Journal of Universal Computer Science, vol. 12, no. 2 (2006), 140-159
submitted: 18/1/05, accepted: 20/10/05, appeared: 28/2/06 © J.UCS

An O(√n) Distributed Mutual Exclusion Algorithm Using Queue Migration[1]

Pranay Chaudhuri
(University of the West Indies, Cave Hill Campus, Barbados
pchaudhuri@uwichill.edu.bb)

Thomas Edward
(University of the West Indies, Cave Hill Campus, Barbados
tedward@uwichill.edu.bb)

Abstract: In this paper a distributed algorithm is proposed that realises mutual exclusion among n nodes in a computer network. There is no common or global memory shared by the nodes and there is no global controller. The nodes of the network communicate among themselves by exchanging messages only. The proposed algorithm is based on queue migration and achieves a message complexity of O(√n) per mutual exclusion invocation. Under heavy load, the number of required messages approaches 2 per mutual exclusion.

Key Words: Mutual exclusion, critical section, distributed algorithm, computer network, queue migration, message complexity
Categories: C.2.4, F.2.2, I.1.2

1 Introduction

In a system with multiple processes which must share a common resource, it may be necessary to avoid multiple simultaneous access to that resource. Mutual exclusion ensures that the shared resource is accessed by at most one process at a time. For example, in a system with multiple processes, some or all of the processes may wish to update a common file. However, no two processes should be allowed to write to that file at the same time in order to ensure the integrity and consistency of the file. The section of code that allows for such mutual exclusive access of the shared resource is often referred to as the critical section.

In a centrally controlled system, it is not too difficult to implement the mutual exclusive use of the shared object. Semaphores and monitors are commonly used. However, in a distributed environment, the solution to this problem becomes far more complex due to the absence of a global or centralised controller, which makes these abstract data types ineffective.

In a distributed system, nodes communicate only by exchanging messages. A distributed mutual exclusion algorithm requires a mechanism such that if a node wishes to invoke mutual exclusion then all other nodes are aware of this (directly or indirectly) that they may themselves not enter into their critical section.

---

1. A preliminary version of this paper was presented in the 16th IASTED Intl. Conf. on Parallel and Distributed Computing and Systems, Cambridge, USA [Chaudhuri , 04].

A good deal of research has been reported in the literature on this problem. These solutions are said to be token based if a special message called a token that is used to grant access to the critical section [Banerjee, 96; Chaudhuri, 91; Chaudhuri, 95; Chaudhuri, 98; Raymond, 89; Suzuki, 85]. Otherwise, it is said to be permission based [Carvalho, 83; Lamport, 78; Maekawa, 85; Ricart, 81; Ricart, 83]. That is a node requesting the use of the critical section is required to send a request message to a group of nodes. When a sufficient number of affirmative replies have been received the node may enter the critical section.

In this paper, we propose an algorithm for distributed mutual exclusion based on queue migration which achieves a message complexity of $O(\sqrt{n})$ per mutual exclusion invocation, in the worst case. Under heavy load, the number of required messages approaches 2 per mutual exclusion.

The rest of the paper is organized as follows. Section 2 presents a literature review. In Section 3 we examine the basis of the algorithm followed by a more formal description of the algorithm. The pseudocode for the algorithm is given in the appendix. The correctness proof of the algorithm is provided in Section 4. Section 5 deals with the message and space complexities of the algorithm. In Section 6 we provide simulation results of our algorithm and provide a comparison of performance of this algorithm with four other algorithms. Finally, in Section 7 we conclude the paper.

2 Related Literature

Lamport provided the first truly distributed mutual exclusion algorithm [Lamport, 78]. In Lamport's algorithm, when a node j requires to enter its critical section it must send a time-stamped REQUEST message to all the other n-1 nodes in the network. When a PERMISSION message is received from all n-1 nodes, node j can then proceed to enter the critical section. Upon exit of the critical section node j sends a RELEASE message to all nodes to indicate that its request has been granted. For a system of n nodes, this algorithm requires $3(n-1)$ messages. Later Ricart and Agrawala [Ricart, 81] proposed an algorithm that requires $2(n-1)$ messages. In their algorithm, Ricart and Agrawala required that a node desirous of entering its critical section sends a REQUEST message to, and receive permission from, all the other nodes in the network. No RELEASE message is necessary since a node will grant its permission only if it is itself not requesting or its request is preceded by the request of the other node. Sequence numbers are used for ordering requests in the system. The above algorithms are symmetric and fully distributed in the sense that all node have identical copies of the algorithm and that all nodes participate in the decision to grant mutual exclusion, respectively.

Carvalho and Roucairol [Carvalho, 83] presented an algorithm which fulfilled similar requirements as Ricart and Agrawala's solution but requires between 0 and $2(n-1)$ messages. Once a node j has received permission from a node i, this permission is kept as valid until node j receives subsequently a request for critical section from node i. When node j wants to enter the critical section, request messages will be sent only to the nodes for which node j does not have valid permissions.

Later, Ricart and Agrawala [Ricart, 83] proposed an algorithm that requires 0 or *n* messages. In this algorithm, a token is used to grant the right to the critical section. A node requesting the use of the critical section sends a request message to all other nodes. The node holding the token, will send the token to the requesting node if it no longer requires the token. If a node holds the idle token then it may enter its critical section without sending further requests. All these algorithms share a common feature. They require that all nodes directly participate in the decision for granting mutual exclusion. Hence, the message complexity of these algorithms will always be bounded from above by O(*n*).

Suzuki and Kasami [Suzuki, 85] used the concept of the privileged node in their solution to this problem. There is one token in the system and the node which possesses the token has the right to enter its critical section. When a node wants to enter the critical section, it sends a request message to all other nodes in the system and waits for the token. The token holder, if not in the critical section and do not wish to enter the critical section, will grant the token to the requesting node. This algorithm required *n* messages per mutual exclusion invocation. Maekawa, extended the concept of privileged node and further reduced the number of messages required to $c\sqrt{n}$ messages per mutual exclusion where c lies between 3 and 5 [Maekawa, 85]. Quorums based on a finite projective plane are obtained such that, for any pair of quorums Q_i and Q_j , $Q_i \cap Q_j \neq \Phi$ where Q_i is the quorum to which node i belong and $|Q_i| \approx \sqrt{n}$. A node requesting the use of its critical section must first obtain permission from all members of its quorum. The condition $Q_i \cap Q_j \neq \Phi$ guarantees mutual exclusion. Another distributed mutual exclusion algorithm was presented by Helary et al [Helary, 88] and requires that the number of messages vary from *n* to $2e+n-1$ depending on the network topology, where *e* is the number of links in the network. A node, in order to enter the critical section, must send request messages only to its immediate neighbours and wait for the token. The request message is propagated through the network until the node with the token is found. This privileged node, if not requesting the use of the critical section will send the token along the reverse path of propagation to the requesting node.

Raymond proposed an algorithm for distributed mutual exclusion based on a static logical tree structure [Raymond, 89]. The nodes of the tree are arranged to form an unrooted tree. Nodes communicate only with their neighbours. The node holding the token becomes the privileged node and a directed path is formed from any node in the network to this privileged node. A node j requesting mutual exclusion sends a request message to its neighbour which then forwards a request to its neighbour node along the directed path, if it is itself not the holder of the token. When the token holder receives the REQUEST message and does not require the use of the token, it relinquishes the token to the requesting neighbour which then passes it on to the neighbour nodes until node j is reached. The tree is now reconfigured to have the directed paths pointing towards node j. Since the number of messages vary depending on the topology of the network, Raymond proposes that the *average* number of messages per mutual exclusion invocation is O(log *n*). The worst case topology gives O(n).

An optimal algorithm for mutual exclusion in a mesh-connected computer network was proposed by Chaudhuri [Chaudhuri, 91]. The network nodes are considered to be available as an $\sqrt{n} \times \sqrt{n}$ mesh with wrap around connections between

nodes of the same row or column. A binary variable token(i, j) is assigned each node. Initially token(i, j) = 1 for all elements of the first row and for any other row i, the (i – 1)th node is assigned a token of value 0 whereas all other node have token value 1. In addition to token(i, j) there exist a PRIVILEGE message. A node(i, j) can enter the critical section if it has acquired the PRIVILEGE, that is, row i is privileged and node (i, j) has now got its turn among the nodes in that row. Row i is said to be privileged if for all nodes in row i, token(i, j) = 1. The PRIVILEGE message is circulated in a round robin fashion in that row. If a row i is not privileged, then one node (i, q) in that row has token(i, q) = 0. Row i interchanges token(i, q) with token(p, q) of the privileged row and becomes the privileged row. Node (i, q) then initiates the circulation of the PRIVELEGE message. This algorithm requires $3.5\sqrt{n}$ messages under light demand and the message requirement reduces to four messages only under heavy demand. Later, Chaudhuri proposed a distributed mutual exclusion algorithm for an arbitrary network that extended the above concept [Chaudhuri, 95]. A two-dimensional $\sqrt{n} \times \sqrt{n}$ mesh is used such that any node (i, j) is connected by a bi-directional link to each node of its row and each node of its column. The mechanism of the algorithm is similar to that given in [Chaudhuri, 91] above and requires $3\sqrt{n}$ messages under light demand and $\sqrt{n}$ messages under heavy demand.

Chaudhuri and Karaata [Chaudhuri, 98] later proposed an $O(n^{1/3})$ distributed mutual exclusion algorithm. The nodes in the network are arranged as a three-dimensional mesh p x q x r such that $p = q = r = n^{1/3}$. A node in the x-y plane requesting the use of the critical section propagates a request message to the corner node which is along the z-axis. If the token is present and is not being used, a PERMIT message will be sent to the requesting node. A release message is sent back to the corner node upon completion of the critical section.

In Banerjee and Chrysanthis [Banerjee, 96] presented a token based distributed mutual exclusion algorithm for a fully connected network which works as follows. A node k (request collector) is selected at initialization to which requests are sent from nodes wishing to enter the critical section. Node k also holds the token. When a request is received at node k the request queue at k is copied to the token and is sent to the node indicated at the front of the queue. Node k broadcasts a message to all nodes announcing a new request collector, the node at the rear of the token queue. The token is passed from node to node in the order presented in the token queue until the last node is reached. The token is now appended to the request queue at the request collector node and the process is repeated. In the event that a node had requested to node k before it received the broadcast but after the token was sent, node k transfers the token to the new request collector. The message complexity is O(n) under light load and tends to 3 messages per critical section invocation under heavy load.

3 Distributed Mutual Exclusion

3.1 The Network

The network is assumed to be fully connected. We partition the n nodes of the network into $\sqrt{n}$ sets of $\sqrt{n}$ nodes each. Each set is called a *local group (LG)*. Nodes in a local group can communicate directly with each other for the purposes of entering

the critical section. That is, all nodes in a local group are fully connected. One node from each group is selected to form part of the *global group (GG)*. This node is called a *link-node*. Each node of the global group can communicate with all other nodes of the global group and also with all nodes of its local group to which they belong. The figure below shows a sample network of 16 nodes. Each local group is made up of four nodes and one node from each local group is a member of the global group.

Each local group is denoted by LGi, i = 0, 1, …, $\sqrt{n}$ –1. Each node is identified by an ordered pair (*i, k*) where $i \in \{ 0, 1, \ldots, \sqrt{n} -1\}$ indicates the local group number and $k \in \{ 1, 2, \ldots, \sqrt{n} \}$, the node number. When $n \neq k^2$ for $k \in \{ 1, 2, 3, \ldots\}$ we first create $\lceil \sqrt{n} \rceil$ groups and add a node to each group node in turn, until we have added all nodes. Here $|LGi| \leq \sqrt{n}$.

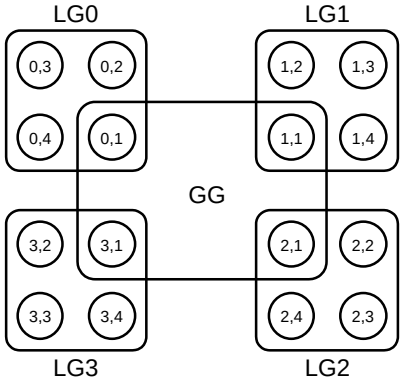

Figure 1: A network of 16 nodes, with the global group (GG) and 4 local groups (LG)

3.2 Basis of the Algorithm

The proposed algorithm is token based and only one token exists in the network except during the transition when a node grants the token to another node and the receiving node is yet to receive it. Permission to enter the critical section is granted by the acceptance of the token.

In every local group, there exists a node designated as the *local request collector (LRC)* known to all nodes in that group. When a node of a local group wants to enter the critical section it sends a request message to the LRC. The LRC enqueues all requests received into its local request queue. Every node maintains a local request queue. If the request collector is holding the idle token, the local request queue is copied to the token queue and the token together with the token queue are sent to the node which is at the front of the token queue. A request collector message is then sent to all nodes in the local group to indicate that the last node in the token queue is the new LRC. It must be noted that whenever a request queue or part thereof is copied to the token queue all nodes copied are deleted from the request queue.

A requesting node, upon receiving the token, deletes its node_id from the token queue and enters the critical section. Upon completion of the critical section

operation it forwards the token to the next node in the token queue. The process is repeated until the last node of the token queue, that is, the new LRC, is reached. Meanwhile, the new LRC may be receiving new request messages form other nodes, that is, the new LRC is in the request collecting phase.

A node, which previously was an LRC may receive a request message sent to it just before the new LRC information was received by the requesting node. This request is simply forwarded to the new LRC.

In the global group, there exists a node designated as the *global request collector (GRC)* known to all nodes in the group. Each link-node maintains a variable that holds the node_id of the GRC of the global group. In addition to the local request queue, a global node also maintains a global request queue. If a local group does not possess the token the link-node will be the LRC of its local group. When a node from such a local group wants to enter the critical section it sends a request message to the LRC which then forwards a request message to the GRC. The GRC, enqueues the request message in a global request queue, maintained for that purpose, and a marker is inserted into the local request queue of the GRC. If the GRC has the token and is idle, the token is sent to the requesting link-node. Otherwise, it must wait for the token to arrive.

When the token arrives at a link-node it checks whether it had requested the critical section. If so, it then performs the critical section operation. After completion of the critical section operation, or after determining that it was not interested in mutual exclusion, it examines its local request queue. If it is empty it retains the token. If the first entry is the marker, the global queue is copied to the token queue and the token together with the token queue are sent to the link-node which is at the front of the token queue. If, however, the front element of the local queue is a local node, then that part of the local request queue above the marker is copied to the token queue followed by the id of the link-node. If the link-node was itself requesting the critical section and its node_id was not the first in the queue, only that portion of the queue up to and including the link-node is copied to the token queue. The token and the token queue are then forwarded to the local nodes. The token will arrive at the requesting nodes in the same order as they appeared in the request queue. In such cases it is not necessary to broadcast the new LRC message since the token will eventually return to the link-node, as it is the last node in the token queue. When the token is returned to the link-node, if the marker is at the front of the local request queue, the token is sent to another link-node and the marker will be deleted from the local request queue.

When a link-node requires to send the token to another link-node but finds that its local request queue is not empty, it does one of the following: (i) if the link-node is the GRC it adds its node_id to the global request queue so that the token is returned to it eventually (ii) if the link-node is not the GRC, a request message is sent to the GRC, indicating that the token is still requested in that local group.

If a link-node is the GRC and its global request queue is empty, the idle token will reside with one of the local nodes in the local group to which the GRC belongs. If the GRC receives a token from a local node and the token queue is not empty, that implies that the link-node is not the LRC and that it requires the use of the critical section. If, however, the global queue of the GRC is not empty the GRC does the following upon completion of the critical section. The nodes above the marker in the

local request queue are copied to the token queue. The link-node then adds its node_id as the last entry of the token queue and the token and the token queue are then sent to the first node of the token queue. The token is eventually returned to the link-node for transmission to another link-node.

When the GRC send the token to another link node, a new GRC message is broadcasted to all other link nodes indicating that the node at the rear of the token queue is now the new GRC, providing that this node is not itself.

The concept of a marker has been used in this algorithm to prevent the token from circulation indefinitely in any one group. When the link-node, which is not the GRC, receives the token a marker is inserted into the local request queue. This causes only requests enqueued before the token was received to be serviced and then making the token available for another group. If the node is the GRC and receives a request from another link-node, a marker is inserted into the local request queue of the GRC. This indicates the point at which the token will be made available to other link-nodes. Only one marker may be inserted into the local request queue of the link-node. This means that once a marker has been inserted into the local request queue, upon receipt of the first global request, no further markers need be inserted into the local request queue.

Although the network is assumed to be fully connected with $n(n-1)/2$ links, the number of links used for message passing to achieve mutual exclusion based on the proposed algorithm is $(\sqrt{n}+1)*[\sqrt{n}(\sqrt{n}-1)/2] = \sqrt{n}(n-1)/2$.

3.3 Variables at a Node

local_request_collector : an integer variable that holds the id of the current local request collector node.

global_request_collector : an integer variable that holds the id of the global request collector node. This is only available at link-nodes.

token_here: a boolean variable which is true if the node is the current holder of the token.

Requested: a boolean variable which is true if the node has made a request for the use of the critical section.

node_id: an ordered pair of integers (local group, node number), to identify a node in the network.

Mutex: a local boolean variable which is TRUE if the corresponding node has got its turn to enter into its critical section and remains TRUE until the critical section operation is completed.

In addition, every node maintains a local request queue (LRQ) which is used to enqueue the node_id of all requesting nodes in its local group if this node is the LRC. Each link-nodes in addition, maintains a global request queue (GRQ) where it enqueues requests from other link-nodes if this link-node becomes the GRC.

We also require the following operations and assume that standard procedures corresponding to them are available.

enqueue (Q,(x, z)): Inserts (*x, z*) into the rear of queue Q.

delete(Q,(x, z)): Deletes an element (*x ,z*) from queue Q.

dequeue (Q,(x, z)): Delete Q.front.

read(Q,(x, z)): Returns the first element of the queue Q as (x, z) without

deleting it from the queue.

copy(Q1, Q2, j): Copies the elements of Q1 to Q2 up to but not including element j, deleting each element from Q1 after being copied to Q2.

We also use the following notation when referring to the elements of a queue Q. Q.front refers to the front element of the queue and Q.rear refers to the rear element of that queue.

Initially, the link-node of LG0 is set to have token and to be the LRC for that group as well as the GRC. All other link-nodes are initialised as the LRC of their respective groups. All the remaining variables are initialised to the empty or null state. The proposed distributed algorithm, assumes that the messages are received without errors and that they require finite time to move from a node to another.

3.4 Messages Used

In this section, we first describe the structure of the messages and the type of the variables used by the algorithm, and then present a more formal description of the algorithm based on the ideas of Section 3.1.

A set of four different types of messages is used by this algorithm. A general syntactic description of these messages is as follows.

‹message› ::= (‹type›, ‹node_id›, ‹parameter›)
‹type› ::= REQUEST | TOKEN| LR_COLLECTOR | GR_COLLECTOR
‹node_id› ::= an ordered pair of integers
‹parameter› ::= queue of ‹node_id›s.

The meaning of the different types of messages is explained below.

REQUEST: A REQUEST message from node (i, *k*), whose final destination is always the request collector (LRC/GRC), implies that the sender wants to enter into its critical section and is waiting to receive the TOKEN message. The node_id field holds the id of the node making the request. The parameter field of a REQUEST message may be empty or hold a request queue of node_ids.

TOKEN: The TOKEN message is sent to a requesting node to grant permission to enter the critical section. The parameter field of a TOKEN message carries a queue, token_Q. If the TOKEN message is sent to a local node it carries the LRQ of the LRC. Otherwise, if sent to another link-node it carries the GRQ. A node (*i*, *q*) upon receiving a TOKEN message enters the critical section. Upon exit of the critical section, it forwards the token to the next node (if any) in the token queue.

LR_COLLECTOR: This is a message broadcasted to all members of a local group to update the local_request_collector variable. When the current LRC sends the token to a node, it broadcasts a new LR_COLLECTOR message which contains the node_id of the last node in the token queue. This node becomes the next LRC. The node_id field contains the id of the new LRC while the parameter field of this message is empty.

GR_COLLECTOR: This is a message broadcasted to all members of the global group to update the global_request_collector variable. When the current GRC sends the token to a link_node, it broadcasts a new GR_COLLECTOR message which

contains the node_id of the last node in the token queue. This node becomes the next GRC. The node_id field contains the id of the new LRC while the parameter field of this message is empty.

3.5 The Algorithm

The detailed pseudo-code presentation of the algorithm DISTRIBUTED_ MUTEX is provided in the Appendix.

3.6 Illustrative Example

Figure 2 illustrates how the algorithm DISTRIBUTED_MUTEX works with a network of 16 nodes. Initially node (0, 1) holds the token and is the LRC of its Local Group and the GRC of the Global Group. The Global Group constitutes the nodes (j, 1), $j \in \{0, 1, 2, 3\}$.

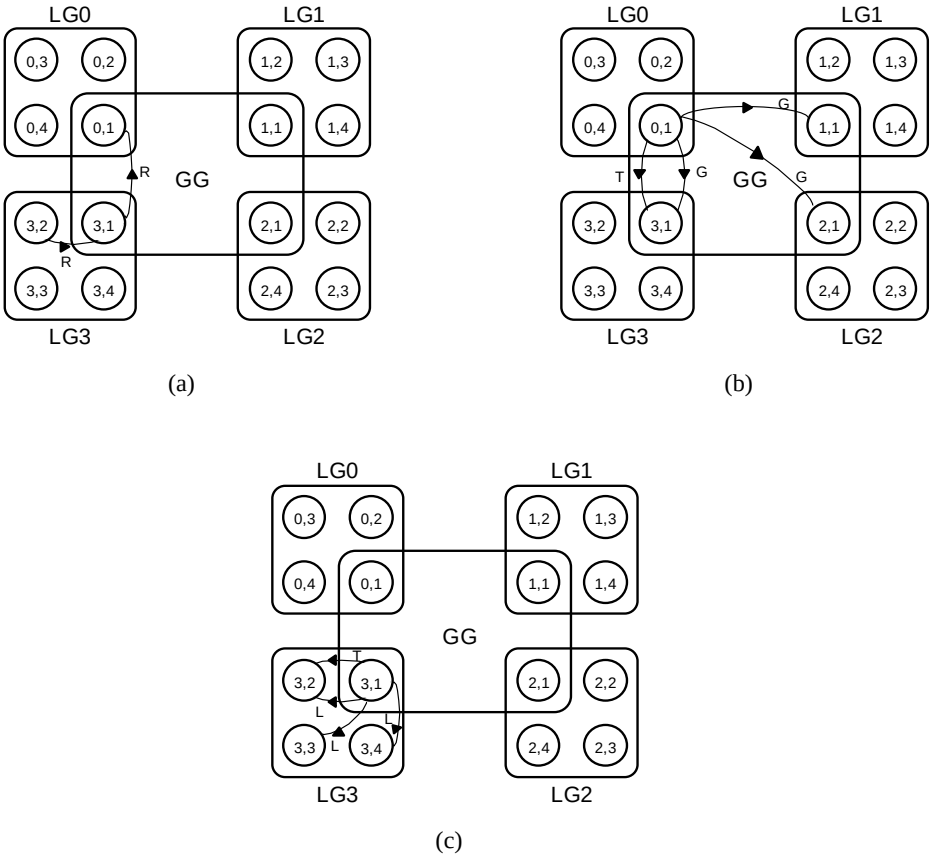

Figure 2: This figure illustrates the sequence of various messages used by algorithm DISTRIBUTED_MUTEX in a network of 16 nodes: (a) sequence of requesting messages (R) (b) token (T) sent from one link-node to another and new GRC (G) broadcasted (c) token (T) sent to the requesting node (3, 2) and new LRC (L) broadcasted

If node (3, 2) requires to enter the critical section it sends a REQUEST message to node (3, 1), which is the link-node of LG0 and also the LRC. Upon receipt of the request, node (3, 1) enqueued the request in the LRQ and a REQUEST message is sent to node (0, 1), the GRC (Figure 2 (a)). When the global request is received at node (0, 1) it is enqueued in the GRQ and a Marker is placed in the LRQ. Since the token is available, the GRQ is copied to the token queue and the token and token queue are sent to node (3, 1). Node (0, 1) then broadcasts a new GRC message to all nodes in the Global Group and deletes the Marker from the LRQ (Figure 2(b)).

Upon receiving the token, node (3, 1) deletes its id from the token queue. The LRQ is examined and the request from node (3, 2) is found. The LRQ is copied to the token queue and the token is forwarded to node (3, 2). The link-node (3, 1) then broadcasts a new LRC message to all nodes (3, k), where $k \in \{1, 2, 3, 4\}$ (Figure 2(c)). Node (3, 2) upon receipt of the token may enter the critical section.

4 Correctness of the Algorithm

Correctness of algorithm DISTRIBUTED_ MUTEX is established through the following lemmas.

Lemma 1 *Algorithm DISTRIBUTED_MUTEX ensures that a node can be inside its critical section, only if it has the token.*

Proof: To enter the critical section a node must make a request for the token. Once the token has been received, the node may enter the critical section. The node keeps the token until it has completed the critical section. Therefore, if a node is in the critical section then it must have the token.

Lemma 2 *Algorithm DISTRIBUTED_MUTEX ensures that there cannot be more than one token in the network at any time.*

Proof: The algorithm initializes the system to possess only one token. Given that there is no token generating function, and assuming that the token is not lost, there can be only one token in the system. A single exception is that, during the transition when a node grants the token to another node but the receiving node is yet to receive the token, then no node holds that token. Therefore, there can be at most one token in the network.

Lemma 3 *Algorithm DISTRIBUTED_MUTEX ensures mutual exclusion.*

Proof: The existence of at most one token in the network ensures mutual exclusion. A node must possess the token to enter the critical section (see Lemma 1) and releases the token only when it exits the critical section. Hence the lemma follows.

Lemma 4 *Algorithm DISTRIBUTED_MUTEX is deadlock-free.*

Proof: Deadlocks occur when the token cannot be passed on to a requesting node or

the token is lost.

We assumed that the message delivery system of the network is reliable and hence there is no loss of the token. We will now show that the token cannot be held indefinitely by one node while there are other requesting nodes in the system.

A node will only possess the token if it is in the critical section or it has completed the critical section and all its request queues are empty. This token is called an idle token. A node requesting the critical section will forward a request to that node (LRC) which may be holding the idle token or waits for the arrival of the token. The token will subsequently be relinquished and forwarded to the requesting node. Other nodes (not LRC), which may possess the token must have a non-empty token queue since a token with a non-empty queue is sent to those nodes. Upon completion of the critical section, the token is forwarded to the first node in that token queue.

If a link-node receives a request from a local node when the token is in another Local Group a request is sent to the GRC and the link-node waits for the token. That request at the GRC will eventually be enqueued in the token, since markers are used to prevent the token from circulating indefinite in a group, and to ensure that the token is passed on to another group.

Hence, the token cannot be held indefinitely by any node, while there are other requesting nodes.

Lemma 5 *Algorithm DISTRIBUTED_MUTEX is starvation-free.*

Proof: We show that a node cannot wait indefinitely for the token while other nodes are entering their critical sections (of course, one at a time). When a node sends a request for the token this request is stored at the request collector. If the request collector is not a link-node, and it is in possession of the token, the node_id of the requesting node is copied to the token queue. The token is then forwarded to the first node of the token queue. Since no node can hold the token indefinitely (by Lemma 4) the requesting node will receive the token within a finite time.

If the request collector is a link-node then the local request from node(i, j), will be enqueued in the LRQ. If the link node is also the GRC and receives a global request(s) then a marker will be inserted into the LRQ. This marker may appear either above or below the local request for (i, j) depending on the time of arrival of the global request. If the request from node(i, j) is enqueued above the marker and the link-node possesses the token, the request will be satisfied when the token is sent to the nodes listed above the marker. If however, the request is enqueued below marker in the LRQ, the token is first sent to other local nodes listed above the marker, then to the global nodes listed in the GRQ. The token is subsequently returned to the local group via the link-node. This time the request of node_id (i, j) will be above the marker (if any) in the LRQ. The node_id of the requesting node (i, j) will be copied to the token queue and the request will be satisfied within a finite time.

If a request from a local node is enqueued at a link-node which does not possess the token, the link-node will have to send a request message to the GRC. On receipt of this REQUEST message, the GRC inserts a marker into its LRQ. When the marker becomes the first element of the LRQ of the GRC, and the token returns to the GRC, the token is sent to the link-node. This link-node then forwards the token to the requesting local node(s).

A node may have forwarded its request to LRC before the new LRC message was received. Upon receipt of the REQUEST message the node will simply forward the REQUEST to the new LRC.

Theorem 1 *Algorithm DISTRIBUTED_MUTEX is correct.*

Proof: Follows directly from Lemmas 3 - 5.

5 Algorithm Complexity

5.1 Message Complexity

We will consider the worst case performance of the algorithm and also the performance of the algorithm under heavy load.

The worst case situation occurs when a non-link node $y \in LGi$, wants to enter the critical section but the idle token is held by a non-link node k in LGj, i j. Without loss of generality, we assume that there is a very low demand for the use of the critical section. Node y must send a request message to the LRC of LGi. This LRC, the link-node, forwards a request to the GRC, the link-node LGj. The GRC forwards a request to the LRC (token holder) of its group. This process requires three request messages to locate the token. The token is then forwarded to the requesting node on the reverse path giving a total of three token messages. The local node in LGj would have broadcasted $\sqrt{n} - 1$ new LRC messages. Also the link-node (ie GRC) of that group would have broadcasted $\sqrt{n} - 1$ new GRC messages and the link-node at LGi broadcasted $\sqrt{n} - 1$ new LRC messages. Hence the total number of messages for the execution of this critical section is $6 + 3(\sqrt{n} - 1) = O(\sqrt{n})$.

Under heavy demand every node in the system is desirous of entering the critical section continually. Suppose that the link-node of LGj is the LRC as well as the GRC and is the current holder of the token. Both queues (LRQ and GRQ) of that link-node will be full. To service all nodes would require $\sqrt{n} \times \sqrt{n} = n$ request messages to the LRCs, $\sqrt{n} - 1$ request messages to the GRC, for a total of $n + \sqrt{n} - 1$ request messages. The token is sent to all nodes in a group in turn, for a total of $n + \sqrt{n} - 1$ token messages. The link-nodes remain the LRC of their respective local group and hence no new LRC message is required. Therefore, for n critical section operations, the number of messages per critical section is $2(n + \sqrt{n} - 1)/n = 2(1 + 1/\sqrt{n} - 1/n)$. This value approaches 2 as n gets large.

Theorem 2 *Algorithm DISTRIBUTED_MUTEX requires only $O(\sqrt{n})$ messages per mutual exclusion invocation in the worse case. Under heavy load the number of messages per critical section invocation approaches 2.*

Proof: Follows from the above discussion.

5.2 Space Complexity

The following theorem provides upper bounds to the message length and the queue length used by the algorithm.

Queues of length √n are required for both the LRQ and GRQ. Two registers are required to hold the node_ids of the LRC and GRC. Other variables such as *mutex, token_here* and *requested* simply require that a corresponding bit be set. The TOKEN message will carry a token queue of length at most √n. Hence, in the worse case we require √n + √n + √n + 3 words of space for the algorithm. Hence, the algorithm has a O(√n) space requirement. This compares favorably with the algorithms of [Banerjee, 96] which requires O(n) space and [Chaudhuri, 98] which requires $O(n^{1/3})$ space. Raymond's algorithm, [Raymond, 89], however requires only O(1) space, but at the cost of a higher worst case message complexity.

Theorem 3 *To implement algorithm DISTRIBUTED_MUTEX, (i) the maximum length of the messages required is bounded from above by O(√n) words; (ii) the maximum space required by a node is bounded from above by O(√n) words.*

Proof: Follows from the above discussion.

6 Simulation Results

We present the simulation results for algorithm DISTRIBUTED_MUTEX in this section. In Figure 2, the performance of the algorithm under 5%, 25% and 70% load levels is shown together with the theoretical best case. In Figures 4, 5 and 6, the performance of algorithm DISTRIBUTED_MUTEX (= E) is compared with four other algorithms ([Chaudhuri, 98] = (A), [Raymond, 89] = B(tree), [Raymond, 89] = C (linear array), [Banerjee, 96] = D). The number of critical sections used is the simulation is k > 2n, where n is the number of nodes, and a critical section lasts for three time units.

The graphs in Figure 3 – 6 show that the DISTRIBUTED_MUTEX compares favorably with the other algorithms, even outperforming these algorithms at the lower load levels (5% and 25%). It is noted that Algorithm DISTRIBUTED_ MUTEX performs almost identically under heavy load (70%), to the network given as a linear array in [Raymond, 89].

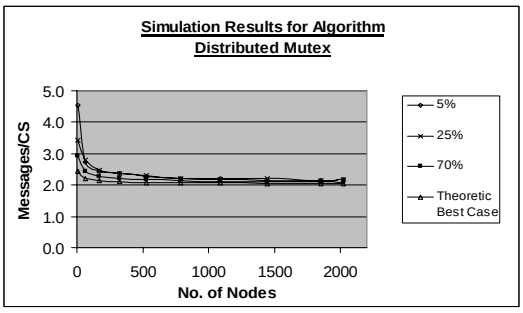

Figure 3: Performance of Algorithm DISTRIBUTED_ MUTEX at various load levels

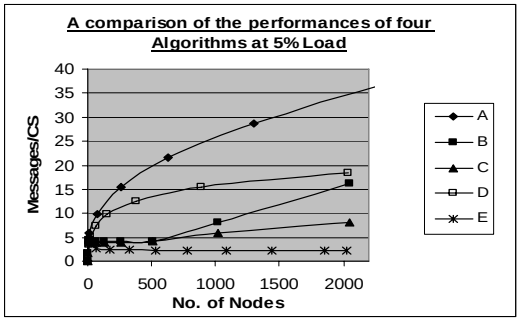

Figure 4: Comparison with four other Algorithms at 5% Load

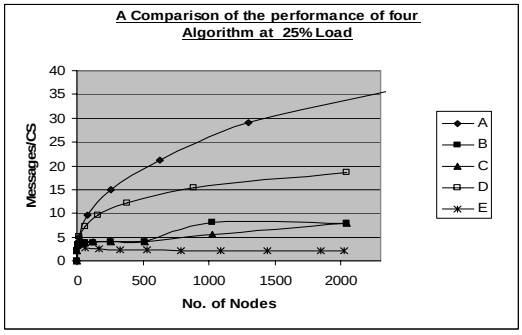

Figure 5: Comparison with four other Algorithms at 25% Load

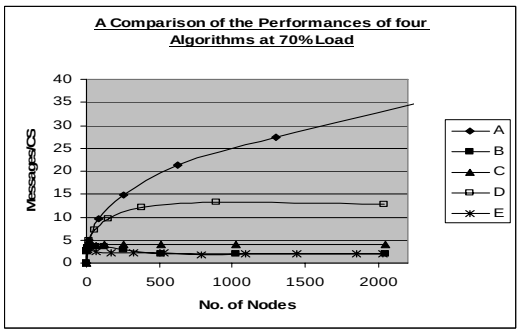

Figure 6: Comparison with four other Algorithms at 70% Load

The graph in Figure 3 is interesting in the sense that as n increases the graph approaches a limiting value of 2 messages per critical section entry, at every load level.

The reason for this seems to be as follows. Let P(load) be the probability that a request for the critical section will be generated. For small n, n = 9 for example, and

load = 0.05 the number of request generated every 4 rounds is small, approximately 2, compared to n = 900, where the number of requests generated will be approximately 45 in one round.

As a result, in the former case, the token is idle more often and many more LRC and GRC messages per critical section are generated as compared to the later. Thus, irrespective of the load level, as long as the number of nodes is large, the number of messages per critical section approaches 2.

7 Conclusion

This paper proposes a non-probabilistic token passing distributed mutual exclusion algorithm. The algorithm relies on queue migration and requires $O(\sqrt{n})$ messages per critical section entry in the worst case. Under heavy load the number of messages required per critical section entry approaches 2. The algorithm is shown to be free from deadlocks as well as from starvation.

We also claim that our algorithm is symmetrical in the sense that all nodes have identical copies of the algorithm. The algorithm is fair with respect to "responsibility sharing" and scheduling of the critical section. The system offers no preference to a node or set of nodes and that the distribution of waiting time for the token is likely to be the same for each node. Suppose, that the token resides at a node x in Local Group A. Assume that at time *t* node x originates a request message as does node y of Local Group B. We can only guarantee that the two nodes will be serviced within a time interval bounded from above by $(n - 1)()t + \vartheta)$, where $)t$ is the upper bound on the time to perform the critical section operation and ϑ is the upper bound on the time for the delivery of a message from one node to another. If the request of node x is inserted in the local queue above the marker, that request will be satisfied when the token is sent to the local nodes in the first round. If however, the request is enqueued below the marker then the token will be sent to other link-nodes before that pending request can be satisfied. Hence, there is no predictable pattern on the sequence in which the nodes will be serviced. Thus, in the worst case a node will have to wait until all other nodes have entered the critical section for a total time of $(n - 1)()t + \vartheta)$.

References

[Banerjee, 96] S. Banerjee and P. K. Chrysanthis, A new token passing distributed mutual Exclusion algorithm, Proceeding of Intl. Conf. on Distributed Computing Systems (ICDCS), Hong Kong, 1996, 717 - 724.

[Carvalho, 83] O. Carvalho and G. Roucairol, On mutual exclusion in computer networks, *Communications of the ACM 26*, (1983) 147 - 148.

[Chaudhuri, 91] P. Chaudhuri, Optimal algorithm for mutual exclusion in mesh-connected computer networks. *Computer Communications 14*, 1991, 627 - 632.

[Chaudhuri, 95] P. Chaudhuri, An algorithm for distributed mutual exclusion. *Information and Software Technology 37*, 1995, 375 - 381.

[Chaudhuri, 98] P. Chaudhuri and M. H. Karaata, An $O(n^{1/3})$ algorithm for distributed mutual exclusion, *Journal of Systems Architecture 45*, 1998, 409 - 420.

[Chaudhuri, 04] P. Chaudhuri and T. Edward, An $O(\sqrt{n})$ mutual exclusion algorithm in decentralized systems using queue migration, Proceeding of the 16th IASTED Intl. Conf. on Parallel and Distributed Computing and Systems, Cambridge, USA, 2004, 197 - 204.

[Coffman, 73] E. G. Coffman Jr. and P.J. Denning, *Operating Systems Theory*, (New York, Prentice-Hall, 1973).

[Hansen, 73] P. B. Hansen, *Operating Systems Concepts*, (New York, Prentice Hall, 1973).

[Helary, 88] J. M. Helary, N. Plouzeau and M. Raynal, A distributed algorithm for mutual exclusion in an arbitrary network, *Computer Journal 31*, 1988, 289 - 295.

[Maekawa, 85] M. Maekawa, A $\sqrt{n}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems 3*, 1985, 344 - 349.

[Maekawa, 87] M. Maekawa, A.E. Oldehoeft and R. Oldehoeft, *Operating Systems: Advanced Concepts*, (Menlo Park, CA, The Benjamin/Cummings, 1987).

[Lamport, 78] L. Lamport, Time, clocks and the ordering of events in a distributed system, *Communications of the ACM 21*, 1978, 558-565.

[Peterson, 86] J. L. Peterson and A. Silberschatz, *Operating Systems Concepts*, (New York, Addson-Wesley, 1986).

[Raymond, 89] K. Raymond, A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems 7*, 1989, 61 - 77.

[Ricart, 81] G. Ricart and A. K. Agrawala, An optimal algorithm for mutual exclusion in computer networks, *Communications of the ACM 24*, (1981) 9 - 27.

[Ricart, 83] G. Ricart and A. K. Agrawala, Author's response to 'On mutual exclusion in computer networks by Carvalho and Roucairol', *Communications of the ACM 26*. 1983, 147 - 148.

[Suzuki, 85] I. Suzuki and T. Kasami, A distributed mutual exclusion algorithm. *ACM Transactions on Computer Systems 3*, 1985, 344 - 349.

[Velazquez, 93] M. G. Velazquez, A survey of distributed mutual exclusion algorithms, Technical Report CS-93-116, September 1993.

Appendix

The algorithm is given as a series of procedures. We have given, for the purpose of simplification, the algorithm for local nodes and the algorithm for link-nodes.

***Algorithm* DISTRIBUTED_MUTEX:**

A1. Initialize node (i, k), i ∈ { 0, 1, …, √n –1}, k ∈ { 1, 2, …, √n }

```
Procedure initialize()
local_request_collector := (i, 0)
requested := 0
mutex  :=  0
LRQ := N
if (i, k)  = (0, 0) then
   token_here := 1
else
   token_here := 0
 fi
if (i, k)  = (i, 0)   /*link-node*/ then
    GRQ := N
    global_request_collector  := (0, 0)   fi
```

A2. All nodes execute the following procedure

```
Procedure request_CS( ) /*node (i, k )  requests CS*/
requested := 1
Send (REQUEST, (i, k ) , N) to LRC
          /*wait for token*/
```

A3. Algorithm for local node (i, k), i ∈ { 0, 1, …, √n –1}, k ∈ { 1, 2, …, √n }

```
/*receives (REQUEST, (i, j), N) from node (i, j)*/
Procedure receive_REQUEST()
if local_request_collector   =  (i, k) then
          enqueue (LRQ, (i, j))
               if token_here = 1 and mutex = 0 then

                    copy ( LRQ , token_Q , N)
                local_request_collector :=   token_Q.rear
               broadcast(LR_COLLECTOR( )
               send (TOKEN, (i, j), token_Q) to token_Q.front    fi
else       /* node is no longer the LRC*/
      send (REQUEST, (i, j), N) to LRC    fi

/* receive (LR_COLLECTOR, token_Q.rear, N) */
Procedure receive_LR_COLLECTOR()
        local_request_collector := token_Q.rear
```

```
 /* receive (TOKEN, (i, j), token_Q)*/
Procedure Receive_ TOKEN()
token_here := 1
if requested = 1 then
 do      requested := 0
         delete (token_Q,  Token_Q.front)  /*remove self from queue*/
          mutex: = 1  od  fi
/* critical section operation is performed here */

Procedure exit_CS    /*  critical section completed */
  mutex: = 0
   if (i, k) = LRC  then
           if LRQ ≠ N  then
                copy (LRQ, token_Q, N)
                 token_here := 0
                 local_request_collector  = token_Q.rear
                  broadcastLR_COLLECTOR( )
                 send (TOKEN, (i, k), token_Q) to token_Q.front  fi
    else    /*(i, k) is not the request collector*/
        send (TOKEN, (i, j), token_Q) to token_Q.front   fi
```

A4. Algorithm for link-node (i, 0), i ∈ { 0, 1, … , √n –1}

```
Procedure receive_REQUEST()
 /*node (i, 0) receives ( REQUEST , (j, k), N)*/
if LRC = (i, 0) and GRC = (i, 0) then
           if j = i     then    /*request is from a local node  */
                 enqueue (LRQ, (i, k))
                  if token here = 1 and mutex = 0 then
                 grant_token() fi fi
           if j ≠ i, then  /*request is from a link node */
              enqueue (GRQ, (j, 0))
              enqueue (LRQ, marker)
             if token_here = 1 then
              grant_token()
                      fi fi fi
if (i, 0) = LRC and (i, 0) ≠ GRC then
         if j = i then   /*request is from a local node */
             enqueue (LRQ, (i, k))
              if token_here = 1 and mutex = 0 then
                 grant_token()
             else
                  send(REQUEST, (i, 0), N ) to GRC  fi  fi
           if (j ≠ i), then  /*request is from a global node*/
               send (REQUEST, (i, k), N) to GRC
           fi  fi
if (i, 0) ≠ LRC and (i, 0) = GRC then
```

```
      if j = i, then  /*request is from a local node*/
            send (REQUEST, (i, k), N) to LRC
               fi
if  j ≠i, then  /*request is from a global node */
            enqueue (GRQ, (j, 0))
            enqueue (LRQ, marker)
             send (REQUEST, (i, k), N) to LRC
            fi  fi

Procedure exit_cs_linknode()        /* Exit CS*/
    Token_here = 0
    Grant_token()

Procedure receive_token_linknode()
/* node (i ,0) = tokenQ.front receives token */
token_here := 1
/*delete self from token_Q*/
delete (token_Q, tokenQ.front )
if (i, 0)   ≠ GRC
   enqueue (LRQ, marker)
   if  token_Q = N then
        grant_token()  fi fi
  if  token_Q ≠ N then
      if token_Q.front =  (j, 0)
     then  /* token is from a  link-node  */
       copy (token_Q, GRQ , N)
             grant_token()    fi     fi
       if  token_Q.front = (i, k) then
    /*token_is from a local node  */
             if  requested = 0 then
                       copy (token_Q, LRQ, N)      /* copy token queue to LRQ*/
                       grant_token()  fi fi

Procedure grant_token()
/*node (i, 0) possesses the token*/
 if  LRQ ≠ N then
    if  node_id = LRQ.front and requested = 1 then
               mutex := 1
               requested  := 0
            /*do critical section at this point */
            fi
    if  LRQ.front  = marker  then
       if  GRQ ≠ N
            copy (GRQ, token_Q, N)
            delete(LRQ, marker)
             if  LRQ ≠ N  then
                  if node_id = GRC  then
```

```
                    enqueue(token_Q, (i, 0))
                else (node_id ≠ GRC)
                    send (REQUEST, (i, j), N) to GRC fi
            fi fi
if  LRQ = N  then
/* broadcast GRC information */
        broadcast GR_COLLECTOR()
           send(TOKEN, tokenQ.front, tokenQ)
                    fi  fi
   if  LRC ◊ (i, 0)  then
            LRC : = (i, 0)
      /*broadcast LRC information*/
 broadcast LR_COLLECTOR()
if  LRQ.front ≠ marker or LRQ.front ≠ (i, 0) then
              copy (LRQ, token_Q,  index)
           /* index = marker or (i, 0)  or = N   */
              if  index = marker then
                    enqueue (token_Q, (i, 0) ) fi
              if index = node_id  then
                    enqueue (token_Q, (i, 0) )
                    delete(LRQ, (i, 0) )  fi
              if  index = N  then
                    LRC := tokenQ.rear
           braodcastLR_COLLECTOR()  /*broadcast  LRC information */
              fi  fi  fi
           send(TOKEN, tokenQ.front, tokenQ)

 /*Receive (GR_COLLECTOR, (k, 0), N) message )*/
Procedure receive_GR_COLLECTOR()
                global_request_collector :=  ( k, 0)

 /*Receive (LR_ COLLECTOR, (i, k), N) message */
Procedure receive_LR_COLLECTOR()
                  local_request_collector := (i, k)

broadcast GR_COLLECTOR( )
     send (GR_COLLECTOR, token_Q.rear, N) to
                              (k, 0),  k ∈ {0, 1, …, √n –1},  k ≠ i

broadcast LR_COLLECTOR( )
     send (LR_COLLECTOR, token_Q.rear, N) to (i, j),    j ∈ {1, 2,…, √n} fi
```