Department of

Computer Science

A Survey of Distributed Mutual Exclusion Algorithms

Martin G. Velazquez

Technical Report CS-93-116

September 6, 1993

Colorado State University

A SURVEY OF DISTRIBUTED MUTUAL EXCLUSION ALGORITHMS

1. INTRODUCTION.

Over the last decade distributed computing systems have attracted a great deal of attention. This is due, in part, to the technological advances in the design of sophisticated software and communication interfaces, the availability of low-cost processors and the rapid decline in hardware costs. The motivations for building distributed computing systems are many. Resource sharing, parallel processing, system availability and communication are four major reasons. By distributing a computation among various sites, processes are allowed to run concurrently and to share resources, but still work independently of each other.

Many distributed computations involving the sharing of resources among various processes require that a resource be allocated to a single process at a time. Therefore, mutual exclusion is a fundamental problem in any distributed computing system. This problem must be solved to synchronize the access to shared resources in order to maintain their consistency and integrity. The major goal of this paper is to get the reader acquainted with all relevant major approaches for solving the mutual exclusion problem in distributed computing systems.

This paper describes the principles and characteristics of diverse distributed mutual exclusion algorithms for distributed computing systems. Their major design approaches, the assumptions made about the distributed environment, and the order of magnitude of message complexity will be described. Three papers are presented that introduce a new concept in allowing multiple processes to enter a critical section simultaneously.

The rest of the paper is organized as follows: in Section 2, the requirements for mutual exclusion algorithms are formulated and characteristics and assumptions about the distributed environment are discussed. In Section 3, distributed mutual exclusion algorithms are classified by two basic design approaches, and the two approches are described. Sections 4 and 5 are dedicated to the description of diverse distributed mutual exclusion algorithms grouped by their major design approach. In Section 6, three algorithms that are designed to allow multiple simultaneous entries to the critical section are described. Concluding remarks are presented in Section 7.

2. MUTUAL EXCLUSION IN DISTRIBUTED COMPUTING SYSTEMS

A distributed computing system is a collection of autonomous computing sites that do not share a global or common memory and communicate solely by exchanging messages over a communication facility.

In a distributed computing system any given site (also refered to as "node") has only a partial or incomplete view of the total system and a system-wide common clock does not exist. Processes must share common hardware or software resources, cooperating in such a way that they can work in parallel and independently of each other. The access to a shared resource must be synchronized to ensure that only one process is making use of the resource at a given time.

Each process has a code segment, called a critical section, in which the process can access the shared resource. The problem of coordinating the execution of critical sections by each process is solved by providing mutually exclusive access in time to the critical section. A process is said to execute repeatedly a sequence of non-critical section code and critical section code segments, each of finite execution time. Each process must request permission to enter its critical section and must release it after it has completed its execution.

A mutual exclusion algorithm must satisfy the following requirements [23, 10]:

1. At most one process can execute its critical section at a given time.

2. If no process is in its critical section, any process requesting to enter its critical section must be allowed to do so in finite time.

3. When competing processes concurrently request to enter their respective critical sections, the selection cannot be postponed indefinitely.

4. A requesting process can not be prevented by another one to enter its critical section within a finite delay.

To simplify, an algorithm must provide mutually exclusive access to a resource, ensure deadlock freedom, ensure starvation freedom, and must provide some fairness in the order that requests are granted.

Two approaches can be used to implement a mutual exclusion mechanism in a distributed computing system. In a centralized approach, one of the nodes functions as a central coordinator. Processes ask only the coordinator for permission to enter their critical section. Only when a requesting process receives permission from the coordinator can it proceed to enter its critical section. The central coordinator is fully responsible for having all the information of the system and for granting permission to make use of a shared resource.

In a distributed approach, the decision making is distributed across the entire system and the solution to the mutual exclusion problem is far more complicated because of the difficulty to obtain a complete knowledge of the total system. This is due to the lack of a common shared memory, a common physical clock and because of unpredictable message delay.

Only distributed algorithms will be presented in this paper. The characteristics and assumptions made in their design about the distributed environment are discussed below.

2.1 The distributed mutual exclusion model.

The effectiveness of an algorithm depends on the suitability of the model as well as the validity of the assumptions made about the distributed environment. All the algorithms presented in this paper share in their design the following assumptions and conditions for the distributed environment:

1. All nodes in the system are assigned unique identification numbers from 1 to N.
2. There is only one requesting process executing at each node. Mutual exclusion is implemented at the node level.
3. Processes are competing for a single resource.
4. At any time, each process initiates at most one outstanding request for mutual exclusion.
5. All the nodes in the system are fully connected.

When reviewing an algorithm, attention should be paid to the assumptions made about the communications network. This is very important because nodes communicate only by exchanging messages with each other. The following aspects about the reliability of the underlying communications network should be considered.

Message delivery guaranteed. Messages are not lost or altered and are correctly delivered to their destination in a finite amount of time.

Message-order preservation. Messages are delivered in the order they are sent. There is no message overtaking.

Message transfer delays are finite, but unpredictable. Messages reach their destination in a finite amount of time, but the time of arrival is variable.

The topology of the network is known. Nodes know the physical layout of all nodes in the system and know the path to reach each other.

Early algorithms did not consider fault-tolerance issues. An algorithm in a distributed computing system must consider fault-tolerance aspects to detect and recover from failures. A resilient algorithm takes advantage of the high availability of the system in a distributed environment. Even when nodes fail, the rest of the system can still work, albeit with a degraded performance.

Due to the nature of a distributed environment, many failures can occur. These can take place either in a channel or in a node, or in both. The following failures should be considered for an algorithm to be resilient.

1. Channel failure, supression/shutdown and recovery/insertion.
2. Node failure, supression/shutdown and recovery/insertion.
3. Network partitioning and system reconfiguration.
4. Complete and partial node failures. Nodes completely fail or behave maliciously.

Recent algorithms incorporate fault-tolerance mechanisms to detect and recover from some failures in the system. Another aspect to consider in an algorithm, is the amount of information that is maintained by each node in the system and how it is used. The state information about the total system can be used to reduce the message traffic in the network and to recover from failures.

The performance of the algorithms presented here will be evaluated using the total number of messages required for a node to enter the critical section as a criterion. Message traffic should be minimized in order to decrease the overhead in the communications network. In resilient algorithms, failures in the system need the exchange of more information than in normal operation, in order to reconstruct the state of the system as it was before the failure. Hence, their performance will be evaluated under normal conditions.

Distributed mutual exclusion algorithms are designed based on two basic principles [19, 24]: the existence of a token in the system, or the collection of permission from nodes in the system. These two approaches are described in the following section.

3. BASIC APPROACHES FOR THE DESIGN OF DISTRIBUTED MUTUAL EXCLUSION ALGORITHMS

Distributed mutual exclusion algorithms can be classified into two groups by a basic principle in their design [19, 24]. These two groups are token-based algorithms and permission-based algorithms. The basic principle for the design of a distributed mutual exclusion algorithm is the way in which the right to enter the critical section is formalized in the system.

3.1 The token-based approach.

In the token-based group the right to enter a critical section is materialized by a special object, namely a token. The token is unique in the whole system. Processes requesting to enter their critical section are allowed to do so when they possess the token. The token gives to a process the privilege of entering the critical section. A token is a special type of message. The singular existence of the token implies the enforcement of mutual exclusion. Only one process, the one holding the token, is allowed to enter to its critical section. At any given time the token must be possessed by one process at most.

Granting the privilege to enter the critical section is performed by a single process, which is the current owner of the token. This process chooses the next token owner and sends it the token.

A distinction has to be made between the mechanisms used to move the token among the processes in the system [19, 23]. If processes are logically organized in a direct ring structure, the token can travel around the ring from process to process to give them the right to enter the critical section. If a process receives the token and it is interested in the critical section (CS), it can proceed to its execution. After the process exits its CS the token is released to circulate again. On the other hand, if the process is not interested in its CS it just passes the token to the next node in the logical ring. If the ring is unidirectional, starvation freedom is ensured. Under light load this method has a high cost since the token message circulates even if no process wants to enter the CS, but it is very effective under high load.

Another method to move the token in the system is by asking for it when a process wants to enter its CS. A requesting process sends a request message to the token holder and waits for the token arrival. After completing the execution of its CS, the process holding the token chooses a requesting process and sends it the token. If no process wants to use the token, the token holder does not need to send the token away. Using this method a major concern is how to locate the token holder in order to minimize message exchanges originated by a requesting process.

The token-based approach is highly susceptible to the loss of the token, since this can induce a deadlock situation. Also, problems can occur with the existence of duplicated tokens. Complex token regeneration must be executed to ensure the uniqueness of the token.

3.2 The permission-based approach.

In the permission-based group the right to enter a critical section is formalized by receiving permission from a set of nodes in the system. A process wishing to enter its critical section asks the others to give it their permission to proceed; and then it waits until these permissions have arrived. A process enters its CS only after receiving permission from all nodes in a set.

Non-requesting processes send their permission to requesting ones. Each process may grant its permission to only one process at a time. A priority or an order of events has to be established between competing requesting processes so only one of them receives permission from all other nodes in the set. Only one process, the one that has received permission from all members of a given set of nodes, is allowed to enter the critical section. This enforces the requirement for mutual exclusion.

Granting the privilege to enter the critical section is performed by the set of nodes that send their permission to requesting processes. Conflicts are solved by a priority or an order of events mechanism.

The problem of finding a minimal number of nodes from which a process has to obtain permission to enter its CS has to be considered. This can be translated as to how many rights does a process have to collect in order to proceed to the execution of the critical section. Many protocols have been developed to find a majority or quorum of processes from which rights have to be collected. The solution to this problem has a direct impact in the cost of messages exchanged per mutual exclusion invocation.

4. DESCRIPTION OF TOKEN-BASED ALGORITHMS.

In these algorithms a special message called "token" or "privilege" is used to pass the right of entering the critical section among a group of uncoordinated, but cooperative processes. The singular existence of the token directly implies that only a single node is allowed to enter the critical section. This provides for mutual exclusion access to shared resources.

Deadlock occurs if no node is in its critical section and there are two or more processes wishing to enter the CS, but they are not able to do so. This could occur essentially if the token is lost or does not eventually reach a node which has requested it. The loss of the token cannot be easily distinguished from system connectivity loss. The existence of more than one token would violate the mutual exclusion requirement, thus the detection of a token loss is not a trivial task.

All the algorithms presented in this section claim to satisfy the mutual exclusion requirement, be deadlock free and starvation free. Some of them do not consider fault-tolerance aspects, and some make use of information about the state of the system or impose a logical structure to reduce the maximal number of messages to be exchanged for a critical section entry.

Their major characteristics and assumptions will be discussed and the number of messages exchanged for an entry to the critical section to take effect will be used as a complexity measure to compare them. The first four algorithms described do not consider fault-tolerance aspects [21, 27, 12, 14]. The next three discuss the effect of failures and suggest recovery mechanisms that can be incorporated to the algorithm [6, 16, 24]. The last three algorithms [13, 11, 15], incorporate some mechanisms for the recovery from different kinds of failures, the regeneration of a new token, the elimination of duplicate tokens, and the reconstruction of the system after a failure. All algorithms in each group are presented in chronological order.

Ricart-Agrawala algorithm. In their algorithm [21] each node keeps a request sequence number counter and maintains an array request_data of size N. This array holds information about the most recent request received from all other nodes in the system. When a node wants to enter its critical section and does not have the token, it increments its request sequence number counter and sends a request message of the form REQUEST(my_sequence_number, my_id), to all other N-1 nodes. It then waits for the arrival of the token message. If it has the token or the token arrives, it can proceed to enter its critical section.

The token message has the form TOKEN(token_data) where token_data is an array of size N: token_data[j] contains j's request sequence number granted most recently. When node i leaves its critical section, it updates token_data[i] with its sequence number to indicate that its current request has been granted. Next, it searches a logical circular list for the first node whose last request has not been granted yet and sends it the token. If no node has sent a request, node i retains the idle token. A node holding an idle token can enter its critical section without the need to send a request message.

When a node k receives a request message from a node j, it updates its request_data[j] with the most current request sequence number received from j that k knows of. This update takes care of out-of-order request messages and old requests already granted. If node k is holding the token and is not in its critical section, then k sends the token to node j.

The algorithm requires N messages exchanges for each critical section entry, or 0 messages if the requesting node is holding an idle token. The token moves in a logical circular ring of nodes ordered by their unique identification number. Therefore, a first-come-first-served service is not guaranteed. The algorithm assumes nodes do not fail and the existence of a fully reliable communications network. Transfer delays are finite, but unpredictable, and message-order preservation is not required.

Suzuki-Kazami algorithm. Suzuki and Kazami developed an algorithm [27] similar to the Ricart and Agrawala algorithm [21] presented above. Each node maintains an array of size N to store the sequence number of the most recent token invocation from all nodes in the system. When node i wishes to enter its critical section and does not hold the token, it increments its request counter RN[i] and sends a request message of the form REQUEST(i, RN[i]), to all other N-1 nodes. It then waits for the arrival of the token message. If it has the token or the token arrives, it proceeds to enter its critical section.

The token message has the form PRIVILEGE(Q, LN) where Q is a queue of requesting nodes and LN is an array of size N which contains the sequence number of the request granted most recently to each node.

After exiting its critical section, node i updates LN[i] with its current request granted RN[i]. Next, it enqueues all nodes not in Q from whom it has received a request which has not been granted yet. Nodes are inserted at the rear of the queue in an ascending node number order. If there exists a process in Q, the token is sent to the process at the front of it. If Q is empty, node i retains the idle token. A node holding an idle token can enter its critical section without the need to send a request message.

When a request from node j arrives, RN[j] is updated with the most current request number ever received from it in order to discard out-dated information. This would take care of old requests already granted and out-of-order request messages. If node i is holding the token, not requesting its critical section and the most current request from j has not been served yet, then the token is sent to node j.

The algorithm requires N messages exchanges for each mutual exclusion invocation or 0 messages if the requesting node is holding an idle token. It is assumed that transfer delays are finite, but unpredictable. Message-order preservation is not required.

The algorithm has only two procedures (see appendix A), P1 and P2; only P2 is executed indivisibly. In procedure P1, if Q is empty and a request message arrives after line 20 and exactly before line 21 and procedure P2 is called, it could happen that this request would starve for some time (at least, until another node originates a request).

Sequence numbers are unbounded, but the algorithm can be modified in order to bound them. The modified algorithm requires L*N + (N-1) message exchages for L mutual exclusion entries, where L is a fixed integer greater than or equal to 2.

Mizuno-Neilsen-Rao. In their algorithm [12], Mizuno et al. incorporate quorum agreements to form groups of nodes in the system and adopt the mechanism to move the token used in Suzuki and Kazami's algorithm [27].

A quorum agreement (Q, Q^{-1}), is a pair of sets which contain subsets of all nodes in the system and where every member of Q intersects with every member of Q^{-1}. A quorum agreement data structure is used to reduce the number of messages exchanged to perform an entry to the critical section.

Each node in the system is a member of at least one set in Q and a member of many sets in Q^{-1}. For a node i, quorums from Q are called the request set R_i, and quorums from Q^{-1} are called the acquired set A_i. The condition, R_i intersection A_i <> 0 holds. This means that R_i and A_i are related by at least one member common to both sets. Messages are exchanged through this relation-link, or intermediate node, between both sets.

Basically, the mechanism for locating the token is as follows. When node i wants to enter its critical section and does not hold the token, it sends request messages, which contain its new sequence number RN, only to all other members of R_i. The best case is when another member of R_i is holding the token, then the process of locating the token ends. Suppose that this is not the case; then each of the other nodes j in R_i are also members of at least one set A_j. This means that at this point one member in all acquired sets in the system know of the request from node i and have updated their RN array.

One node in a set A_j is holding the token (unless the token is traveling in the system, but this condition will be met in a finite amount of time) and this node j is not a member of R_i. Node j has sent an ACQUIRED(j, LN_j) message to all other nodes in A_j. This message indicates that j has received the token. Upon receiving the acquired message from j, all other nodes k in A_j check if they have requests that have not been granted yet. One of these k nodes is the relation-link (or intermediate node) to R_i, so it confirms it has an outstanding request from i and sends a REQUEST(RN_k) to node j. The request contains the whole RN array because there could exist many pending requests.

The token holder now knows of the request from node i. It updates its RN array and the LN array in the token, and enqueues i's request in the token's queue. Requests are enqueued in an ascending node number order. After node j leaves the critical section, the token is sent to the next node in the token's queue. If the queue is empty, node j holds the idle token.

The performance of the algorithm, in terms of the number of messages exchanged for an entry to the critical section, depends upon the underlying quorum agreement. If it is based on the binary tree protocol, the upper bound is in the order of magnitude of log N. The upper bound using quorum agreements based on finite projective planes is approximately 3 * SQRT(N). A modified grid-set protocol may be used to generate the quorum agreements and the upper bound in this case would be (2 * SQRT(2) * SQRT(N) - 2). The algorithm from Suzuki and Kazami is a special case of this algorithm if all nodes except i are in R_i and i is the only node in A_i.

Neilsen-Mizuno algorithm. Neilsen and Mizuno impose a static logical structure on the communications network. Nodes are arranged in a directed acyclic graph and communicate only with their neighbors [14]. A node or a token does not need to maintain a queue of pending requests. This queue is implicity maintained by the state of each node in the system.

Each node holds two integer variables, LAST and NEXT, and a boolean variable HOLDING. LAST indicates the last neighbor node from which a request was received, either on its behalf or on one of its neighbors' behalf. If this node is the origin of the request, then LAST = 0. The variable NEXT indicates the node which will be granted the token after this node makes use of it (the next node in the implicit queue of pending requests in the system).

When a node j wants to enter its critical section (CS) and does not hold the token, it sends a request of the form REQUEST(my_id, origin) to LAST. In this case my_id = origin = j. Node j sets $LAST_j$:=0, becoming a sink node, and waits for the token to arrive. If node j receives a request from node k at this point, it sets $NEXT_j$:=origin and $LAST_j$:=k. Node j has become an intermediate node in the path along which a request message travels.

If an intermediate node j receives a REQUEST(y, origin) from a neighbor node y, it will forward a REQUEST(j, origin) to $LAST_j$, on behalf of its neighbor. It will then add y to the path by setting $LAST_j$:=y. The request message will travel along the path until it arrives to the sink node. The request from the origin node will be "enqueued". The origin node will become the sink node and the last in the path ($LAST_{origin}$:= 0).

When j receives the token, it can enter its CS. After exiting it, if there is a pending request ($NEXT_j > 0$), the token is sent to $NEXT_j$ and $NEXT_j$:=0. If there were no pending requests, node j keeps the idle token. A node holding an idle token does not need to send a request to enter its critical section again.

The queue of pending requests can be deduced by following the state of NEXT in each node, starting at the node holding the token. The algorithm does not need sequence numbers and requires very simple data structures. Messages are very small, reducing the overhead in the communications network, which it is assumed to be reliable. The total number of messages exchanged per critical section entry depends on the topology of the logical structure, but has an upper bound equal to (D+1), where D is the length of the longest path in the network; the diameter of the network. In a star topology the upper bound is 3. For a linear topology, the upper bound is N.

Helary-Plouzeau-Raynal algorithm. In their algorithm [6], a process wanting to enter its critical section and not possessing the token sends a request only to its neighbors and waits for the token. If it holds the token or the token arrives, it can proceed to execute the critical section. After completing the execution of its critical section, the node calls a procedure for transmiting the token.

A request message contains the identification of the node originating the request, the request time based on a logical clock following Lamport's rules [8], the identification of the node forwarding the request, and a set of nodes for which a request has already been sent.

Requests are propagated in the network based on a knowledge-transfer control method. A node receiving a request knows who originated it, which neighbor forwarded it, and finds out to which of its own neighbors the request has not been sent yet. Next, it propagates the request only to those neighbors. A return path to the requesting node is constructed with the identification of the nodes that have propagated the request. When the node holding the token has completed the execution of its critical section it can send the token directly to the requesting node following the return path.

Upon receiving a request from its neighbor node j, node i updates out-dated information that it maintains for j. This would take care of out-of-order messages from node j and of deleting requests already granted to j. The request is added to i's set of known pending requests. Node i synchronizes its logical clock, finds out the set of neighbors to which it will propagate the request, adds itself to the return path, and propagates the request. If node i is holding an idle token, then it calls a procedure for transmiting the token.

In the procedure for transmiting the token, node i finds the oldest request from its own set of pending ones, updates the time of that request in the token's array with its logical clock, and sends the token through the return path.

When a node k receives the token message and the token final destination node is not itself, it checks the return path and forwards the token to its neighbor following the return path. If the token is addressed to k, then it can proceed to enter its critical section.

The algorithm assumes the existence of a reliable communications network and finite but unpredictable transfer delays. The algorithm does not require message-order preservation. Nodes need not to have any prior knowledge of the network topology. The only knowledge owned by a node is the name of its neighbors. The number of messages sent to locate the token is reduced by using a flooding broadcast technique and a knowledge-transfer control technique. The number of messages required per critical section entry depends on the actual network topology. Whatever topology is considered, if the requesting node owns the token there is no need to send any request messages. For a linear topology, the total number bounds are N and 2(N-1). The total number of messages varies from N to 2N in a ring topology. In a complete network the total message number is N. Requests are fully ordered by the use of logical clocks and are granted in a first-come-first-served manner.

Raymond's algorithm. In Raymond's algorithm [16] a static logical tree structure is used. Nodes are arranged in an unrooted tree structure and communicate only with their neighbors. Each node holds information pertaining to its own neighbors only. The location of the token is relative to those neighbors. A node, say A, holding the token becomes the privileged node and its neighbors, say B, C, and D, know A holds the token. Neighbors of D, say E and F, do not know that A is holding the token. They only know that D represents the relative location of the token.

If node E wants to enter its critical section, it needs to send a request to D after placing itself in its own request queue. Node D adds E's request in its request queue and sends a request to A on its own behalf. Suppose now that node D receives a request from its neighbor node F, it enqueues F's request, but it does not send another request to A since it has already done so. Furthermore, node D now wants to enter its critical section, so it enqueues its own request in its request queue, but it does not send a request to A since it has already done it.

Node A receives only one request from node D. It enqueues D's request on its request queue. After completing the execution of its critical section, A sends the token to the first node in its queue, say D, and learns that node D will now be the relative location of the token. If A discovers that there are still nodes enqueued in its request queue, or if A wants to enter its critical section again, it sends a request to node D.

When node D receives the token, it finds that node E's request is the oldest one in its request queue. It sends the token to E, and learns that the relative location of the token is now node E. It also discovers that its request queue is not empty (F's request and D's own request are enqueued) and sends a request message to E (the relative location of the token).

Upon receiving the token, node E finds its own request as the oldest in the queue. When it receives the request from node D, it adds it to its queue of pending requests. When it releases the critical section, finds the oldest request in its queue, D's request in this case; sends the token to D and learns that the relative location of the token is now node D.

Node D receives the token and finds that the oldest request is from node F. It sends the token to F and learns that the relative location of the token is node F. Since its queue is not empty, it sends a request to F. And the process continues.

If for node D the relative location of the token is node F, this could be seen as a directed edge D->F. As the token travels along the unrooted tree, the direction of the edges changes. At one point, the edges in the system would represent a directed acyclic graph, and a single directed path could be deduced from each node to the token holder.

The algorithm assumes a reliable communications network and finite but unpredictable transfer delays. It does not require message order preservation. Messages do not need sequence numbers to enforce the order of events and requests are granted in a first-come-first-served manner. The total number of messages exchanged for an entry to a critical section is typically in the order of magnitude of log N. A piggyback strategy could be used to reduce the number of messages. A recovery procedure from the failure of a node is presented and it could be incorporated to the algorithm. The system can recover from a failed node, providing that not all of its neighbors also fail. Nevertheless, it cannot recover from the failure of the node holding the token and all of its neighbors.

Singhal's algorithm. In his algorithm [24], each node maintains information about the state of the system. This information is disseminated implicitly within request and token messages. Whenever a node wants to enter its critical section, it uses a heuristic to deduce from its available state information what nodes are probably holding the token and sends a request only to those, rather than to all other nodes in the system. The heuristic is used in order to minimize the number of messages sent to locate the token.

Each node uses a local sequence number counter to keep track of its last request invocation. Two vectors are used by each node to store the information about the state of the system. The state vector stores the latest known states of all sites. The possible states are: requesting, not requesting, executing its crital section, and holding an idle token. The other vector indicates the latest known request invocation for each site. The token message also contains two vectors, one for storing the state of each site and the other one for storing sequence numbers for each node.

When a process i wants to enter its critical section and is not holding the token, it increments its sequence number counter and sends a request message of the form request(i, SN[i]), where SN[i] indicates its latest request. Node i uses the state information it has about the system and sends a request to only those nodes which are in the "requesting" state. One of this nodes is likely to know the location of the token, or the token will be granted to it in a finite amount of time. After sending its request message, node i waits for the arrival of the token. If node i is holding the token or the token arrives, it can proceed to execute its critical section.

When node j receives a request from i, it checks the request sequence number against its sequence number vector to discard out-dated requests. If the request is a new one from node i, then it verifies the information in its state vector to update i's state. If node j is requesting and the state information for i, before the update, was not "requesting," then j sends a request message to i because it became one of the nodes that probably know the location of the token. If node j is holding an idle token, then it sends the token to i.

When a node completes the execution of its critical section, it compares the information of its own state vectors against the vectors in the token, and updates all vectors with the most current information about the state of each node. The update rule is such that if the vectors in the token hold out-dated information, these are updated with the information contained in the node's state vectors, and vice versa. After the state information has been updated, the node uses arbitration rules to determine which requesting node should get the token. The token will be granted to the nearest requesting node with the lowest sequence number. Nodes are ordered in an unidirectional logical ring by their unique number identification. This rule guarantees that nodes which have executed their critical section least frequently will get the token, and prevents a node from obtaining the token twice while some other node is waiting for it.

The algorithm assumes that message propagation delay is finite, but unpredictable. The number of messages exchanged for an entry to a critical section is (N+1)/2 in case of light traffic, and N in the case of heavy traffic. In light traffic a node holding an idle token does not need to send a request message if it wants to enter its critical section. Singhal discusses the impact of node and communication link failures, and presents recovery procedures that could be incorporated into the algorithm.

Naimi-Trehel algorithm. In their algorithm [13] an underlying dynamic logical structure is used on the communications network. Requesting processes are logically arranged, by their requests, as a rooted tree. As a request from node i travels along the path from node i to the root node, node i becomes the new parent of each node on the path, except for itself. Thus, node i becomes the new root node of the tree.

Neither the nodes nor the token needs to maintain a queue of pending requests. This queue is implicity maintained by the state of each node in the system. Each node keeps two integer variables, LAST and NEXT. The former indicates the last node from which a request was received and the neighbor node in the path to the root to whom this node will send a request message the next time it wants to enter its critical section. NEXT indicates the node to whom the token will be granted after this node leaves its critical section.

When a node i wants to enter its critical section (CS) and $LAST_i$ <> i (node i is not holding the token), it sends a request to node $LAST_i$, it sets $LAST_i$:=i, and waits for the token to arrive. If it has the token or the token arrives, it enters its CS directly.

When the request from node i arrives at a non-privileged node j in the path to the token holder node (the privileged node), node j forwards the request from i to node $LAST_j$. Node j sets $LAST_j$:=i. When the request from node i arrives at the privileged node, say node k, and k is the root node of the tree ($LAST_k$:=k) and it is in its critical section, then node k sets $NEXT_k$:=i and $LAST_k$:=i. If k is the root node and is holding an idle token, then it sends the token to node i and sets $LAST_k$:=i. In the case that node k is not the root node, it forwards the request from node i to node $LAST_k$. The latter happens when node k received a request from another node prior to the request from node i and thus it became part of the path form node i to the root node.

When a privileged node k holding the token leaves the critical section, it sends the token to node $NEXT_k$ and sets $NEXT_k$:=0. If there were no pending requests ($NEXT_k$:=0), the node keeps the idle token.

The queue of pending requests can be deduced by following the path of the NEXT state in each node. The head of the queue is the privileged node. The token moves sequentially traversing this path in the tree. The algorithm does not require sequence numbers for ordering the events. Messages are very small since very simple variables are transmitted. This reduces the overhead in the network.

The average number of messages exchanged for an entry to the critical section is in the order of log N. A node holding an idle token does not need to send a request to enter its critical section again.

The algorithm assumes the existence of a fully reliable communications network. Transmission delays are finite and messages need not be delivered in the order they are sent. The algorithm incorporates a mechanism for the detection of and recovery of the system from node failures. The mechanism is based on the use of two delays, one indicates a presumption of failure (Twait) and the other permits the broadcast of a question and the reception of the answers (Telec). Each node is in one of 5 possible states: waiting, consulting, query, candidate or observer.

When a node i sends a request message, it enters a waiting state. If it does not receive the token within Twait, there is a presumption of failure. If a failure has occured, node i consults if any node k has record of its request and $NEXT_k$=i. After Telec has timeout, it queries the other nodes to detect if the token is present in the system. When this Telec expires, node i becomes a candidate to regenerate the token, broadcasts an election message and activates another Telec. When several other are candidates in the same Telec interval, the one with the smallest node number will be elected. All other nodes in the system become observers and wait for a "candidate_elected" message from the elected node to resume operations. The elected node possesses the new token and becomes the root of the reorganized tree. All nodes set their LAST to be the elected node number and NEXT is set to 0.

During the interval delays at the consulting and query phases, the token or an answer from another node may arrive, and the inquiring node goes back to the waiting state. Queries for the presence of the token in the system are recorded by the nodes for the case in which the token is travelling. The algorithm assumes that when a node fails, it continues to fail during the election process. It does not consider what happens when a failed node recovers and is incorporated to the system. This node may be holding an old token. Failures in the communications network are not considered in the algorithm.

Mishra-Srimani. Mishra and Srimani extended the algorithm of Suzuki and Kazami [27] and incorporated a fault tolerance mechanism to recover the system from a single node failure [11]. A lost token can be regenerated and the state of each site can be reconstructed. The algorithm ensures the elimination of duplicated tokens in the system.

Fault tolerance is implemented using a time-out mechanism. When process j wants to enter its critical section it sets a time-out after sending its request to all other nodes. The algorithm behaves very much the same as the original [27] under the condition that no node fails and the requesting process receives the token within the time-out.

When the token does not arrive within the time-out, process j checks if another node has started the regeneration of the token. If that is the case, j waits for a message indicating it can reset its time-out and wait for the token again. When no other node has started the regeneration procedure, j commences it. It sends a PRIVILEGE_CHECK(j) message to all nodes and waits for their response. If a response indicates that the token is not lost, then j sends a message to all other nodes indicating that it was a false alarm and waits again for another time-out for the token to arrive.

If a node k receives a PRIVILEGE_CHECK(j) message, it either sends a REP_CRITICAL(k) message to j if k has the token, or sends a NO_CRIT(k) message to node j if it has not started a token regeneration procedure. If node k already started to regenerate the token when a PRIVILEGE_CHECK(j) message arrives, if j's node number is less than k's, then node k stops the generation procedure. Otherwise, k ignores the message.

A process k that has started the regeneration of the token will know that the token is lost and that no other node is trying to regenerate it, when it receives no REP_CRITICAL messages and N-2 NO_CRIT messages (at most one node could have failed and did not respond). It then sends CREATE_PRIVILEGE(k) messages to all correct nodes and gets an UPDATE message from each one of them. This message contains enough information to construct the state of the token's queue of pending requests. Node k sends messages to all nodes indicating the recovery of the system and starts executing its critical section.

The algorithm assumes that transfer delays are finite but unpredictable and that messages might not arrive in the order they are sent. The total number of messages exchanged for a critical section entry is L*N+(N-1) under the absence of failures.

A second algorithm is presented in which a central coordination control for mutual exclusion is moved among the nodes in the system. At any given time, there exists only one central coordinator in the system.

Nishio-Li-Manning algorithm. Their algorithm [15] is an extension of the Suzuki and Kazami algorithm [27], but the movement of the token is different. The token is granted to the nearest node, in a logical circular list, whose last request has not been granted yet, as in [21].

Fault-tolerance, based on time-out values, is incorporated in their algorithm. It is assumed that each node in the system consists of a processor and a communication controller. The algorithm can recover from processor failures, controller failures and communication link failures. A lost token can be regenerated and duplicated tokens are eliminated from the system.

The controller of each node k manages message exchanges with other nodes, controls processor k's right to enter its critical section, and is able to regenerate a new token if need be. Therefore, state arrays are stored in the controller's memory. Processor and controller interact, exchanging information. Based on that information, the controller can decide to regenerate a new token or eliminate a duplicated one.

Request messages include the identification of each node to which it is being sent. A field indicating the age of the token is added to the token message. Each node keeps the age of the most current unique token generated in the system. When a process i has sent a request to all other N-1 nodes in the system, it sets a time-out for the token to arrive. If the token arrives within this time and its age is not older or equal to the age value at site i, then it is a duplicated token and is discarded. The process re-sends its request to all other nodes and the same procedure is repeated. If the age of the token is at least equal to the age value at site i, the age value of i is updated and it can proceed to enter into its critical section.

If the token did not arrive within the time-out, the token regeneration procedure commences. A TOKEN_MISSING message is sent to all other nodes. If one of them does not respond, the procedure starts again. If all other N-1 nodes responded with an ACK message, a new token is regenerated and an incremented age is given to it. The process now can proceed to the execution of its critical section. In the case that a single node replied with a NACK message, the procedure re-starts at the point where node i sends again its request to all other nodes.

The TOKEN-MISSING message includes an incremented proposed_age field from node i. ACK and NACK messages also include an age value that corresponds to the proposed_age in the TOKEN-MISSING message, in the case of ACK messages, or is different, in the case of NACK messages. NACK messages help in resolving conflicts among processes that started the regeneration of a new token, or to indicate that the token has not been lost. The node with the highest proposed age is candidate to regenerate the new token. A node k is allowed to regenerate a new token only if it has received ACK messages, corresponding to its TOKEN_MISSING enquiry, from all other N-1 nodes. ACK messages contain j's most current request granted; therefore, the array of most recently requests granted can be constructed in the new token.

The information stored at the controller is assumed not to be lost in case the controller fails. Since this is difficult to keep and might not be recoverable in a catastrophic controller failure, a mechanism that uses information propagation is described for the recovery from these failures.

The algorithm assumes finite transfer delays and does not require message-order preservation. The number of messages exchanged for a critical section entry is N. The resiliency mechanism presented can be easily modified to include the recovery from a node insertion, or removal. Channel insertion/removal can be treated in the same way as their failure/recovery.

4.1 Recapitulation of the performance of token-based algorithms.

Table 1 below shows the performance of the algorithms described above. The column in the center indicates the total number of messages required for an entry to the critical section to take effect.

ALGORITHM	TOTAL MESSAGES	OBSERVATIONS
Ricart-Agrawala [21]	N	
Suzuki-Kazami [27]	N	L * N + (N - 1) for bounded sequence numbers.
Mizuno-Neilsen-Rao [12]		
binary tree protocol	avg log N	Uses quorum agreements.
finite proj. planes	3 * SQRT(N)	
Neilsen-Mizuno [14]	D + 1	Uses a Direct Acyclic Graph.
linear topology	N	
star topology	3	
Helary-Plozeau-Raynal [6]		Discusses the effect of failures and presents suggestions for the recovery. Uses a knowledge-transfer control technique.
tree toplogy	N to (N-1+D)	
linear topology	N to 2(N-1)	
ring topology	N to 2N	
complete topology	N	
Raymond [16]	2*D avg log N	Discusses the effect of failures and gives suggestions for the recovery. Uses a static logical tree structure.
Singhal [24]	N	Discusses the effect of failures and gives suggestions for the recovery. Uses state information and heuristics.
Naimi-Trehel [13]	avg log N	Considers node failures, and the token regeneration. The state is not reconstructed. Uses a dynamic logical structure.
Mishra-Srimani [11]	L*N+(N-1)	Considers node failures, the regeneration of the token and the elimination of duplicated tokens. The system state is reconstructed.
Nishio [15]	N	Considers processor, communications controler and communication link failures. The regeneration of a token and the elimination of duplicated tokens. The system state is reconstructed.

N = the number of nodes in the system.
D = the diameter of the network (the longest path).
L = an integer value >2, used to bound sequence numbers.
avg = average.

Table 1. Performance of token-based algorithms

A drawback of the algorithms that use sequence numbers to order the events in the system, is that the sequence numbers are not bounded. Suzuki and Kazami [27] proposed a way to bound them, but this increments the total number of messages in their algorithm to L*N + (N-1) for L mutual exclusion invocations by a single node.

The algorithms that incorporate fault-tolerance aspects base their detection mechanisms on the use of timeouts. The resilient algorithms shown in Table 1 exhibit the performance indicated, under no failures conditions and the reception of messages within the required timeouts. It is difficult to analyze their performance under failure conditions because of the probability that a failure will occur, the inherent delays in the transmission of messages and the appropriateness of the size of the timeouts chosen.

In the following section, a description of various permission-based algorithms is given.

5.- DESCRIPTION OF PERMISSION-BASED ALGORITHMS.

In these algorithms, requesting processes wait to obtain permission from a set of processes in the system. Once a process obtains permission from a sufficient number of members in a set, it is allowed to enter the critical section (CS). Only one process at a time can get enough rights to execute its CS. Each node grants its permission to only one node at a time. This ensures the condition for mutual exclusion.

Two inter-related aspects are considered in these algorithms to reduce the number of messages exchanged for an entry to the CS to take effect. The number of "enough" rights that should be collected, and which nodes should grant those rights. Some algorithms require that a node should obtain permission from all nodes in the system. In other algorithms, nodes are divided into groups that intersect with each other in a non-null pairwise manner. Any possible group must have one node in common with any other group to ensure mutual exclusion. A node needs to obtain permission only from all the other members in its group.

Thomas [28] used a voting technique based on a majority consensus algorithm that requires a requesting node to obtain permission from only a majority (N+1)/2 of nodes. The intersection of any two majorities has at least one node in common. This means that for any two requests that are received, at least one node grants its permission to one of them, and defers it to the other. Agrawal and El Abbadi [1] called this consensus the majority quorum.

The concept of obtaining permission from a group of nodes, which are not necessarily a majority, was formalized by Gifford [5]. He introduced the notion of quorums, which are nonempty sets of nodes. Garcia-Molina and Barbara [4] introduced the notion of coteries. A coterie is a nonempty set of quorums in which any two quorums must have at least one common node (intersection property), and no quorum is a subset of any other one (minimality property).

Agrawal and El Abbadi [1] discuss different protocols to construct quorums. The unstructured quorum protocol can be used to derive majority quorums. The grid protocol is used to form a square grid of quorums as the ones described in Maekawa's algorithm [9]. In the tree protocol, nodes are logically organized to form a complete binary tree, and tree quorums can be derived from this structure.

All the algorithms presented in this section claim to satisfy the mutual exclusion requirement, be deadlock free and starvation free. The first four algorithms [8, 20, 3, 18] described below, require a node to obtain permission from all other nodes. The next three algorithms [9, 22, 2] impose a logical structure on the system to group nodes into sets, and require a node to obtain permission only from the other members in its set. In the last algorithm [25], a dynamic information structure is used to form quorums. Algorithms in each group are presented in chronological order.

Lamport's algorithm. In [8] Lamport describes a mechanism based on logical clocks for the total ordering of requests in the system. A timestamp is associated to each request and the order among them is guaranteed by the following two rules. (a) Each process P_i increments its logical clock C_i between any two succesive events. (b) Each message m from process P_i contains a timestamp $T_m=C_i(a)$, where a is the event of sending the message. When process P_j receives a message m, it sets C_j greater than or equal to its present value and greater than T_m. This ensures that if a is the sending of a message by process P_i and b is the receipt of that message by process P_j, then $C_i(a) < C_j(b)$.

Each process maintains its own request queue. A requesting process sends a timestamped request to all other N-1 processes and can enter its critical section when permission from all other processes is received, and its request is next in its ordered request queue. This ensures the mutual exclusion condition. A process that receives a request message sends back a timestamped reply message to the requesting process. When a process releases its critical section it sends a release message to every other process to notify that its request has been granted. Each process in the system receiving a release message updates its queue of pending requests and checks if it can proceed to enter its critical section.

When a process i wants to enter its critical section it sends a request message of the form REQUEST(T_m, i) to all other N-1 nodes and puts that message in its request queue. When process j receives the request from node i, it places it in its request queue, updates its C_j and sends a timestamped reply message to i. When process i receives reply messages from all N-1 other processes with a timestamp greater than the timestamp in its request message, and its request is next in its queue, then it can enter its critical section. When i releases its critical section, it deletes any request (T_m, i) from its request queue and sends a timestamped release message to all other processes. When process j receives a release message from i, it deletes any request message (T_m, i) from its request queue.

Requests are served in a first-come-first-served manner. The algorithm assumes the communications network is fully reliable. Message-order preservation is required and the total number of messages exchanged for an entry to a critical section is 3 * (N-1).

Ricart-Agrawala algorithm. In their algorithm [20] a node has to receive permission from all other N-1 nodes in the system to enter its critical section (CS). A node wishing to execute its CS sends a request message to every other node and waits for their permission to arrive. When a node receives a request, it sends its permission to the requesting node if either it is not requesting itself or it is requesting itself, but the other node's request precedes its own. The sending of its permission is deferred otherwise. Requests are ordered by using sequence numbers in the system.

When node i is going to send a request message to all other nodes, it increments the highest sequence number it has knowledge of and includes it in its request message. After request messages are transmitted, node i waits for the arrival of N-1 reply messages to enter its CS. When process j receives the request from node i, it sends back its permission if it is not requesting the CS itself, and updates its highest sequence number value. If node j is requesting to enter its CS and the sequence number in the request from i is lower than the one in its own request, then j sends its permission to i. If j is requesting and its request sequence number is lower, then it defers a response to i and keeps record of it in its Reply_Deferred[i] array. Ties are solved by granting the permission to the node with the lowest node number. When a node releases its CS, it sends its permission to all nodes for which a reply to their requests was deferred.

The algorithm requires 2 * (N-1) messages for an entry to the CS to take effect. It assumes the existence of an error-free underlying communications network. Message transfer delays are finite, but unpredictable, and message-order preservation is not required. Requests are serviced in a first-come-first-served manner.

Carvalho-Roucairol algorithm. Their algorithm [3] is a variation of the Ricart and Agrawala algorithm [20]. Once a node i has received permission from a node, it can keep it for future use until a request is received from that node. The next time node i wants to enter its critical section (CS), it will send request messages only to those other nodes whose permission is not already kept by i. This indefinite permission reduces the number of messages substantially when only a few nodes are frequently invoking mutual exclusion.

When node k grants its permission to node i, its authorization remains valid until it wishes to enter its CS again and sends a request message to i. When node i wants to enter its CS, it increments the highest sequence number it has knowledge of and sends a request message of the form REQUEST(my_sequence_number, i) to only the other nodes which need to be consulted. Then, node i waits for those reply messages to arrive to execute its CS.

When node j receives the request from i, it updates its highest sequence number known and sends its permission to node i if either it is not in its CS or requesting it, or is requesting the CS, but the request sequence number from i is lower than its own. Node j records in its array[i] that node's i permission is no longer valid. If node j is either in its CS, or is requesting it and its request is lower than that from i, then the sending of the permission is deferred. There is one special case in which node j is requesting the CS, the permission from node i is still valid, but the new request from i contains a lower sequence number than its request. Node j records in its array[i] that node's i permission is no longer valid, sends its permission to i, and sends a request message to it.

When a node releases the CS, it sends its permission to all nodes for which a reply to their requests was deferred, and updates its array of valid permission. The last node entering its CS can reenter it if no other node requests it.

The algorithm requires from 0 to 2 * (N-1) messages for an entry to the CS to take effect. It assumes there is a reliable underlying communications network. Message transfer delays are finite, but unpredictable, and message-order preservation is not required.

Raynal's algorithm. Raynal introduces the use of prime numbers for describing the global state of the system. In his algorithm [18], prime numbers are used to order the events in the system. His aim is not to obtain an efficient algorithm from a number of messages point of view, but to show that prime numbers and their properties can be a useful tool in the design of distributed algorithms.

Each node i is endowed with a different attribute A_i. These attributes are natural integers, different from 1, and pairwise prime. Each node maintains a variable X_i initialized to A_i except for one X_k initialized to 1. They also know that Q is the total product of all prime numbers A_i. When a node i wants to enter its critical section (CS), it sends a request message of the form REQUEST(i) to all other N-1 nodes and waits to receive a reply message from all other nodes. Reply messages have the form REPLY(j, X_j). When all replies have arrived, node i computes T, the total product of all values X_j received. If T equals Q/A_i, then node i can proceed to enter its CS. Otherwise, it waits for a time and then re-sends request messages to all other nodes. When node i releases the CS, it updates its variable X_i to (X_i * A_i/A_j), where j = (i + 1) mod N. The permission to enter the CS rotates around a logical ring of nodes.

A node updates its variable X_i only after it makes use of the CS. The effect of this update is that i loses the permission and the next node in the circular ring obtains the permission. The algorithm can suffer from deadlock if the next node in the logical ring does not want to enter the CS; it never updates its variable X_j. Therefore, it never gives the privilege to another node.

The algorithm assumes the existence of a reliable communications network. Transmission delays are unpredictable, but finite, and message order preservation is not required. The number of messages exchanged has an upper bound of $2(N-1)^2$.

We propose a modification to the algorithm to avoid deadlock situations and reduce the number of messages exchanged. At initialization, each node knows the value of all X_i, and the node endowed with the value $X_i=1$ has the permission to enter the CS, the same as in the original algorithm. Nodes do not need to send request messages. They know the values of each X_i, and are waiting for the privileged node to release the CS and update its X_i. The permission from all other nodes is implicit, except for the permission from the privileged node. Each node maintains a local variable to indicate whether it is requesting the CS. When a node i is not requesting the CS, and is the next in the ring to obtain the permission, it should behave as if it were releasing the CS. It updates its variable X_i to give the permission to another node. This avoids deadlocks. After the update, it sends a message of the form INFORM(i, X_i) to all other N-1 nodes. A node k receiving a message, computes T to check if it equals (Q/A_k). If it does and the node is requesting, then it can enter the CS. If it is not requesting, but it is next, then it behaves as if it were releasing the CS. The total number of messages exchanged for a CS entry has an upper bound of $(N-1)^2$ in the worst case, when the next node wishing to enter the CS is the farthest node in the circular ring from the node currently owning the permission.

Maekawa's algorithm. Maekawa imposes a logical structure on the network. In his algorithm [9], a set of nodes is associated with each node, and this set has a nonempty intersection with every set associated with each other node. A node i must obtain permission from all other nodes in its home set S_i before it can enter its critical section (CS). Since the set intersects with every other set of other nodes, mutual exclusion is guaranteed. Each other node k in S_i is associated with another set S_k for every set in the system. Node k acts as an arbitrator for requests received from i and also from members of its home set S_k. Hence, when each node k in S_i gives its permission to node i, then i has collected enough rights and can enter its CS. Each arbitrator node grants only one permission to one single node. No other node in all sets S_k in the system will have enough rights and thus, it will not be allowed to enter its CS.

Requests are ordered by the use of sequence numbers in the system. Conflicts are solved by requiring a node to yield if its request sequence number is larger than the sequence number of any other request. Ties are solved by favoring the node with the lowest node number.

When node i wants to enter its CS, it increments the highest sequence number known and sends a request message to every other member of S_i. Node i itself pretends to have received a request, since it is also an arbitrator node.

When receiving a request from i, each node k member of S_i checks if it has already granted its permission. If it has not, it sends its permission to i. Otherwise, it enqueues i's request in its ordered waiting queue. If any request in the queue has a lower sequence number than that of i's, a failed message is sent to node i. If the request from node i has the lowest sequence number, then an inquire message is sent to the node to which permission was granted, to check whether this node has obtained permission from all other members in its home set. If this node has succeeded, it is free to access the CS. If it has not succeeded, it cedes its permission to free its member node k to service a request with a lower sequence number. Node k enqueues the unsuccessful request. It removes from its queue the request from node i, and sends it its permission.

When node i obtains permission from all other members of S_i, it can enter its CS. Upon releasing the CS, a release message is sent to all members of S_i. When a release message arrives at node k, it grants its permission to the next request in its waiting queue. No action is taken if the queue is empty.

The construction of the sets in the system significantly impacts the number of messages required to effect an entry to the CS. The rule for constructing these sets is based on the structure of finite projective planes of N points. The size of these sets is square root of N. Hence, a node communicates with only SQRT(N) nodes to obtain permission. The total number of messages exchanged in a mutual exclusion invocation is C * SQRT(N), where C is a constant between 3 and 5.

The algorithm assumes the existence of an error-free underlying communications network. Message transfer delays are finite, but unpredictable, and message-order preservation is required.

Sanders' algorithm. Sanders introduces the concept of an information structure as a unifying principle behind several algorithms that have been proposed [9, 20, 3]. Sanders develops a generalized mutual exclusion algorithm [22] based on this approach. The information structure describes which nodes maintain information about the state of other nodes, and the set of nodes from which each node should request information or permission before it enters its critical section (CS). Three sets of nodes are associated to each node i. The informing set I_i, the request set R_i, and the status set S_i. When constructing the sets, two conditions must be satisfied: a) I_i is a subset of R_i, and b) for all i and j either there is a non-null intersection between I_i and

I_j, or both j belongs to R_i and i belongs to R_j. The status sets are determined by the informing sets where j is a member of S_i if i is a member of I_j.

When node i wants to enter its CS, it must obtain permission from all other members in R_i. Every other member node k of R_i acts as an arbitrator to grant its permission to requesting nodes from different sets for which k is a member. When k sends its permission to a requesting node which is a member of its S_k, it sets its variable $CSSTAT_k$ to indicate that this node is in the CS. When the node is not a member of its S_k, then $CSSTAT_k$ indicates the CS is free. Node k arbitrates incoming requests only when it has already granted its permission to a node in its S_k to enter the CS. Otherwise, according to its information in $CSSTAT_k$, the CS is free and k sends its permission right away to the next request in its queue of pending requests.

When arbitrating, node k solves conflicts by following a mechanism based on Lamport's timestamped request messages [8]. When it receives the request from node i, the request is placed in k's requesting queue. If the CS is not free and i's timestamp is larger than that of the node indicated in $CSSTAT_k$, then a fail message is sent to i. If i's timestamp is lower, then an inquire message is sent to the node indicated in $CSSTAT_k$ to find out whether it has been successful in obtaining permission from all other members in its request set. A fail message is sent to a node with a larger timestamp that has not yet been sent one. If the CS is free, then the permission is sent to the next request in the queue and it is removed from it. If the privileged node is a member of S_k, then $CSSTAT_k$ is set to indicate that the privileged node is in the CS.

When node i releases the CS, it sends a release message to all nodes in I_i. Each member node in I_i sets its CSSTAT to indicate the CS is free. They send their permission to the next requesting node in the queue and the request is removed from the queue. If the privileged node is a member of its status set, then CSSTAT is set to indicate which node is in the CS. They repeat to send their permission until the privileged node is in their status set, or until the queue is empty.

When a node cedes its permission, it sends a yield message to the node inquiring. Its request is returned to the requesting queue and the inquiring node proceeds as if a release message had been received.

The total number of messages required for an entry to the CS to take effect is $|I_i-\{i\}| + 2(|R_i-\{i\}|)$. The algorithm assumes the existence of a reliable communications network. Message transfer delays are finite, but unpredictable, and message-order preservation is required.

Agrawal-El Abbadi algorithm. In their algorithm [2], Agrawal and El Abbadi impose a logical tree structure on the network and use the notion of coteries. Nodes in the system are logically organized into a binary tree structure. A mechanism is used to construct tree quorums (sets of nodes) derived from the tree structure. The tree quorum protocol is an alternative approach for the construction of quorums. A coterie consists of all the tree quorums generated from the binary tree. Any two tree quorums, members of the coterie, have a nonempty intersection and none of the tree quorums is a superset of any other one. A requesting node must obtain permission from all members in a tree quorum before it can make use of the critical section. The two conditions that must hold for the tree quorums in a coterie guarantee the mutually exclusive access to the critical section.

Their mutual exclusion algorithm is similar to Maekawa's [9] and Sanders' [22]. When a node i wants to enter its critical section (CS), it determines a tree quorum and sends a timestamped request to all nodes in the quorum. It then waits to receive permission from all members of the quorum. Each node maintains an ordered queue of pending requests. When a request message is at the head of the queue, the node sends its permission to the node that originated that request. When node j receives the request from i, it checks if the request at the head of the queue, if any, has a smaller timestamp. The request from i is placed in the queue. If the request at the head (before i's request was enqueued) has a greater timestamp, an inquire message is sent to check if the node has been successful in collecting permission from all other nodes in its quorum. If it has not, it cedes its permission by sending a yield message to the inquiring node. If it has succeded, it ignores the inquire messsage. If a yield message is received, node j sends its permission to i. When a node releases the CS, it sends a relinquish message to all nodes in its quorum. Upon the reception of a relinquish message, a node removes the served request from the queue and sends its permission to the request at the head of the queue.

All nodes in the binary tree that form a path from the root to a leaf define a tree quorum. If some node on the path fails, it is replaced by two paths starting from the children of the failed node and terminating with the leaves. If a leaf node on a path has failed, then a quorum cannot be formed.

Under the absence of failures the algorithm requires O(log N) messages exchanged for an entry to the CS to take effect, and it requires (N+1)/2 messages when some failures occur. The algorithm incorporates fault-tolerance by providing several alternative tree quorums to a requesting node. The algorithm may not be able to form a tree quorum in some cases after the failure of log N nodes. The algorithm is resilient to both node and communication failures and can be generalized to arbitrary trees.

Singhal's algorithm. In his algorithm [25], a dynamic information structure is used. The information structure at a node changes with the state of the system as the node receives messages from other sites. Every node i maintains an information structure which consists of two sets. The request set R_i, specifies the nodes from which i must obtain permission to enter into its critical section (CS). The informing set

I_i, specifies the nodes to which i must send its permission after it releases the CS. For all requesting nodes i and j, there is a non-null intersection between R_i and R_j; this guarantees the mutual exclusion condition.

Requests are ordered by the use of logical clocks according to Lamport's rules [8]. A node i must acquire permission from all nodes in its R_i to enter its CS. The set R_i changes by adding those nodes to which permission is sent, and by removing those nodes from which permission is received. The set I_i changes by adding those nodes from which a request is received when either node i is requesting and its request has a lower timestamp, or when the request is received when i is already in the CS. Nodes to which i sends its permission after it releases the CS are deleted from the set I_i.

When node i wants to enter its CS, it sends a timestamped request to all nodes in its R_i and waits to receive their permission. When a permission is received, the granting node is removed from R_i. When node j receives the request from node i, it can be either requesting, inside the CS, or not interested in the CS. When node j is requesting and its request has a lower timestamp, it adds node i to its I_j set. If its request has a greater timestamp, it sends its permission to i and if node i was not in its R_j set, it is added and a request is sent to it. When node j is in the CS, node i is added to its I_j set. If node j is neither requesting, nor executing the CS, it sends its permission and node i is added to its R_j set.

Under light load, the algorithm requires an average of (N - 1) messages exchanged for an entry to the CS to take effect. Under heavy load, it requires 3 * (N - 1)/2. The algorithm assumes that message transmission delay is finite, but unpredictable, and that messages are delivered in the order they were sent. The underlying communications network is assumed to be reliable. The impact of message loss and site failures is discussed and methods to tolerate these failures are proposed.

5.1 Recapitulation of the performance of permission-based algorithms.

The total number of messages exchanged for an entry to the CS to take effect can be reduced considerably if the nodes are logically organized [1]. By constructing intersecting quorums or coteries, mutual exclusion can be achieved efficiently as in [9], [22], [2], and [25].

Table 2 below shows the performance of the algorithms described above. The column in the center indicates the total number of messages required for a node to enter the CS.

ALGORITHM	TOTAL MESSAGES	OBSERVATIONS
Lamport [8]	3*(N-1)	
Ricart-Agrawala [20]	2*(N-1)	
Carvalho-Roucairol [3]	0 to 2*(N-1)	Indefinite permission.
Raynal [18]	$2*(N-1)^2$	Uses prime numbers.
Maekawa [9]	3*sqrt(N) to 5*sqrt(N)	Uses quorums.
Sanders [22]	|Ii - {i}| +2(|Ri - {i}|)	Uses quorums.
Agrawal-El Abbadi [2]	O(log N)	Uses tree quorums.
Singhal [25]	(N-1) to 3*(N-1)/2	Uses a dynamic information structure.

Table 2. Performance of permission-based algorithms.

Algorithms based on quorums provide more tolerance to faults than other algorithms [2, 1], because in the case of both node and communication failures, several alternative quorums are provided to a requesting node. These algorithms exhibit the property of graceful degradation [2]. The cost, in messages, of forming a quorum increases as failures increase, and the probability of forming a quorum decreases.

All the algorithms described in this section, as well as the ones in Section 4, allow only one node to be in the CS at a time. In the following section, three algorithms that allow multiple nodes to execute the CS simultaneously are presented.

6. DISTRIBUTED MUTUAL EXCLUSION ALGORITHMS THAT ALLOW MULTIPLE PROCESSES TO EXECUTE THE CRITICAL SECTION SIMULTANEOUSLY

In the last two sections, algorithms that allow only one process to execute in the CS at a time were described. Processes become sequentialized because they must wait to access a shared resource. In some cases, multiple processes could be allowed to execute in the CS simultaneously. Hence, a higher level of concurrency could be attained in the distributed system.

Raymond [17] and Kakugawa et al [7] developed permission-based algorithms to allow k nodes to execute the CS at the same time. Srimani and Reddy [26] use k tokens to allow up to k nodes to execute the CS simultaneously.

Raymond's algorithm. Raymond [17] extends the algorithm of Ricart and Agrawala [20] to allow up to k nodes in the system to execute the critical section (CS) simultaneously. A node is allowed to enter the CS when at least N - (k - 1) nodes are not executing within the CS. A requesting node sends a request message to all other N-1 nodes in the system and waits to receive permission from N-k nodes to enter its CS. A permission is implicitly received with a reply message. When the requesting node receives N-k reply messages, it is free to execute its CS. The remaining k-1 reply messages can arrive at the requesting node while it is in the CS, or after it has released it. Each node keeps record of the reply messages received by any other node in the system. A node i may defer many reply messages to a node j, when i is executing the CS and the requesting node j receives a reply from other N-k nodes while node i is still in the CS. This can occur many times while i is still executing its CS. When this happens, node i sends a reply message which contains the number of replies deferred for requests received from j.

When node i wants to enter the CS, it sends a request message to all other N-1 nodes. A request message contains the maximum sequence number in the system that the node has knowledge of. Node i then waits to receive permission from N-k nodes. When a reply arrives at node i, it checks the number of nodes that are not executing the CS, and if there are at least N-k+1 nodes out of the CS, node i is free to enter the CS.

When node j receives a request from i, it updates its highest sequence number known. If j is in the CS, or it is requesting the CS and its sequence number is smaller than that in i's request, the sending of a reply is deferred. Otherwise, node j sends a reply message to i. Ties are broken by favoring the smallest node number. Node j keeps record of the number of times it has deferred the sending of its permission to any other node.

When a node releases the CS, it sends a reply message to all other nodes for which a reply to their requests was deferred. The reply message contains the number of requests received which are being replied to.

The algorithm requires at most 2 * (N - 1) messages exchanged for an entry to the CS to take effect. It assumes the existence of a reliable communications network. Message transfer delays are finite, but unpredictable, and the order of reception of messages is unpredictable.

Srimani-Reddy algorithm. Srimani and Reddy developed an algorithm [26] to allow multiple simultaneous entries to the critical section (CS). Their algorithm is based on Suzuki and Kazami's algorithm [27]. There exist K tokens in the system to allow K nodes to execute simultaneously in their CS. K is fixed and 1<= K < N.

Each token contains information about the state of the system. This information can be updated in each token only when a node possesses the token. The K tokens are generally updated at different nodes. Therefore, each token will have different state information about the system. A major task in this algorithm is to keep the system information up-to-date.

Sequence numbers are bounded by the use of a large integer L > N. Because of the unpredictable delay in communication, as well as in the order of reception of messages, the information about all requests messages in the "previous bounded round" of any node i must be synchronized to ensure that they have been recorded in each token in the system. Each node maintains an array PLN of size N to keep record of the most recently request serviced for each node. This array is updated from the information contained in the tokens, and is used to update the queue of pending requests in any token. This will prevent a node from sending an additional token to a requesting node. A node i that possesses a token does not know if another node j has sent its token to the same requesting node to which i will send its own. The information in PLN prevents a node from sending unnecessary tokens whenever the information needed is available at the node. Each node keeps track of the number of tokens it possesses.

When a node i wants to enter its CS and does not hold a token, it increments its sequence number and initializes the variables for the "next bounded round," if necessary. Then, a request message (of the same form as in the original algorithm [27]) is sent to all other N-1 nodes, and i waits for the arrival of a token. Many tokens can arrive to service the request. When node i wants to enter its CS and possesses one or more tokens, it is free to enter. Whatever the case, when i can proceed to enter its CS, it decrements the number of tokens it possesses and enters its CS. When it leaves the CS, it updates the information in the token with its current request serviced. If necessary, it waits for the acknowledgement from all other N-1 nodes that indicates they all know a "new bounded round" of sequence numbers will commence for node i. It updates the information in the token it just made use of and, if there is a pending request in this token's queue, it sends the token to the requesting node. If no request was in the queue, node i increments the number of tokens it possesses.

When a request from i arrives at node j, it increments the request count for i. When necessary, j updates the information in each token it holds to indicate the start of a "new bounded round" of sequence numbers for node i. Also, j sends its acknowledgement to i, when a new request count will commence. Node j updates PLN[i], and if it holds a token and is not requesting, or it has a spare token, then the information in the token is updated and the token is sent it to node i.

A token has the form PRIVILEGE(Q, LN, LT), where Q and LN are the same as in the original algorithm [27], and LT is an array of size N which indicates that all requests from any node i in the "previous bounded round" have been recorded in the token. When a token arrives at node i, it updates the information of "new bounded rounds" for any node, including itself. If it is requesting the CS, it can make use of the right to enter. Otherwise, it updates the information in the token and if there is a request in its queue, it sends the token to the requesting node. If no request is in the queue, node i increments the number of tokens it possesses.

The algorithm requires N + K - 1 messages exchanged (L is a very large integer) for an entry to the CS to take effect. If a node holds at least one idle token, it can enter the CS without the need to send a request. The algorithm assumes the existence of a reliable communications network. Message transfer delay are finite, but unpredictable, and message-order preservation is not required.

Note: In their algorithm, the condition if RN[i]:=1 in procedure P1, should be equal to 2 rather than to 1. This typographical error must be corrected when reviewing the algorithm.

Kakugawa et al algorithm. Kakugawa et al use the notion of coteries in their algorithm [7], where k nodes in the system are allowed to enter the critical section (CS) simultaneously. A coterie is a set of sets of nodes for which two conditions must hold. The first, is that any two sets must have a non-null intersection, and the second condition is that no set is a subset of another one. Each set is a nonempty set and is called a quorum. All quorums in the coterie intersect with each other, so when a node receives permission from all other nodes in its home quorum it is guaranteed that only that node will be allowed to enter the CS. To allow multiple nodes to execute the CS at the same time the intersection condition in the coterie must be modified.

A k-coterie is an extension of a coterie. To construct a k-coterie, three conditions must hold. a) For any h quorums ($h < k$) that have a null intersection with each other, there exists another quorum that has a null intersection with one of these h quorums. b) For any $k + 1$ quorums there exists a pair that have a non-null intersection. c) For any two distinct quorums, none is a superset of the other. By the nonintersection property, if less than k processes are in the CS, then a requesting process can enter the CS by selecting an appropriate quorum. By the intersection property, at most k nodes can enter the CS simultaneously.

Kakugawa et al adopted Maekawa's algorithm [9], but a k-coterie is constructed. When node i wants to enter its CS, it increments the highest sequence number known and sends a request message to every other member of an appropriate quorum Q_i. An appropriate quorum would be one that has a null intersection with another quorum. Node i itself pretends to have received a request. When receiving a request from i, each member node j of Q_i checks if it has already granted its permission. If it has not, it sends its permission to i. Otherwise, it enqueues i's request in its ordered waiting queue. Either if the request granted, or if any request in the queue has a lower sequence number than that of i's, a failed message is sent to node i. Suppose that the request from node i has the lowest sequence number, then an inquire message is sent to the node to which the permission was granted, to check whether this node has obtained permission from all other members in its quorum. If this node has succeeded, it is free to access the CS. If it has not succeeded, it cedes its permission to free its member node j to service a request with a lower sequence number. Node j enqueues the unsuccessful request. It removes from its queue the request from node i, and sends it its permission.

When node i obtains permission from all other members of Q_i, it can enter its CS. Upon releasing the CS, a release message is sent to all members of Q_i. When a release message arrives at node j, it grants its permission to the next request in its waiting queue. No action is taken if the queue is empty.

As a simplistic example, assume k=2, N=4 and the quorums in the k-coterie are $q_1 = \{1,2\}$, $q_2 = \{1,3\}$, $q_3 = \{2,4\}$, and $q_4 = \{3,4\}$. Two nodes can execute the CS simultaneously. Suppose nodes 2 and 4 want to enter the CS, if node 2 chooses q_1, it acquires the permission from all of its members; if node 4 chooses q_4, it can acquire the permission from all of its members and hence, at most two nodes are in the CS simultaneously.

The algorithm assumes a fixed k, 1<= k < N. The construction of the quorums in the k-coterie significantly impacts the number of messages required to effect an entry to the CS. The message complexity of this algorithm is O(s), where s is the size of the largest quorum. The total number of messages required is C * s, where C is a constant between 3 and 5.

6.1 Recapitulation of the performance of multiple entries to the critical section algorithms.

Table 3 below shows the performance of the algorithms that allow multiple nodes to execute in the CS simultaneously. The column in the center indicates the total number of messages required for an entry to the critical section to take effect for each node.

ALGORITHM	TOTAL MESSAGES	OBSERVATIONS
Raymond [17]	2*(N-1)	Permission-based.
Srimani-Reddy [26]	0 or N+K-1	Token-based.
Kakugawa et al [7]	3*s to 5*s	Permission-based.

K = number of processes allowed simultaneous access.
s = size of the largest quorum.

Table 3. Performance of multiple entries algorithms.

In Raymond's [17] and Srimani-Reddy's [26] algorithms, a major concern is to maintain the information about the state of the system. In the former, the algorithm has to take care of replies received which correspond to a request already serviced. In [26], the information in the tokens must be updated to avoid sending a token to a request already serviced and sending unnecessary tokens.

A major task in Kakugawa's et al algorithm [7] is to construct the quorums for the k-coterie. It is not a trivial task.

7. CONCLUSIONS

In this survey, 21 distributed mutual exclusion algorithms have been presented. Their principles and characteristics have been described, and their cost in the number of messages exchanged for an entry to a critical section (CS) to take effect has been shown.

These algorithms can be classified into two groups, according to their major design approach. These two groups are token-based algorithms and permission-based algorithms. In the token-based approach, the right to enter the CS is given by the possession of a special object called "the token." The singular existence of the token ensures the requirement for mutual exclusion. The possession of the token implies the right to enter to the CS.

In the permission-based algorithm, the right to enter the CS is given by collecting enough rights from all nodes in a set. A node must obtain permission from all nodes in a particular set to enter the CS. Each node grants its permission to only one node at a time. Sets must intersect in a non-null pairwise manner. This ensures the mutual exclusion condition.

In token-based algorithms, different mechanisms are used for locating the token, circulating the token among requesting processes, and for ordering the events in the system. Token-based algorithms are highly susceptible to the loss of the token. Complex mechanisms, based on time-outs, must be executed in order to regenerate a lost token and to discard duplicates tokens.

In permission-based algorithms, a major concern is to find a minimum size of a set of nodes from which to obtain permission to enter the CS. Different structures are used to reduce the overhead of achieving mutual exclusion. A certain type of structure called coterie reduces the cost considerably. Coteries provide for a high level of fault-tolerance as well.

A major task in a distributed mutual exclusion algorithm is to reduce the number of messages exchanged for an entry to the CS to take effect. The number of messages can be reduced considerably if the nodes are logically strutured.

Three of the algorithms presented are designed to allow multiple processes to execute the CS simultaneously. Hence, a higher level of concurrency could be achieved.

REFERENCES

1 AGRAWAL, D., and EL ABBADI, A., "Exploiting logical structures in replicated databases," Information Processing Letters, vol. 33, no. 5, january 1990, pp. 255-260.

2 AGRAWAL, D., EL ABBADI, A., "An efficient and fault-tolerant solution for distributed mutual exclusion," ACM Transactions on Computer Systems, vol. 9, no. 1, feb. 1991, pp. 1-20.

3 CARVALHO, O., ROUCAIROL, G., "On mutual exclusion in computer networks, Technical Correspondence," Communications of the ACM, vol. 26, no. 2, feb. 1983, pp. 146-147.

4 GARCIA-MOLINA, H., BARBARA, D., "How to assign votes in a distributed system," Journal of the ACM, vol. 32, no. 4, 1985, pp. 841-860.

5 GIFFORD, D. K., "Weighted voting for replicated data," Proc. 7th Symposium on Operating Systems Principles, 1979, pp. 150-159.

6 HELARY, J., PLOUZEAU, N., RAYNAL, M., "A distributed algorithm for mutual exclusion in an arbitrary network," Computer Journal, vol. 31, no. 4, 1988, pp. 289-295.

7 KAKUGAWA, H.; FUJITA, S.; YAMASHITA, M. and AE, T., "Availability of k-Coterie," IEEE Transactions on Computers, vol 42, no. 5, may 1993, pp. 553-558.

8 LAMPORT, L., "Time, clocks, and the ordering of events in a distributed system," Communications of the ACM, vol. 21, no. 7, july 1978, pp. 558-565.

9 MAEKAWA, M., "A sqrt(n) algorithm for mutual exclusion in decentralized systems," ACM Transactions on Computer Systems, vol 3, no. 2, may 1985, pp. 145-159.

10 MAEKAWA, M.; OLDEHOEFT, A.E.; and OLDEHOEFT, R.R., "Operating Systems, Advanced Concepts," Benjamin-Cummings, 1987.

11 MISHRA. S. and SRIMANI, P., "Fault-tolerant mutual exclusion algorithms," Journal of Systems Software, vol. 11, no. 2, feb. 1990, pp. 111-129.

12 MIZUNO, M.; NEILSEN, M.L. and RAO, R., "A Token based distributed mutual exclusion algorithm based on Quorum Agreements," 11th Intl. Conference on Distributed Computing Systems, 20-24 may 1991, pp. 361-368.

13 NAIMI, M. and TREHEL, M., "How to detect a failure and regenerate the token in the log(n) distributed algorithm for mutual exclusion," Proc. of the Second Int Workshop on Distributed Algorithms, Lecture Notes in CS, 1987, pp. 155-166.

14 NEILSEN, M.L. and MIZUNO, M., "A DAG-based algorithm for distributed mutual exclusion," 11th Intl. Conference on Distributed Computing Systems, 20-24 may, 1991, pp. 354-360.

15 NISHIO, S.; LI, K.F. and MANNING, E.G., "A resilient mutual exclusion algorithm for computer networks," IEEE Transactions on Parallel and Distributed Systems, vol. 1, no. 3, july 1990, pp. 344-355.

16 RAYMOND, K., "A tree-based algorithm for distributed Mutual Exclusion," ACM Transactions on Computer Systems, vol. 7, no. 1, feb. 1989, pp. 61-77.

17 RAYMOND, K., "A distributed algorithm for multiple entries to a critical section," Information Processing Letters, no. 30, feb. 1989, pp. 189-193.

18 RAYNAL, M., "Prime numbers as a tool to design distributed algorithms," Information Processing Letters, vol. 33, no. 1, oct. 1989, pp. 53-58.

19 RAYNAL, M., "A simple taxonomy for distributed mutual exclusion algorithms," Operating Systems Review, vol. 25, no. 2, apr. 1991, pp. 47-49.

20 RICART, G. and AGRAWALA, A., "An optimal algorithm for mutual exclusion in computer networks," Communications of the ACM, vol. 24, no. 1, jan. 1981, pp. 9-17.

21 RICART, G.and AGRAWALA, A. K., "Author response to 'on mutual exclusion in computer networks' by Carvalho and Roucairol," Communications of the ACM, vol. 26, no. 2, feb. 1983, pp. 147-148.

22 SANDERS, B., "The information structure of distributed mutual exclusion algorithms," ACM Transactions on Computer Systems, vol. 5, no. 3, aug. 1987, pp. 284-299.

23 SILBERSCHATZ, A. and PETERSON, J.L., "Operating System Concepts," Addison-Wesley, Alternate edition, 1988.

24 SINGHAL, M., "A heuristically-aided algorithm for mutual exclusion in distributed systems," IEEE Transactions on Computers, vol. 38, no. 5, may 1989, pp. 651-662.

25 SINGHAL, M., "A dynamic information-structure mutual exclusion algorithm for distributed systems," IEEE Transactions on Parrallel and Distributed Systems, vol. 3, no. 1, jan. 1992, pp. 121-125.

26 SRIMANI, P.and REDDY, R., "Another distributed algorithm for multiple entries to a critical section," Information Processing Letters, vol. 41, no. 1, jan. 1992, pp. 51-57.

27 SUZUKI, I. and KASAMI, T., "A distributed mutual exclusion algorithm," ACM Transactions on Computer Systems, vol. 3, no. 4, nov. 1985, pp. 344-349.

28 THOMAS, R. H., "A majority consensus approach to concurrency control for multiple copy databases," ACM Transactions on Database Systems, vol. 4, no. 2, June 1979, pp. 180-209.

APPENDIX A

```
Procedure P1;
begin
  Requesting:=true;
  if not HavePrivilege then
    begin
      RN[I]:=RN[I] + 1;
      for all J in {1, 2,..., N} - {I} do
        Send REQUEST(I, RN[I]) to node J;
      Wait until PRIVILEGE(Q, LN) is received;
      HavePrivilege:=true;
    end;
  Critical Section;
  LN[I]:=RN[I];
  for all J in {1, 2,..., N} - {I} do
    if not in (Q, J) and (RN[J] = LN[J] + 1) then
     Q:=append(Q, J);
    if Q <> empty then
      begin
        HavePrivilege:=false;
        Send PRIVILEGE(tail(Q), LN) to node head(Q)
      end;
  Requesting:=false
end;


procedure P2; (* REQUEST(J, n) is received; P2 is indivisile *)
begin
  RN[J]:=max(RN[J], n);
  if HavePrivilege and not Requesting and (RN[J]=LN[J]+1)then
    begin
      HavePrivilege:=false;
      Send PRIVILEGE(Q, LN) to node J
    end
end;
```

Fig. 1. Suzuki and Kazami's algorithm.