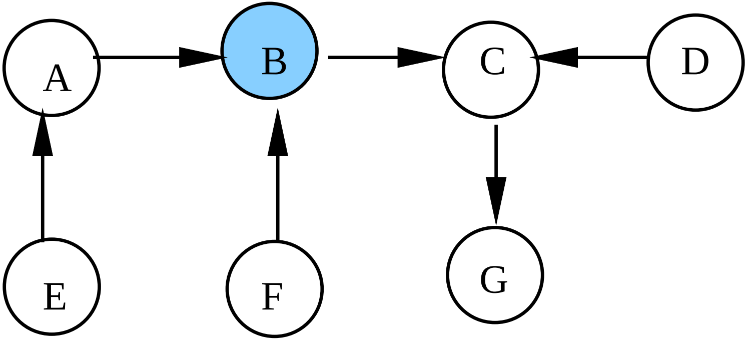


Figure 6: Tree with logically directed edges, all pointing in a direction towards node G - the privileged node.

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:

- ASSIGN_PRIVILEGE
- MAKE_REQUEST

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."

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.

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.

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..

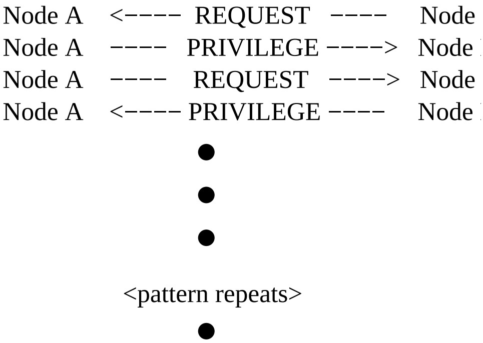
A node wishes critical section entry:
If it is the privileged node, the node could enter the critical section using the ASSIGN_PRIVILEGE routine. Otherwise, it sends a REQUEST message using the MAKE_REQUEST routine.

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.



Node A <---- REQUEST ---- Node

Node A ---- PRIVILEGE ----> Node

Node A ---- REQUEST ----> Node

Node A <---- PRIVILEGE ---- Node

●

●

●

<pattern repeats>

●

Figure 7: 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 7.
- 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.
- 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:

1. The privilege cannot be transferred to a node because no node holds the privilege.
2. The node in possession of the privilege is unaware that there are other nodes requiring the privilege.
3. 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 tranfer 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 * 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.