A DISTRIBUTED K-MUTUAL EXCLUSION ALGORITHM

A Thesis

by

SHAILAJA GURUPAD BULGANNAWAR

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 1994

Major Subject: Electrical Engineering

A DISTRIBUTED K-MUTUAL EXCLUSION ALGORITHM

A Thesis

by

SHAILAJA GURUPAD BULGANNAWAR

Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Approved as to style and content by:

D. Ross (Co-Chair of Committee)	N. H. Vaidya (Co-Chair of Committee)
Mi Lu (Member)	A. Datta (Member)
A. D. Patton (Head of Department)	

August 1994

Major Subject: Electrical Engineering

ABSTRACT

A Distributed K-Mutual Exclusion Algorithm. (August 1994)

Shailaja Gurupad Bulgannawar, B.E., University Visvesvaraya College of Engineering

Co–Chairs of Advisory Committee: Dr. D. Ross
Dr. N. H. Vaidya

This thesis presents a new token-based K-mutual exclusion algorithm for distributed systems. The proposed algorithm uses K tokens to achieve K-mutual exclusion. The system of N nodes is organized as a logical forest, with the node possessing the token forming the root of a tree. For a system with K tokens, there are K such forests. The token with its associated token-queue and tag is passed among the nodes. Only nodes with a token are allowed to enter the Critical Section. Nodes that do not possess a token, send a request for the token along the edges of the forest. On receiving the token they enter the critical section.

We show that the proposed algorithm achieves K-mutual exclusion and is free from deadlock and starvation. We analyze the algorithm at low and high rates of request for critical section execution. The simulation and implementation results show that the average number of messages and the average time to enter the critical section for our algorithm is small compared with other distributed K-mutual exclusion algorithms.

To my parents

ACKNOWLEDGMENTS

I would like to acknowledge the guidance given to me, time and again, by Dr. N. H. Vaidya and Dr. D. Ross. I would like to acknowledge the support given to me by my committee members Dr. A. Datta and Dr. Mi Lu.

TABLE OF CONTENTS

CHAPTER		Page
I	INTRODUCTION	1
	A. Motivation to Choose a Distributed System	2
	B. Topology of a Distributed System	2
	C. Mutual Exclusion	2
	D. Communication in a Distributed System	3
II	REVIEW OF DISTRIBUTED 1-MUTUAL EXCLUSION	5
	A. Permission Based Algorithms	5
	B. Token Based Algorithm	8
	1. Tree based logical structure	9
III	REVIEW OF DISTRIBUTED K-MUTUAL EXCLUSION	13
	A. Permission Based Algorithms	13
	1. Raymond's algorithm	13
	B. Token Based Algorithms	14
	1. Srimani and Reddy's algorithm	15
	2. Makki et al.'s algorithm	16
IV	PROPOSED K-MUTUAL EXCLUSION ALGORITHM	18
	A. Proposed Approach for Distributed K-Mutual Exclusion	18
	1. Data structures maintained at the node	20
	2. Data structure associated with the token	20
	3. Types of messages	21
	4. Proposed algorithm	22
	B. Proof of Correctness of the Algorithm	32
	1. Mutual exclusion	32
	2. Deadlock and starvation free	32
	C. Performance	37
	1. Heuristics for choice of token	38
V	SIMULATION RESULTS	39
	A. Simulation Model	39

CHAPTER Page

B. Simulation Results 40
1. Comparison of simulation results 40

VI IMPLEMENTATION 64

A. Details of Implementation 64
B. Results 65

VII CONCLUSIONS AND FUTURE WORK 70

REFERENCES 72

APPENDIX A 75

VITA 79

LIST OF TABLES

TABLE Page

I Standard deviations for the proposed algorithm 68

II Standard deviations for Raymond's algorithm 69

LIST OF FIGURES

FIGURE		Page
1	An example for Trehel and Naimi's algorithm	11
2	Token trees at initialization	19
3	Token-queue with its associated tag	21
4	A typical token tree	27
5	Forest structure for token 2 before and after a request from node 4	29
6	Token-queue of token 2 at node 3	30
7	Forest structure for token 1	34
8	Case of deadlock for a request for token t	36
9	General case of deadlock for a request for token t	37
10	Average time to enter the critical section for $T_r = 0.1$, $T_s = 0.1$ and $T_t = 0.8$	41
11	Makki et al.'s algorithm: token-queue for $K = 2$	43
12	Average time to enter the critical section for $T_r = 0$, $T_s = 0$ and $T_t = 1.0$	44
13	Average time to enter the critical section for $T_r = 0.05$, $T_s = 0.05$ and $T_t = 0.9$	45
14	Average # of messages per critical section entry for $T_r = 0.1$, $T_s = 0.1$ and $T_t = 0.8$	46
15	Average # of messages per critical section entry for $T_r = 0.05$, $T_s = 0.05$ and $T_t = 0.9$	48
16	Average # of messages per critical section entry for $T_r = 0$, $T_s = 0$ and $T_t = 1.0$	49

FIGURE Page

17 Average information (in words) per message for $T_r = 0.1$, $T_s = 0.1$ and $T_t = 0.8$ 50

18 Average information (in words) per message for $T_r = 0.05$, $T_s = 0.05$ and $T_t = 0.9$ 51

19 Average information (in words) per message for $T_r = 0$, $T_s = 0$ and $T_t = 1.0$ 52

20 Average time to enter the critical section for $T_r = 0.1$, $T_s = 0.1$ and $T_t = 0.8$ and $E = 1$ 53

21 Average time to enter the critical section for $T_r = 0.05$, $T_s = 0.05$ and $T_t = 0.9$ and $E = 1$ 54

22 Average time to enter the critical section for $T_r = 0$, $T_s = 0$ and $T_t = 1.0$ and $E = 1$ 55

23 Average # of messages per critical section entry for $T_r = 0.1$, $T_s = 0.1$ and $T_t = 0.8$ and $E = 1$ 56

24 Average # of messages per critical section entry for $T_r = 0.05$, $T_s = 0.05$ and $T_t = 0.9$ and $E = 1$ 57

25 Average # of messages per critical section entry for $T_r = 0$, $T_s = 0$ and $T_t = 1.0$ and $E = 1$ 58

26 Partitioning scheme: average time to enter the critical section for $T_r = 0.1$, $T_s = 0.1$ and $T_t = 0.8$ 60

27 Partitioning scheme: average time to enter the critical section for $T_r = 0.05$, $T_s = 0.05$ and $T_t = 0.9$ 61

28 Partitioning scheme: average time to enter the critical section for $T_r = 0$, $T_s = 0$ and $T_t = 1.0$ 62

29 Partitioning scheme: average # of messages per critical section entry for $T_r = 0.1$, $T_s = 0.1$ and $T_t = 0.8$ 63

30 Implementation: average # of messages per critical section entry . . 66

FIGURE Page

31 Implementation: average time to enter the critical section 67

CHAPTER I

INTRODUCTION

There has been a trend in computer systems to distribute computation among several processors. There are two schemes for building such systems. In a *tightly coupled* system processors share memory. In these multiprocessor systems, communication usually takes place through a shared memory. In a *loosely coupled* system processors do not share memory or a clock. Every processor has its own local memory. The processors communicate via messages through various communication networks such as ethernet and FDDI. These systems are referred to as *distributed* systems. The processors in a distributed system may vary in size and function. These processors are referred to by different names such as *sites, nodes*, etc.

A distributed program is a concurrent (or parallel) program in which processes communicate by message passing. There are two models of systems that communicate by message passing, namely, Client Server model and Peer to Peer model. In a Client Server model, clients make requests that trigger responses from server processes which are perpetually waiting for client requests. Thus we essentially have a master-slave relationship between the client and server and these roles, once assigned, are fixed. In a Peer to Peer model, processes that are on an equal level (in the sense that no master-slave relationship exists between them), cooperate to perform some computation or to provide a service.

The journal model is *IEEE Transactions on Automatic Control.*

A. Motivation to Choose a Distributed System

Distributed systems allow efficient and convenient resource sharing between the sites that are loosely coupled and interconnected by a communication network. Thus sharing of files at remote sites, printing files at remote sites, processing information in a distributed database, using remote hardware devices (such as array processors) is made possible. Load sharing, where a computation is partitioned and the subcomputations are run concurrently on various sites, is also possible.

B. Topology of a Distributed System

The sites in the system can be physically connected in a number of ways. In a *fully connected* network, each site is directly linked with all other sites in the system. A system is *partitioned* if it has been split into two (or more) such subsystems that lack any connection between them.

In a *partially connected* network, direct link exists between some, and not all pairs of sites. However, *logical* channel can be established between any pair of sites. In a *hierarchical* network, the sites are organized as a tree. The other types of networks are *star* network and *ring* network.

C. Mutual Exclusion

As distributed systems are becoming popular, problems such as synchronization of distributed concurrency, distribution of work among processors, maintaining the integrity of a dynamic, distributed file system etc. are being studied and various algorithms are being put forward.

In a distributed system where sites communicate via message passing, it often becomes necessary to guarantee the integrity of shared resources by restricting the

use of such resources to a single user or to a small number of users at a time. *Mutual exclusion* is a process whereby concurrent accesses to a shared resource by several uncoordinated user-requests are serialized to secure the integrity of the shared resource. The site that has acquired an access to a resource and is performing the relevant operations, is said to be in the *critical section* (CS). Hence, when a shared resource is accessed, the site (node) executes in the critical section.

In a distributed system, mutual exclusion algorithms are based on communication of messages between processes. The processes exchange information among themselves by communicating it explicitly via messages.

Most of the mutual exclusion algorithms reported allow at most one node to execute in the critical section. The problem of allowing at most one node in the critical section is called the 1-*mutual exclusion* problem. In order to utilize the resources better, it may be possible, in some cases, to allow a bounded number of nodes (≥ 1) to execute simultaneously in the critical section. The problem of allowing at most K nodes, for a fixed K ($K \geq 1$), to simultaneously execute in the critical section is called the K-*mutual exclusion* problem.

D. Communication in a Distributed System

Communication is required to allow arbitrary or related processes to exchange data or synchronize execution. In UNIX there are several features provided to allow processes to communicate with each other. The common schemes are pipes, named pipes, signals, shared memory, semaphores, message passing and sockets and streams. The main disadvantage of pipes is that processes cannot communicate without common ancestors. Named pipes do allow this facility, though they cannot generally be used to communicate across networks. To communicate across networks, sockets and streams

provide good mechanisms for IPC. Arbitrary processes can also communicate with each other by sending the kill signal. The 4.2BSD sockets interface is a popular transport-level interface. The socket interface makes it easy for UNIX programmers to develop powerful network applications. On most networked UNIX systems, OSI layers 1 to 4 are implemented through a combination of 802.3 Ethernet, TCP/IP, and a transport level programming interface that is usually implemented as a library of system calls. Sockets and streams provide this library of calls for implementing the transport layer and it is desirable for UNIX networking applications to use either of these for achieving interconnectivity in heterogeneous networked environments. The client server model finds application in many areas of distributed systems.

The thesis is organized as follows. Chapter II summarizes previous work on distributed 1-mutual exclusion. Chapter III reviews previous work on distributed K-mutual exclusion. Chapter IV presents the proposed K-mutual exclusion algorithm. It discusses a few salient features of the proposed algorithm and proves its correctness. Chapter V presents the simulation results. The simulation model is presented and the results of the proposed algorithm are compared with that of the other distributed K-mutual exclusion algorithms. Chapter VI discusses an implementation of the proposed algorithm and experimental results. The thesis concludes with Chapter VII.

CHAPTER II

REVIEW OF DISTRIBUTED 1-MUTUAL EXCLUSION

Mutual exclusion algorithms that allow at most one node into the critical section are referred to as 1-mutual exclusion algorithms. Mutual exclusion in distributed systems can be classified into two categories: *permission based* algorithms and *token based* algorithms. In the permission based algorithms, the nodes (sites) enter the *critical section* (CS) only after receiving permission from a subset of nodes in the system. In the token based algorithms, the nodes enter the critical section after receiving a *token* (a unique privilege).

A. Permission Based Algorithms

Lamport proposed one of the first distributed mutual exclusion algorithms in 1978 [1]. The algorithm imposes no a priori order on the way the privilege moves from one process to another. The system events are ordered by dating of the system events, using a message timestamping process. Every message is given a number by its transmitter, called its timestamp or logical date. The queue that records the critical section requests and release messages is distributed over all the nodes (sites) in the system. Hence any request to critical section and release from critical section is broadcast to all the nodes in the system. For a node to take the decision to enter the critical section based only on its own queue, it needs to receive a reply message from each of the other nodes. This is made possible by *acknowledgment* type of messages that each node sends in reply to a request message. The algorithm is, therefore, permission based requiring $3 * (N - 1)$ messages per critical section entry, where N is the number of nodes in the system. When requesting entry into critical section, a node sends $(N - 1)$ *request* messages; on receiving the request, each of the $(N - 1)$

nodes send the acknowledgment message and on exiting the critical section, a node sends $(N-1)$ *release* messages.

In 1981, Ricart and Agrawala proposed another permission based algorithm that reduced the number of messages required per critical section entry to $2*(N-1)$ [2]. The improvement is achieved by eliminating the *acknowledgment* messages used in Lamport's algorithm. The only messages are, $(N-1)$ request messages and $(N-1)$ favorable replies. The algorithm assumes the presence of an error free transport network in which transmit time of the messages may vary and the messages may overtake each other. When a node i wishes to enter the critical section, it generates a timestamp and broadcasts the request message to all the other nodes along with the timestamp. When a node j receives the request, it either replies favorably, or defers the reply if it is in the critical section. On leaving the critical section, the node j sends any deferred reply to the requesting nodes. When a node receives replies from all the other nodes in the system, it enters the critical section. If a node that receives a request message, also wants to enter the critical section, then it compares the timestamp of its request with that of the request that it has received. If its value is less than that of the received request, it defers the reply message. Otherwise reply is sent immediately.

In the same year, Carvalho and Roucairol proposed an improved version of Ricart and Agrawala's algorithm [1] and were able to reduce the number of messages to be between 0 and $2*(N-1)$. The fundamental idea on which the improvement is based is that, when a node receives the permission from $(N-1)$ other nodes, it can enter the critical section as many times as it needs to, as long as no other node wishes to enter the critical section. Hence, once a node i receives a favorable reply from a node j, the implicit permission from node j is valid until node j sends a request to node i. Hence, between the moment that node j sends the authorization to node i and the

moment that it sends a request message to node i, node i can enter the critical section any number of times. Hence the number of messages can vary from 0 to $2*(N-1)$.

In 1985, Maekawa proposed a permission based algorithm in which the number of messages required is $O(\sqrt{N})$ [3]. Here *coteries* are constructed based on finite projective planes. A node needs to obtain permission from all the other nodes in its coterie.

Sanders generalized the permission based algorithms by defining an information structure that each node has to maintain [4]. The information structure describes which nodes maintain the state information about the other nodes and from which nodes information must be requested before entering the critical section. The information structure consists of an *inform set* and a *request set*. Every node maintains these sets. A node sends a message to every node in its inform set whenever it changes its state. The state of a node can be either *in-cs*, *not-in-cs* or *waiting*. Before entering the critical section, every node must send the request message to and receive permission from every node in its request set. The information structure can be *static*, in which case, the entire information structure can be known to all the nodes. *Dynamic* information structures change during the evolution of the algorithm and the global information structure may not be known to all the nodes.

Singhal proposed a *dynamic information structure* in which the set of nodes from which a particular node must request permission to enter the critical section evolves as the nodes learn more about the state of the system [5]. The information set consists of the inform set and request sets as proposed by Sanders. This algorithm requires $2*(N-1)$ messages per critical section entry in the worst case.

B. Token Based Algorithm

In the token based algorithms, a token is passed around the nodes and the node with the token can enter the critical section. In 1982 Suzuki and Kasami proposed the first token based algorithm [6]. In 1983, Ricart and Agrawala proposed a token based algorithm [7] as an improvement to their permission based algorithm presented in 1981 [2]. This algorithm was essentially the same as Suzuki and Kasami's token based algorithm [6]. In both the algorithms, when a node i needs to enter the critical section, it sends a request for the token to all the other nodes along with a timestamp. The token stores the timestamps of the last visit it made to each of the nodes. So if a node j has the token, then when it exits the critical section, it checks for the first process whose last request has a timestamp greater than the timestamp stored on the token during its last visit to the requesting node and sends the token to that node. The number of messages required by this algorithm is N where $(N-1)$ messages are needed to broadcast the request and one message to receive the token.

Singhal proposed a heuristically aided algorithm that uses state information to guess the location of the token more accurately [8]. In this algorithm, each site maintains two vectors, one called *state vector*, which stores the latest known state of the sites, which can be either requesting or not-requesting and the other called *sequence number vector*, which stores the highest known sequence number for each site. The token is tagged with similar arrays. All the sites which have a state of *requesting* in the state vector of a node form a probable set to which the node will send its request messages. The information about the state of the system is obtained through the messages that a node receives and the information received with a token. The maximum number of messages per critical section entry required by this algorithm is N, where N is the number of nodes in the system.

Mizuno, Neilsen and Rao proposed a token based algorithm that uses *quorum agreements* [9]. Quorums have data structures similar to coteries of Maekawa's $\sqrt{N}$ algorithm [3]. The quorums consist of *request set* and *acquired set*. To obtain the token, a node sends a request to all the nodes in its request set excluding itself and on obtaining the token, it sends an *acquired* message to all the nodes in its acquired set excluding itself. The number of messages depends on the type of quorum used.

In 1993, Makki, Pissinou and Yesha proposed a token based algorithm using timestamps, theory of finite projective planes and token queue [10]. Here the sites are grouped into sets using the theory of finite projective planes. In this algorithm, when a site A, in a particular set S needs to enter the critical section, it sends request messages to all other sites in set S. If the token is present at site B in another set T, then one of the sites in set S which is also in set T, will send an inform message to site B. Site B then sends a message to all the sites in its set T informing that the token is being sent to another site and it waits for their replies. On receiving the replies from the other sites in its set T, site B sends the token to the requesting site A. Upon receiving the token, site A sends a message to all the sites in its set to inform them that it has the token. The algorithm requires zero messages in the best case and $4\sqrt{N} - 2$ messages in the worst case, per critical section entry.

1. Tree based logical structure

Some token based algorithms use tree based logical structure for the nodes. All the nodes other than the root node (a sink node) are on a path to the root node. The logical structure determines the path along which a request message travels. The structure is defined by a pointer maintained by each node. The logical structure can be of two types, either *dynamic* or *static*. The neighbor of node i can be defined as a set of nodes j such that there exists an edge from node i to node j or vice versa. In

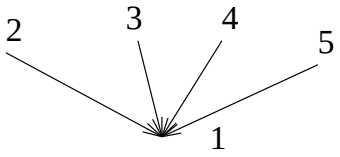
the dynamic structure the neighbors of a node change from time to time. Whereas, in a static structure the neighbors of a node are always fixed.

Trehel and Naimi proposed an algorithm based on a dynamic logical tree structure [11]. A similar algorithm was later proposed by Bernabeu-Auban and Ahamad [12]. These algorithms use *path reversal* to reduce the height of the tree structure. In a path reversal, when a request from node i travels along the path to the root node, all nodes in the path except node i make node i as their new parent. The upper bound on the amortized cost of path reversal is discussed in [13]. The average number of messages required per critical section entry is $O(\log N)$. The number of messages depend on the height of the rooted tree. An example is illustated in Figure 1, where solid lines represent the tree and dotted lines represent an implicit queue structure maintained by the algorithm.

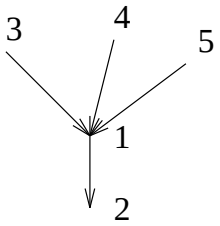
Raymond proposed an algorithm based on a static logical structure [14]. The algorithm uses an unrooted tree structure. Edge reversals are performed when a request travels towards the root node. As the request from node x travels to the root node, if node i along the path passes the request to node j then node j points to node i after passing the request to the next node along the path. The average number of messages required is $O(\log N)$. An improvement to this algorithm was proposed by Neilsen and Mizuno in their Dag-Based algorithm [15]. The algorithm uses a static, directed acyclic graph (dag) structure. The number of messages required for mutual exclusion depends on the logical topology imposed on the nodes. It reduces the number of messages required to half the number required in Raymond's algorithm. Using the best topology, the *star topology*, the number of messages required is reduced to three.

Woo proposed an algorithm using *Huffman trees* as a logical structure for dynamic mutual exclusion [16]. The algorithm uses a dynamic tree where the nodes

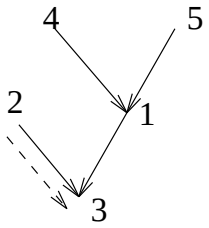
* Initial state of the distributed system. Site 1 has priviledge.



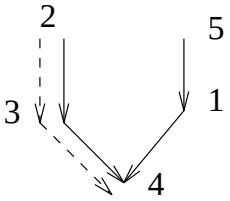
* Site 2 invokes CS. It sends a request to site 1. Site 2 becomes the root. 2 has the priviledge and enters the CS.



* Site 3 invokes the CS. It sends request to 1 which transmits to 2. Site 2 is in CS. 3 waits . (shown by dotted line).



* Site 4 requests the CS. It sends request to 1 which transmits to 3. Site 2 is in CS; 3 and 4 wait.



* Site 2 releases the CS. Gives priviledge to 3. Site 2 requests again. It sends request to 3 which transmit to 4. Site 3 is in the CS. 4 and 2 wait.

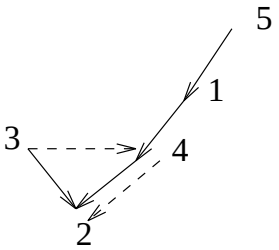


Fig. 1. An example for Trehel and Naimi's algorithm

with higher probabilities of entering the critical section are placed near the root of the tree. The number of messages required for each entry to the critical section is at most $O(H)$ where H is the height of the Huffman tree.

Exploiting logical tree structures for efficient management of replicated data and services is discussed in [17]. All the above algorithms allowed at most one node to execute in the critical section. It is possible to extend some of these algorithms to enable K nodes to be in the critical section simultaneously.

CHAPTER III

REVIEW OF DISTRIBUTED K-MUTUAL EXCLUSION

When it becomes necessary to guarantee the integrity of shared resources by restricting the use of such resources to a small number of users at a time we have the problem of K-mutual exclusion. Distributed K-mutual exclusion could be achieved using permission based algorithms or token based algorithms. In token based algorithms, a single token could be used or K-tokens could be used to achieve K-mutual exclusion. Fault tolerant algorithms for K-mutual exclusion have also been proposed e.g. [18].

A. Permission Based Algorithms

In these algorithms, the node that needs to enter the critical section (CS) sends request messages to some nodes in the system and waits to receive permission from these nodes.

1. Raymond's algorithm

In 1989 Raymond proposed a permission based algorithm to allow K nodes to be in the critical section simultaneously [19], as an extension of Ricart and Agrawala's algorithm [2]. This algorithm uses two types of messages, one is the *request* message and the other is the *reply* message. When a node needs to enter the critical section, it sends request messages to $(N-1)$ nodes and waits for $(N-K)$ reply messages. Hence, the lower bound for this algorithm is $(2N-K-1)$ messages per critical section entry where N is the number of nodes in the system and K is the number of allowed simultaneous critical section entries. However, each request message could eventually generate a reply message. Hence, the upper bound for this algorithm is $2*(N-1)$. The node could enter the critical section once it receives $(N-K)$ reply messages and

the remaining $(K-1)$ reply messages arrive when the node is in the critical section or after the node has exited the critical section. Therefore, to prevent mistaking the reply messages of a previous attempt to that for the current attempt, a reply count array is maintained at each node, for all the nodes in the system. If the reply count for a node is zero, it implies that that node has sent all the reply messages to the requesting node.

When a node receives a request from another node, it defers the reply to the requesting node if it is in the critical section or has requested entry into critical section itself and its sequence number is smaller than the sequence number of the request received from another node. If both the sequence numbers are equal, then if the receiving node's identifier is smaller, the request is deferred. Otherwise, the node will send a reply to the requesting node.

If the reply messages have been deferred due to a node being in the critical section for a long time compared to other requesting nodes, then the count of deferred reply messages is sent in a single reply message. Thus, when a node exits the critical section, the deferred replies for each node are sent collectively as one physical message. This would result in fewer messages occasionally .

B. Token Based Algorithms

In token based algorithms, when a node needs to enter the critical section, it sends a request to the other nodes requesting for a token (also called a privilege message). Upon getting a token, critical section can be entered.

1. Srimani and Reddy's algorithm

Srimani and Reddy proposed an algorithm for multiple entries to a critical section [20] that improved upon the algorithm presented by Raymond [19] and reduced the number of messages required per critical section entry to half of that required by Raymond's algorithm. Their algorithm is based on the token based algorithm proposed by Suzuki and Kasami [1]. The algorithm uses K tokens (privilege messages) to allow K simultaneous entries into the critical section. For any node to be in the critical section, it needs to possess a token. Hence, the lower bound on the messages is zero. If the requesting node does not have the token, it sends $(N-1)$ request messages and in the worst case when no other node is requesting and each token is residing at a different node, all the K tokens are sent to the requesting node resulting in a total of $(N+K-1)$ messages (the upper bound).

The algorithm uses two types of messages, *request* messages and *privilege* messages. Each of the K privilege messages (tokens) has a token queue Q and an array $LN[j]$ of size N (where N is the number of nodes in the system), that stores the last serviced request number of node j. Each node maintains an array $RN[j]$ of size N, that records the arrival of a request from node j, an array PLN of size N, that collects the most recent information about the sequence numbers of the requests already serviced and a *privilege count* that keeps count of the number of privilege messages present at the node. The PLN array is updated using the LN arrays of the privilege messages received by the node. The PLN array is used to update the Q in any privilege message so that there is no unnecessary transmission of additional privilege message to a requesting node after it has already received the privilege.

When a node wants to enter the critical section, if it has privilege count greater than or equal to 1, it can do so immediately. Otherwise, it sends a request message

to $(N-1)$ nodes along with the sequence number of the request. It then waits for the privilege count to become greater than or equal to 1 and when that happens it enters the critical section. On exiting the critical section, the node updates the information content in the privilege message.

When a node receives a request message, it updates $RN[j]$ to record the arrival of this request and $PLN[j]$ is made equal to $(RN[j]-1)$ to record the most recent information about the sequence number already serviced for node j. If this node has a spare privilege message, then it updates its LN array and Q before sending it to node j.

When a node gets a privilege message, it checks if it is waiting to enter the critical section. If the node is waiting to enter the critical section, it just increments the privilege count and exits. Otherwise, it updates the privilege messages LN and Q before sending it to the next node on the token queue Q. If Q is empty, it stores the privilege message.

2. Makki et al.'s algorithm

Another token based K-mutual exclusion algorithm was proposed by Makki et al. [21]. This algorithm uses only one token. The single token has a queue attached to it which stores the requests from the nodes and a token semaphore consisting of the number of sites in the critical section and the maximum number of sites allowed in the critical section. As the token is passed among the requesting nodes, a node enters the critical section when it either receives the token and a non-zero token semaphore or it receives the token with a zero semaphore (indicating that K other nodes are in critical section entries) and later receives a *release* message from another node. The token queue has a "*good site*" which is the last node on the token queue. This node receives the token requests from nodes that are not on the token queue and stores

them in its local queue.

There are four types of messages used in this algorithm: the *token* message which includes the token queue and the token semaphore, the *request* message, the *good site* update message and the *release* message.

A release message is sent by each node that exits the critical section, to the k^{th} node on the token queue after it has removed itself. This is because, if only K nodes are allowed in the critical section at one time, then the k^{th} node from the current node will receive the token with a zero semaphore and hence would need the release message to enter the critical section. If, however, the good site is before the k^{th} node on the token queue, then the release messages are sent to the good site. This causes the good site to receive a release message of the K previous sites, which is used to increment the semaphore value back to its maximum of K.

When the token is eventually received by the good site, it executes its critical section and and then appends its local queue on to the token queue. At this point a new good site is chosen which is the last node on the token queue. The old good site sends update messages (about the new good site) to all the nodes that are not on the token queue. The old good site then waits for a fixed time period, long enough to collect all the remaining late token requests and the release messages, before sending the token to the first node on the token queue.

On heavy load, only three messages per critical section entry are required, consisting of the information of the goodsite, the request for the token sent to the good site, and the token sent to the requesting node. Under light load, number of messages in the order of N are required per critical section entry.

A disadvantage of this algorithm is that it is useful only when the upper bound on message transmission time is known.

CHAPTER IV

PROPOSED K-MUTUAL EXCLUSION ALGORITHM

We propose a new token based distributed K-mutual exclusion algorithm that uses K tokens allowing K simultaneous entries in the critical section (CS). This algorithm shows considerable improvements in the number of messages and time to enter the critical section compared to the other K-mutual exclusion algorithms proposed by Raymond [19], Srimani and Reddy [20] and Makki et al. [21].

A. Proposed Approach for Distributed K-Mutual Exclusion

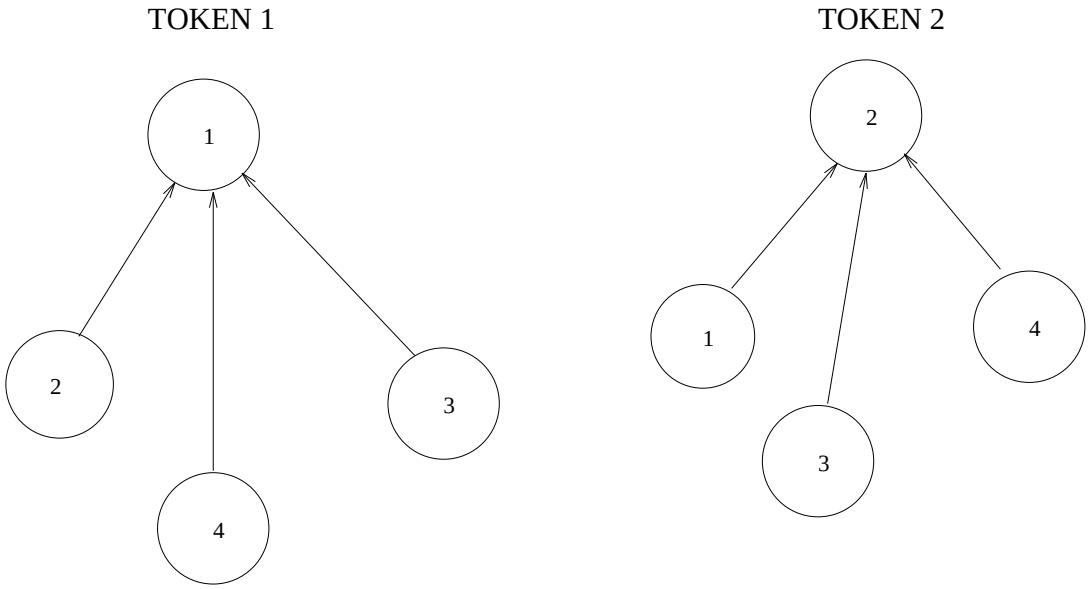
The system is assumed to consist of N nodes numbered 1 to N. Each node can have at most one outstanding request to enter the critical section at any given time. All the nodes are assumed to be fully connected. The nodes and the network are assumed to be reliable. Also, the network is assumed to deliver messages in FIFO[1] order on each channel.

In our algorithm, the K tokens are initially resident at a predefined set of K nodes such that, a token t is possessed by node t where $1 \leq t \leq K$. Each node maintains a *pointer* array where, initially pointer[i] contains i because initially node i has token i. This is illustrated in Figure 2.

The pointer array for token i defines a forest structure for token i, with a node holding the token forming a root of a tree in the forest. Hence, for a system of K tokens, there exist K tree structures in the system.

The approach for K-mutual exclusion is derived by improving on the 1-mutual exclusion algorithm by Trehel and Naimi [11]. For $K = 1$, our approach is expected

[1]FIFO denotes First-In-First-Out.



node 1: Pointer[1] = 1
node 2: Pointer[1] = 1
node 3: Pointer[1] = 1
node 4: Pointer[1] = 1

node 1: Pointer[2] = 2
node 2: Pointer[2] = 2
node 3: Pointer[2] = 2
node 4: Pointer[2] = 2

Fig. 2. Token trees at initialization

to perform better than Trehel and Naimi's by reducing the number of messages.

1. Data structures maintained at the node

The data structures maintained at the node are as follows:

- *token*: *boolean*; TRUE indicates presence of token at a node and FALSE indicates absence of token at a node.
- *token_id*: *integer*; indicates the identifier of the token.
- *holding*: *boolean*; TRUE indicates the node is in the CS and FALSE indicates node is not in CS.
- *waiting_for_token*: *integer*; indicates the identifier of the token that the node has requested and is waiting for.
- *pointer*: *array[1..K] of integer*; pointer[i] indicates the path for token i.
- *node-queue*: *FIFO queue*; stores the token requests that arrive at a node and can be empty sometimes. If node i is waiting for token t, then any request for token t from other nodes is stored in the node-queue of node i. The node-queue contains identifiers of the nodes that send a message requesting token t. If node i is holding token t, then any requests for any token are stored in the node-queue of node i.

2. Data structure associated with the token

Every token is associated with a data structure that is sent along with the token, as the token is passed from one requesting node to the next. The data structure is as follows:

- *token-queue*: *FIFO queue*; contains the identifiers of the nodes to which the token must be forwarded in a FIFO order.
- *token_changed_pointer*: *FIFO queue*; store a "tag" associated with each node identifier in the token-queue. Some of the tags may be NULL (–).

Figure 3 shows a typical token-queue and its associated tag.

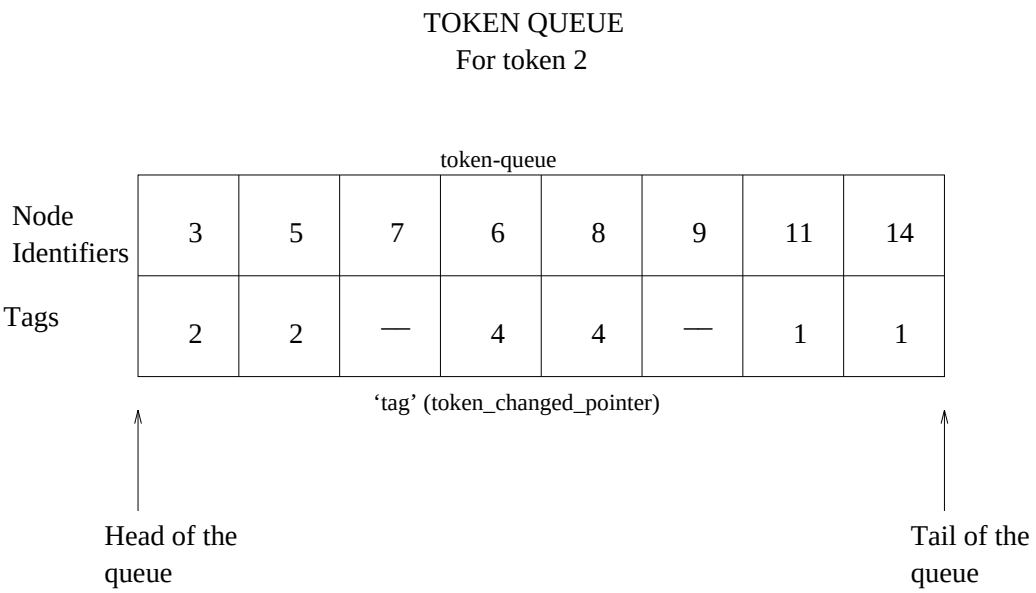


Fig. 3. Token-queue with its associated tag

3. Types of messages

There are three types of messages used by the algorithm:

- REQUEST message, using which a node sends a request for a particular token t to the node whose identifier is given by pointer[t].
- TOKEN message, using which a node sends a token and its token-queue to another node.

- INFORM message, using which a node informs a few other nodes that it has a particular token.

4. Proposed algorithm

The algorithm is presented below. The algorithm uses five procedures to maintain K-mutual exclusion. The procedure ENTRY_CS is executed whenever a node wants to enter the critical section. The procedure EXIT_CS is executed when a node exits the critical section. The other three procedures are *message handlers* for the messages of the three types defined above.

The pseudo-code of the algorithm is presented first followed by an explanation of the procedures. Note that in these procedures, 'I' is the identifier of the node that is executing the procedures.

```
Procedure Entry_CS:
{
    if (token = FALSE) then
    {
        Choose token t using some heuristics;
        send REQUEST(I,I,t) to pointer[t];
        waiting_for_token := t;
        wait until HOLDING becomes TRUE;
    }
    Enter Critical Section
}
```

```
Procedure Exit_CS:
{
    dest := First node on the token-queue;
    if (dest ≠ NULL) then
    {
        token_id := NULL;
        token := FALSE;
        pointer[t] := last_nulltag(t);
                /* pointer[t] is set equal to the last node whose tag is null
                If no such node exists then pointer[t] is set equal to the
                first node in the token-queue */

        send TOKEN(I,t) to dest;
    }
    else /* dest = NULL */
    send INFORM(I,t) message to any ν nodes;
}

Procedure Handle_REQUEST(X,Y,t):
                /* X denotes the adjacent node which sent the REQUEST message
and Y denotes the node where the request originated and t denotes the token being
requested */
{
    if (token = TRUE) and (holding = TRUE) then
    {
        enqueue Y into the token-queue;
```

```
        if (token_id ≠ t) then
            set tag of Y equal to I;
        else
            set tag of Y equal to NULL;
    }
    else if (token = TRUE) and (holding = FALSE) then
    {
        if (token_id ≠ t) then
            set tag of Y equal to I;
        else
            set tag of Y equal to NULL;
        send TOKEN(I,token_id) to Y;
        pointer[token_id] := Y;
        token := FALSE;
        token_id := NULL;
    }
    else if (waiting_for_token = t) then
        enqueue Y into the node-queue;
    else
        send REQUEST(I,Y,t) to pointer[t];
}

Procedure Handle_TOKEN(t):
{
    if (waiting_for_token ≠ t) then
    {
```

```
        pointer[waiting_for_token] := tag for node I on token-queue;
            /* To maintain proper pointer for the token that was originally re-
quested */
        Append the node-queue to the token-queue;
        For all the nodes that were in the node-queue, set their tags
        (in token_changed_pointer) equal to pointer[waiting_for_token];
    }
    else
    {
        Append the node-queue to the token-queue;
        For all the nodes that were in the node-queue, set their tags equal to NULL;
    }
    dequeue node I and its tag from the token-queue;
    waiting_for_token := NULL;
    token := TRUE;
    holding := TRUE;
    token_id := t;
    pointer[t] := I;
}

Procedure Handle_INFORM(Y,t):
            /* Y is the node that has token t */
{
    if (waiting_for_token ≠ t)
        pointer[t] := Y;
}
```

The procedures are explained below.

Entry_CS:

This routine is invoked when node I wants to enter critical section. If node I has a token then it enters the critical section without any further delay. Otherwise, using some heuristic it chooses a token t, where $1 \leq t \leq K$, and sends a request for token t to the node indicated by pointer[t]. In Figure 4, if node 4 wants to request for token 1, then node 4 sends a REQUEST message to node 3.

Exit_CS:

This routine is executed after a node exits the CS. Let node I have the token t. When node I exits the critical section, it sends the token t to the node x at the head of the *token-queue* (of token t), if the token-queue is not empty. The pointer[t] of node I is set equal to the last node on the token-queue that has its tag field null. If none of the tag fields are null, then pointer[t] of node I is set equal to node x, which is at the head of the token-queue.

If node I has token 2 with the token-queue shown in Figure 3, then it sends the token 2 to node 3 and sets pointer[2] equal to 9, the last node on the token-queue that has its tag field null. If none of the tag fields were null then, node I would set pointer[2] to 3, identifier of the node at the head of the token-queue.

If there are no requests on the token-queue then the current node continues to hold the token and sends INFORM messages to ν nodes chosen at random, informing them that it has token t. This is done to reduce the average distance of a node from a token which can reduce the average number of messages required per CS entry. The quantity ν is an input parameter of the algorithm. By varying ν, performance of the algorithm can be varied.

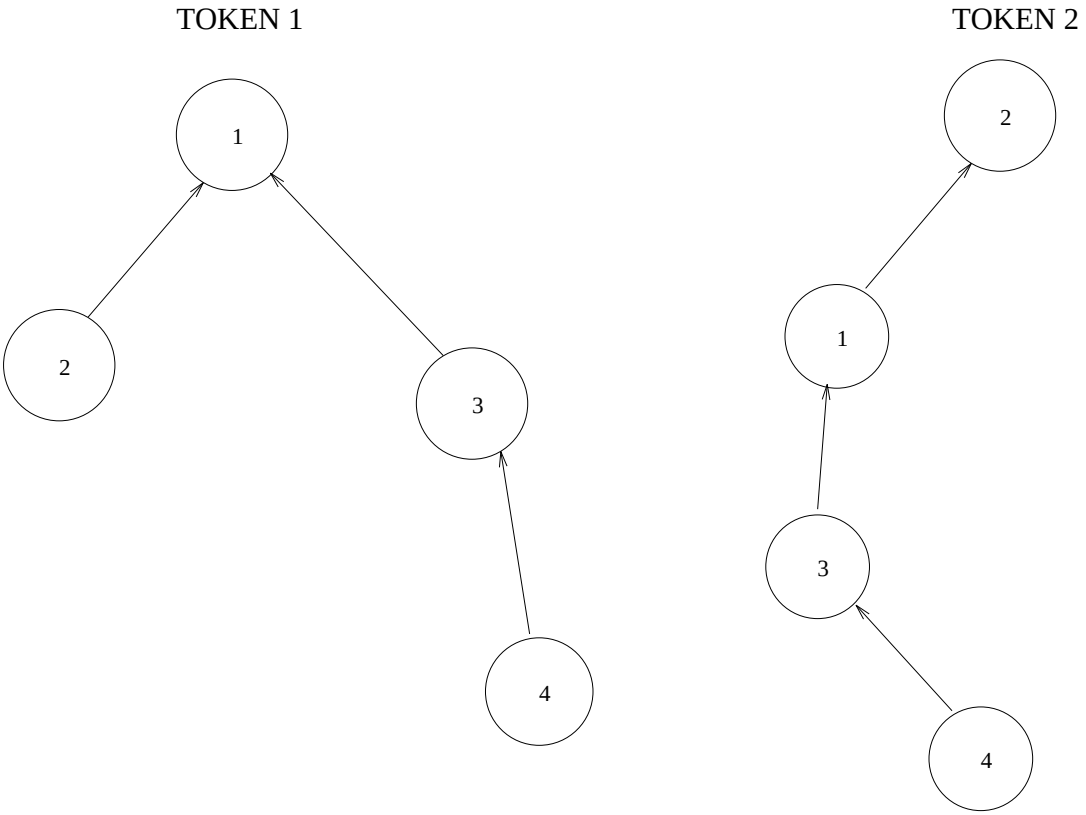


Fig. 4. A typical token tree

Handle_REQUEST(X, Y, t):

When a node receives a request for a token this routine is invoked. X is the node that forwarded the request to node I, Y is the node that is requesting token t. On receiving the request, the action taken by node I depends on the state of the node.

- *Case 1:* node I does not possess a token and is waiting to enter the CS. The action taken by node I depends on the token requested by node I.
  - *case a:* If node I is waiting for token t then, Y is stored in the node-queue.
  - *case b:* If node I is waiting for token p ($p \neq t$) then, REQUEST(I, Y, t) is sent to pointer[t] and pointer[t] set to Y.
- *Case 2:* node I does not possess a token and is not waiting to enter the CS. The request is forwarded to pointer[t] and pointer[t] is set equal to Y. Referring to Figure 5, if node 3 receives a request from node 4 for token 2, node 3 forwards the request to node 1 (as pointer[2] = 1) and changes pointer[2] to 4.
- *Case 3:* node I possesses a token and is not in the CS. Action taken by node I depends on the token present at node I.
  - *case a:* If node I possesses token t then, it sends the token to node Y, with Y inserted into the token-queue and sets pointer[t] to Y. The tag for node Y is set to null. An example is illustrated in Figure 6(a) where node 3 possesses token 2 and receives a request for token 2 from node 4.
  - *case b:* If node I possesses token p ($p \neq t$) then, it sends token p to node Y with Y inserted into the token-queue and sets pointer[t] to Y. The tag (in token_changed_pointer) for node Y is set to I. An example is illustrated in Figure 6(b) where node 3 possesses token 2 and receives a request for token 1 from node 4.

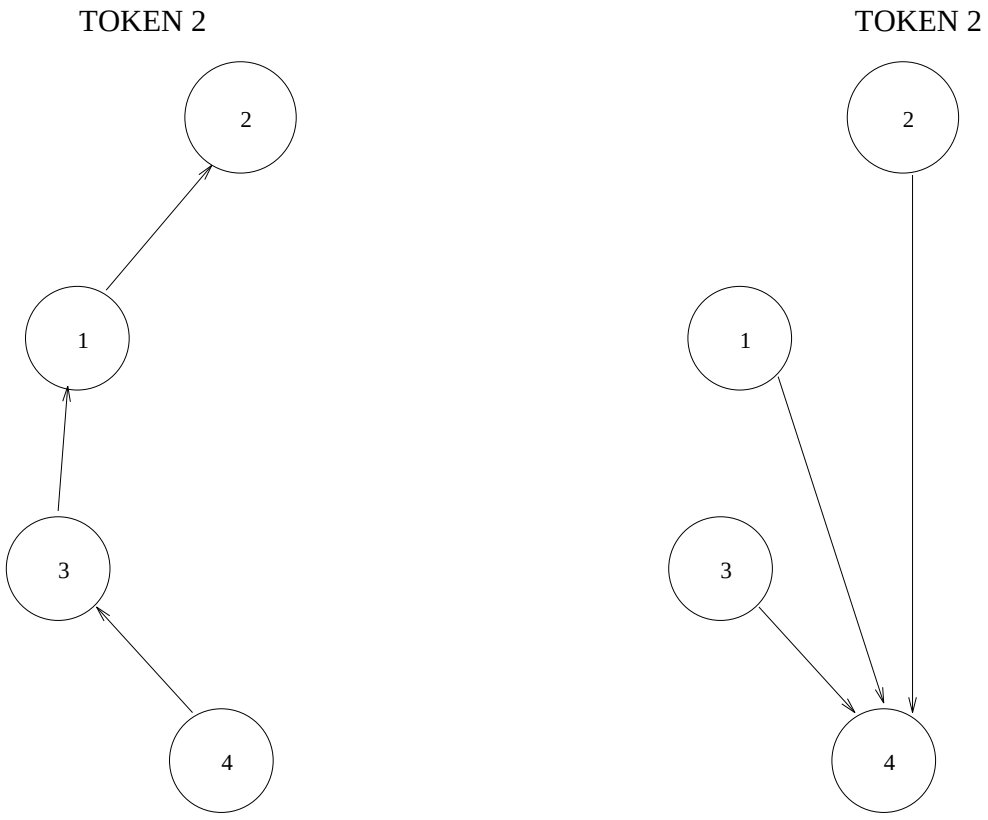


Fig. 5. Forest structure for token 2 before and after a request from node 4

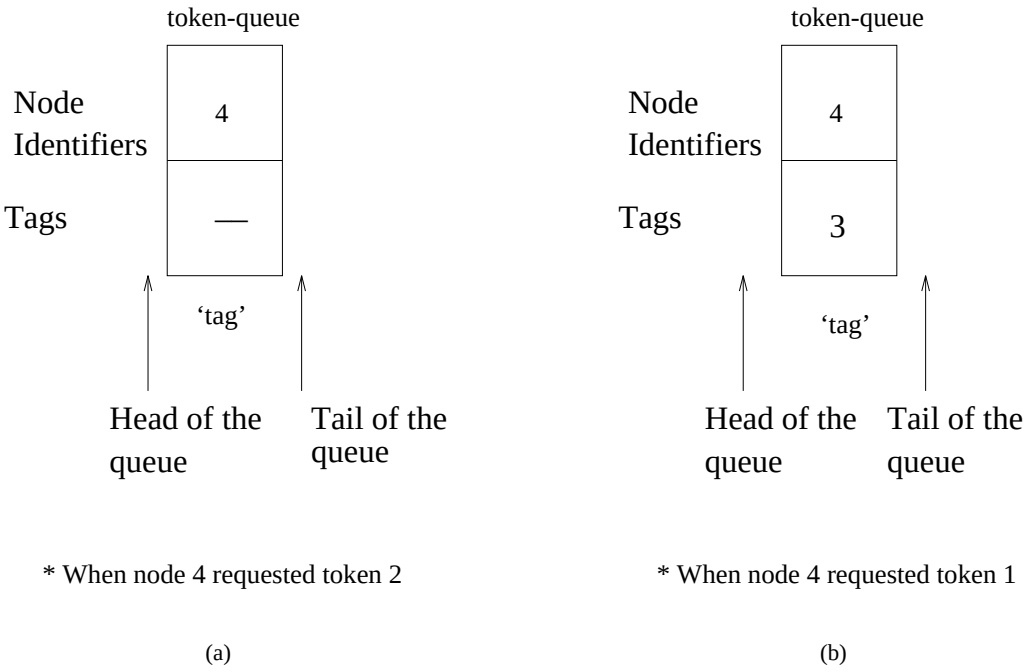


Fig. 6. Token-queue of token 2 at node 3

- *Case 4:* node I possesses a token and is in the CS. Action taken by node I depends on the token present at node I.
  - *case a:* If node I possesses token t then, it adds the request of node Y to the token-queue of token t. The tag in token_changed_pointer for node Y is set to null.
  - *case b:* If node I possesses token p ($p \neq t$) then,it adds the request of node Y to the token-queue of token p. The tag in token_changed_pointer for node Y is set to I.

The four cases above cause the "forest" structure of the token to change dynamically.

Handle_TOKEN(t):

This routine is executed when a node receives a token. Before entering the CS,

node I checks if token t is the same as the token it requested. The action taken by the node depends on the token received.

- *Case 1:* If node I had requested token p ($p \neq t$)then, pointer[p] is set to the tag (in token_changed_pointer) of node I at the head of the token-queue.

  Node I sets pointer[t] equal to I and appends the requests in the node-queue to the token-queue. The tags of the nodes, that were on the node-queue, are set equal to pointer[p] of node I.

  Referring to Figure 3, if node 3 receives the token 2, but had requested token 1, node 3 sets pointer[1] to 2, which is in the tag of node 3.

- *Case 2:* If node I had requested token t then, no special action is taken by node I. In this case, the tag of I in the token-queue of token t will be null.

  Node I sets pointer[t] equal to I and appends the requests in the node-queue to the token-queue. The tags of the nodes, that were on the node-queue, are set to null.

In both cases, node I dequeues its identifier and tag from the head of token-queue.

Handle_INFORM(Y, t):

This routine is executed when the node receives an INFORM message from node Y. On receiving this message, pointer[t] is set to Y. This helps in reducing the distance of a node from a token, as the most current information about the location of token t is made available via INFORM messages.

B. Proof of Correctness of the Algorithm

The algorithm achieves K-mutual exclusion and is free from deadlock and starvation.

1. Mutual exclusion

To achieve K-mutual exclusion we must show that at most K nodes can be in the critical section simultaneously. In our algorithm, since we have K tokens to implement K-mutual exclusion and a node could be in critical section only if it has a token, there can only be a maximum of K nodes in the critical section at a time. Hence, K-mutual exclusion is achieved.

2. Deadlock and starvation free

Initially, pointers for each token form a tree structure, where the nodes without the token point to the nodes that have the token. We claim that any modifications that are done to the pointers by the algorithm will maintain a forest structure for each token. (Note that a tree is a special case of a forest.) There are four situations where the pointers get modified.

- *Case 1:* When node j sends a token t to the requesting node i, node j sets its pointer[t] equal to i. If node j was initially the root for token t, then sending the token to node i and setting its pointer[t] to i will retain the forest structure.

  For example, in Figure 5 node 2 changes its pointer for token 2 to node 4 after sending it the token. This still retains the forest structure for token 2.

- *Case 2:* When node j receives a request from node i for token t and node j does not have a token and has not requested token t, node j forwards the request to pointer[t] and sets pointer[t] equal to i. If we started with a forest originally,

then this change in the pointer will result in a forest too.

For example, in Figure 5, node 3 changes its pointer for token 2 to node 4 after forwarding node 4's request to node 1. This still retains the forest structure for token 2.

- *Case 3:* When node j gets token t and it had originally requested token p ($p \neq t$), then node j sets its pointer[p] equal to the tag of node j on the token-queue. The tag is essentially the identifier of the node, say x, that modified the request of node j and put node j on the token-queue of token t. The fact that the request of node j for token p reached node x, indicates that node x was on the path between node j and the node with the token p. Hence, node x has the correct pointer for the token p. Hence, making pointer[p] of node j equal to x will still maintain the forest for token p.

  Also making the pointer[p] for the nodes on the node-queue of j, equal to x, instead of j, only improves the performance while still maintaining the forest for token p.

  For example, the solid arrows in Figure 7 represent the forest structure for token 1 where node 2 and node 7 have requested for token 1 and sent their requests to node 4.

  Suppose node 4 had token 2 when it received the request from nodes 2 and 7, it puts these nodes on the token-queue of token 2 and makes their tag equal to 4 (its own id). When node 4 exits the CS, it sends the token to node 2 which is at the head of the token-queue. When node 2 gets token 2, though it had originally requested token 1, it sets its pointer[1] to node 4, which is the tag value associated with node 2 on the token-queue. This change of the pointer for token 1, will also result in a forest structure if we began with one initially.

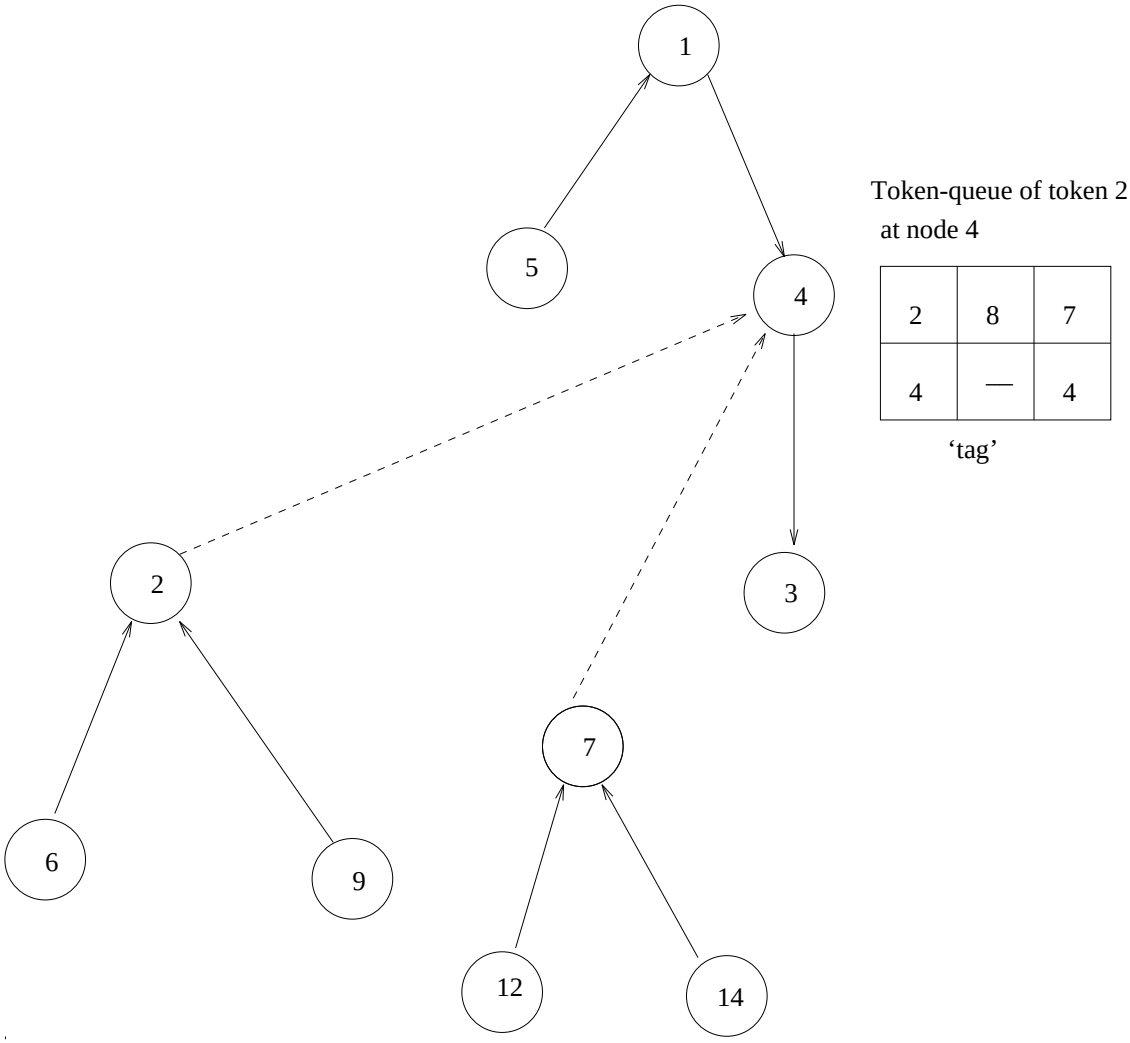


Fig. 7. Forest structure for token 1

- *Case 4:* When node j, not waiting for token t, receives an INFORM message from node i for token t, node j sets pointer[t] to i. This still maintains the forest structure for token t if we initially started with a forest.

Hence we prove that a forest is maintained at all times for each token.

The modifications to the forest structure discussed in the four cases above imply that pointer[t] of a node say, X always leads to a node that has entered CS using t after node X's most recent entry into the CS using token t.

To prove that the system is deadlock and starvation free, we can examine the protocol when a node i in the system wants to enter the critical section. If node i has a token, it enters the critical section without any delay. If, however, it does not have a token, then request for some token t is transmitted along the edges corresponding to the pointers for token t, till it reaches a node j that has a token or has sent out a request for token t. If node j has a token then it puts the request of node i in the token-queue (if j is in the critical section) or sends the token to node i (if j is not in the critical section). In any case, node i will eventually get the token. If node j has also sent out a request for token t, then request of node i is put in the node-queue of node j. Thus if node j's request is serviced then node i's request will also be served. In this case i is said to be blocked by node j.

Deadlock occurs when nodes block each others request for the same token in a cyclic fashion. Figure 8 depicts a deadlock situation. If request of node 1 is blocked at node 2, and request of node 2 is blocked at node 5, then node 1 is essentially blocked at node 5. Applying this argument recursively, node 1 is essentially blocked at node n. Now, if the request of node n is also blocked at node 1, then a situation of deadlock exists.

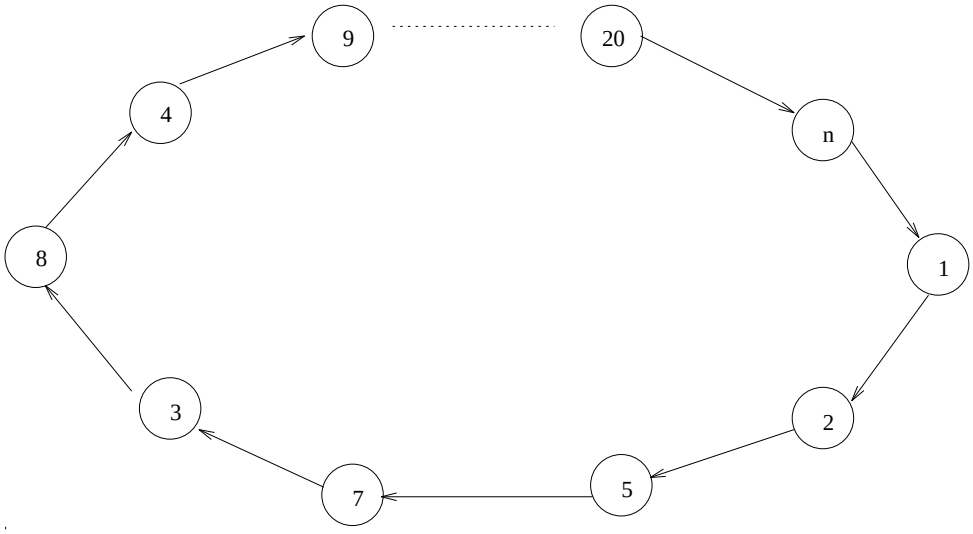


Fig. 8. Case of deadlock for a request for token t

In general, deadlock occurs when a request of node X gets blocked at node Y for token t and request of node Y gets blocked at node X for the same token t as depicted in Figure 9. A loop is formed by the blocked nodes. Let the hashed node B represent a set of those nodes that entered CS using token t after node X's most recent entry into the CS using token t and also are on a path from X to Y. Let the hashed node A represent a set of nodes that entered CS using token t after node Y's most recent entry into the CS using token t and also are on a path from Y to X. If Y's request is blocked at node X for token t, then there exists a path from Y to X through A and hence from Y to B. This implies that B entered the CS using token t after Y. Hence a path from B to Y cannot exist. Hence no loop can be formed.

Also, all the requests will be serviced in a finite delay as the token-queue is finite (as we have only one request for entry to critical section at any time for a given node).

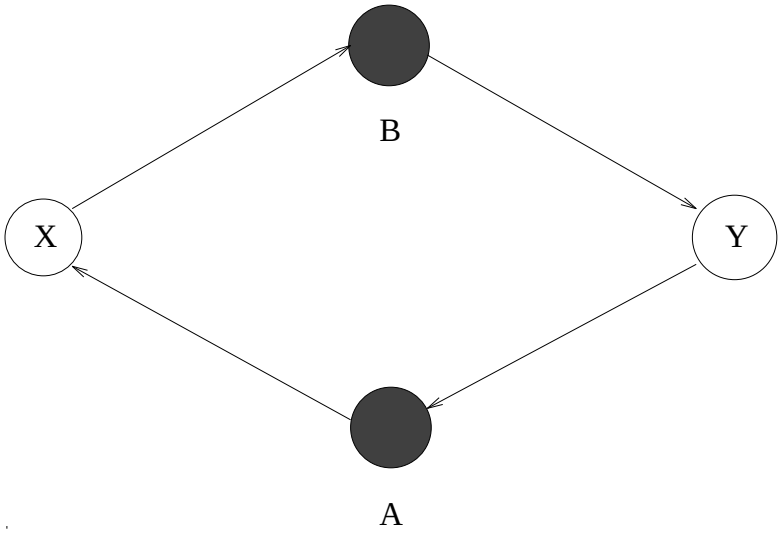


Fig. 9. General case of deadlock for a request for token t

C. Performance

Under light load, there is a smaller chance that the token-queue will contain any nodes. Whenever the token-queue is empty, the Exit_CS procedure informs ν nodes the whereabouts of a token, say t. This reduces the average delay in entering the critical section at the cost of ν extra messages. Under light load, this also reduces the average distance of a node from from the token t, which in turn results in the reduction of the average number of messages. Thus, the net effect of the INFORM messages is often to reduce the average number of messages required.

Under heavy load, there is a good chance that the token-queue is not empty when the Exit_CS procedure is performed. In such a case, our algorithm will not send the INFORM messages. Thus the algorithm improves the performance by sending the INFORM messages only when beneficial.

Also, when a node i exits from the critical section and sends the token to the first node on the token-queue, of say token t, it makes pointer[t] point to the last node x on the token-queue that requested the token t. This reduces the distance from the

token as any further requests from node i will be sent to node x. So average number of messages as well as delay are reduced.

When node i gets a request of node j for a token t and passes the request along the edge of the forest for token t, it makes its pointer[t] equal to j. This reduces the distance from the token and hence the average number of messages and delay for an entry into the critical section are reduced.

When a node i requesting token t receives a request message of another node j for the same token t, then node i will put node j's request in its node-queue rather than propagating the request as in [11]. Hence, unnecessary message transmission is avoided. This reduces the average number of messages.

1. Heuristics for choice of token

Performance of the algorithm is dependent on the decision mechanism used by each node to decide which token to send the request for. One possibility is to choose the token randomly. Heuristic that we experimented with chooses the *last seen token*. t is the last seen token if:

- The node recently received token t.
- The node recently received an INFORM message from a node possessing token t.

If node i remembers that it had last seen the token t and makes a request for that token, there is a better chance of node i's request reaching the token t with a small number of hops. The INFORM messages help in updating the last seen token with the most current information.

CHAPTER V

SIMULATION RESULTS

We simulated our algorithm and compared it with the other three K- mutual exclusion algorithms proposed by Raymond [19], Srimani and Reddy [20] and Makki et al. [21]. We simulated a slightly modified version of Srimani and Reddy's algorithm. This algorithm is presented in Appendix A. The parameters for comparison are the *average time to enter the critical section*, the *average number of messages per critical section entry* and the *average information* per message. The information content was calculated by adding the length of the token-queue and its associated tags, the source, origin and destination addresses, the token identifier and the sequence number as applicable to the different algorithms.

A. Simulation Model

The simulation model used here is a refinement of the model presented by Singhal [8]. There are N nodes in the system where each node may request an entry into critical section τ time units after completing the previous execution of the critical section, τ being exponentially distributed with mean $1/\lambda$. λ is specified as CS requests per unit time. The time spent by each node in the critical section is E units. Each node spends T_s time units when sending a message (time spent in the network layer). Similarly, each node spends T_r time units when receiving a message. T_t is the transmission time between two nodes.

The simulation model presented by Singhal [8] assumes that $T_s = T_r = 0$. Essentially, his model assumes that T_s and T_r are negligible compared to the transmission delay T_t. However T_s and T_r are no longer insignificant when the communication medium becomes fast. For example, when a high-speed network such as FDDI is

used for communication, the time spent executing the network layer software may not be negligible as compared to the transmission delay. We have shown by our simulations that T_s and T_r can affect the results and cannot be neglected.

B. Simulation Results

Simulations were carried out for a system of thirty nodes and three tokens, for various values of rate of the critical section requests (λ) and for three different values of T_s, T_t and T_r. The time spent in the critical section E was chosen to be 0.0002 (as in Singhal [8]) and 1 for two sets of simulations. The number of nodes ν to which INFORM messages are to be sent was fixed at 2.

The simulations were run for 5000 critical section entries and the average delay in entering the critical section and the average number of messages were calculated. This number was chosen because we observed that the results of the simulation converged by 5000 entries into the critical section.

Simulations were also carried out for ten nodes and one token with the rest of the parameters same as above. This was to simulate a *Partitioning Scheme*, to verify if a system of thirty nodes and three token performed better when partitioned into three systems of ten nodes and one token each. We came up with some interesting results.

The results of the simulation are discussed below.

1. Comparison of simulation results

Figure 10 shows the *average time* taken to enter the critical section, by the four different algorithms where $T_r = 0.1$, $T_s = 0.1$ and $T_t = 0.8$. In the graph, 'proposed' refers to our algorithm with the heuristic in Chapter IV section C and 'proposed_no_heuris'

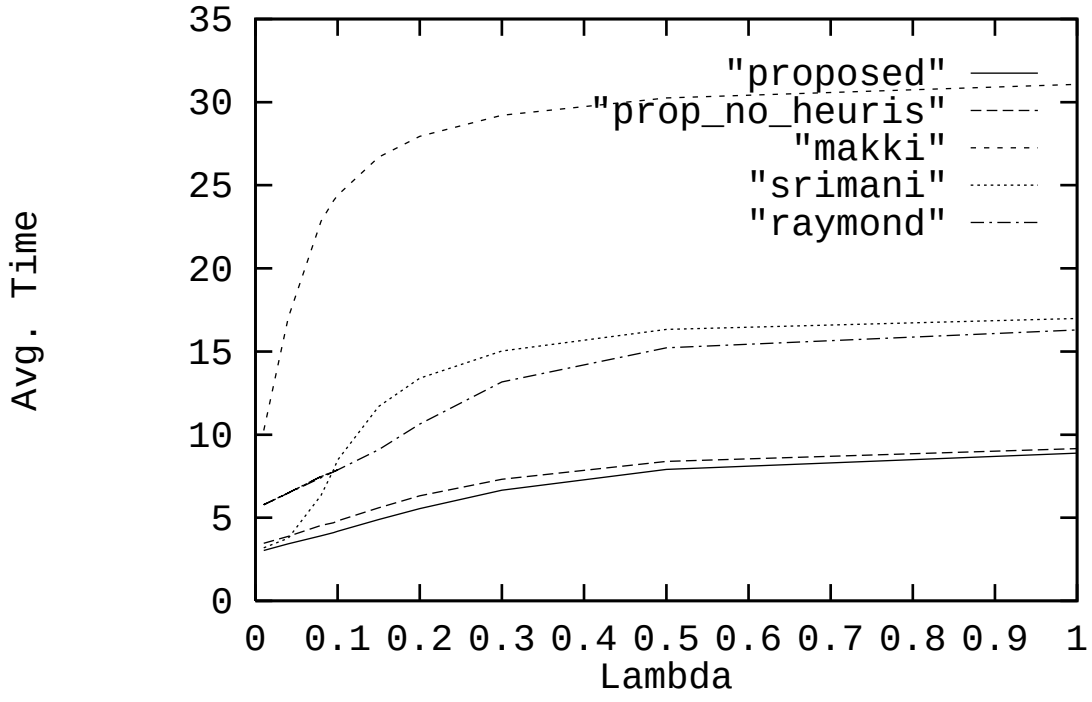


Fig. 10. Average time to enter the critical section for $T_r = 0.1$, $T_s = 0.1$ and $T_t = 0.8$

is when token is chosen randomly. Our algorithm performed better than all the other algorithms. As λ increases, the number of requests for entry into critical section increase, which causes a larger delay for each node. Hence the curves show a steady rise initially but gradually flatten out for greater values of λ. This is because there exists a maximum rate at which a node can enter the critical section using a given algorithm. Increasing λ does not increase this rate.

For a system of thirty nodes and three tokens, our algorithm requires about nine units of time for a node to get the token when the arrival rate is 1 *request/unit time*, i.e. every node makes a request every second. This can be easily explained because, each of the three tokens basically divides the system into three subsections and on an average there are nine to ten nodes in the token-queue for each token at very high request rates. So it takes nine to ten hops for the token to get to the node eventually. It can be seen that adding heuristics has improved the performance by a small amount.

Raymond and Srimani's algorithm takes around sixteen to eighteen time units delay in getting the permission to enter the critical section and the token, respectively. This is due to the time spent in sending the request to all the nodes and waiting for the reply. In Srimani's algorithm it was observed that every privilege message visited the requesting node before being forwarded to the next requesting node at low loads. This resulted in more delays.

The time to enter the critical section is maximum for Makki as the effect of having $K = 3$ is nullified by the fact that there is a finite transmission time involved in the transmission of the TOKEN. Though the RELEASE messages may reach a node, the TOKEN takes much longer to reach. The result is that the system effectively behaves as if $K = 1$ on heavy loads. For example in Figure 11 where $K = 2$, the time taken to transmit the TOKEN from node 2 to 7 (via node 4) is more compared to time

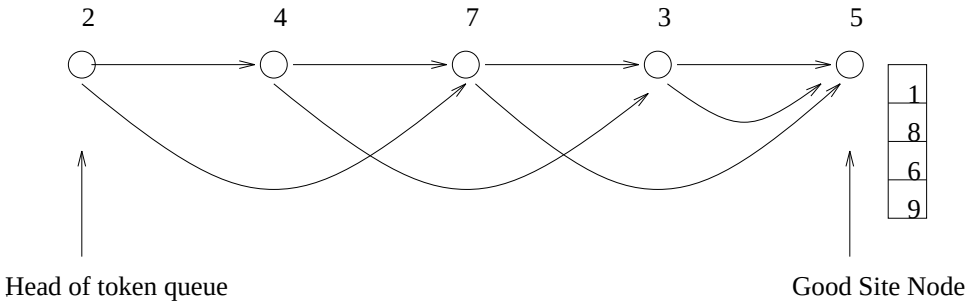


Fig. 11. Makki et al.'s algorithm: token-queue for $K = 2$

taken to transmit the RELEASE message from node 2 to node 7. Hence node 7 will have to wait till it gets the token to enter the CS. Even if the algorithm is modified so that a node can enter CS on getting a RELEASE message, the node will not know where the RELEASE message is to be sent on exiting the CS unless it receives the token.

The performance of Makki's algorithm could be improved significantly if the token-queue was sent with the RELEASE message also. This would enable the node, that received the RELEASE message, to send a RELEASE message to the appropriate node on the token-queue after it exited the CS.

For T_r, $T_s \neq 0$, Makki's algorithm does not work correctly as such. We simulated a slightly modified version that yield optimistic results for Makki's algorithm when T_r, $T_s \neq 0$. In particular, Makki's algorithm [21] assumes that all *delayed requests* will be received within $2T_t$ time units after sending the *good site* message. When T_r, $T_s \neq 0$, this assumption is not valid and some messages may arrive after $2T_t$ time units. In our simulation, we "accelerate" such messages so that the algorithm will work correctly.

By making T_r and T_s zero and $T_t = 1$ time unit, Raymond's algorithm shows a considerable drop in the average time to enter the critical section. This is shown in Figure 12. This is expected due to the nature of the algorithm where, for each

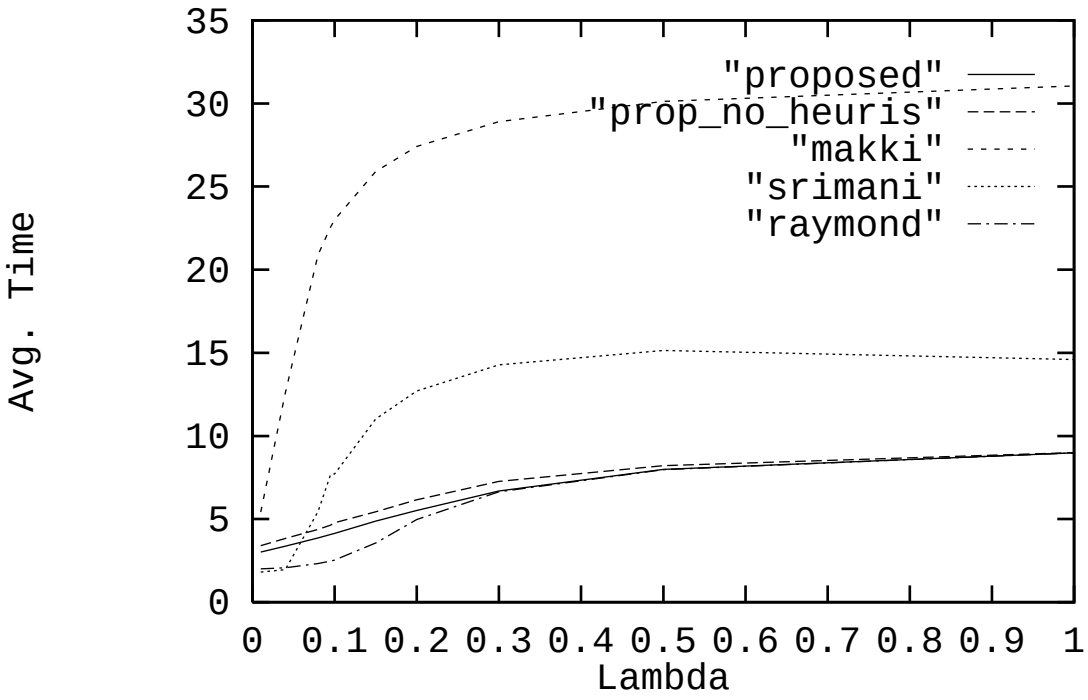


Fig. 12. Average time to enter the critical section for $T_r = 0$, $T_s = 0$ and $T_t = 1.0$

entry into the critical section, a node has to send $(N-1)$ request messages. Srimani's algorithm too shows some decrease in the average time to enter the critical section. If T_r and T_s are zero the number of messages sent parallely does not affect the time delay. In Makki's algorithm, since the good site information is sent to the nodes that are not on the token-queue, at high loads, these messages are very few because most of the nodes are on the token-queue. Hence making T_r and T_s zero does not have much effect on the curves. In our algorithm the broadcasting of INFORM messages are done to a small, fixed number of nodes, so the performance is not affected significantly by the sending and receiving delays.

Figure 13 shows the measurements for $T_r = 0.05$, $T_s = 0.05$ and $T_t = 0.9$. The

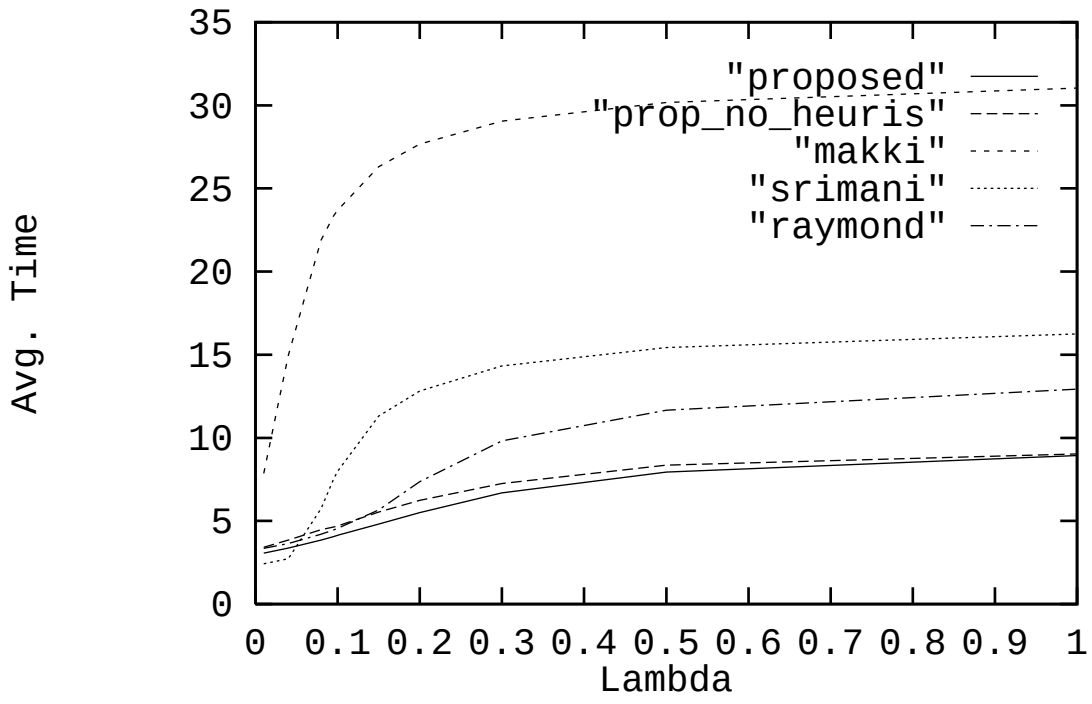


Fig. 13. Average time to enter the critical section for $T_r = 0.05$, $T_s = 0.05$ and $T_t = 0.9$

curves are in between the curves in the graph for $T_r = 0.1$, $T_s = 0.1$ and $T_t = 0.8$ and

the curves in the graph $T_r = 0$, $T_s = 0$ and $T_t = 1.0$. This shows that the sending and receiving delays cannot be neglected as it affects the performance of some algorithms.

For values of T_r and T_s greater than 0.1, our algorithm would show a greater improvement compared to other algorithms.

The *average number of messages* per CS entry for the four algorithms was measured. Figure 14 shows the average number of messages versus λ. Our algorithm

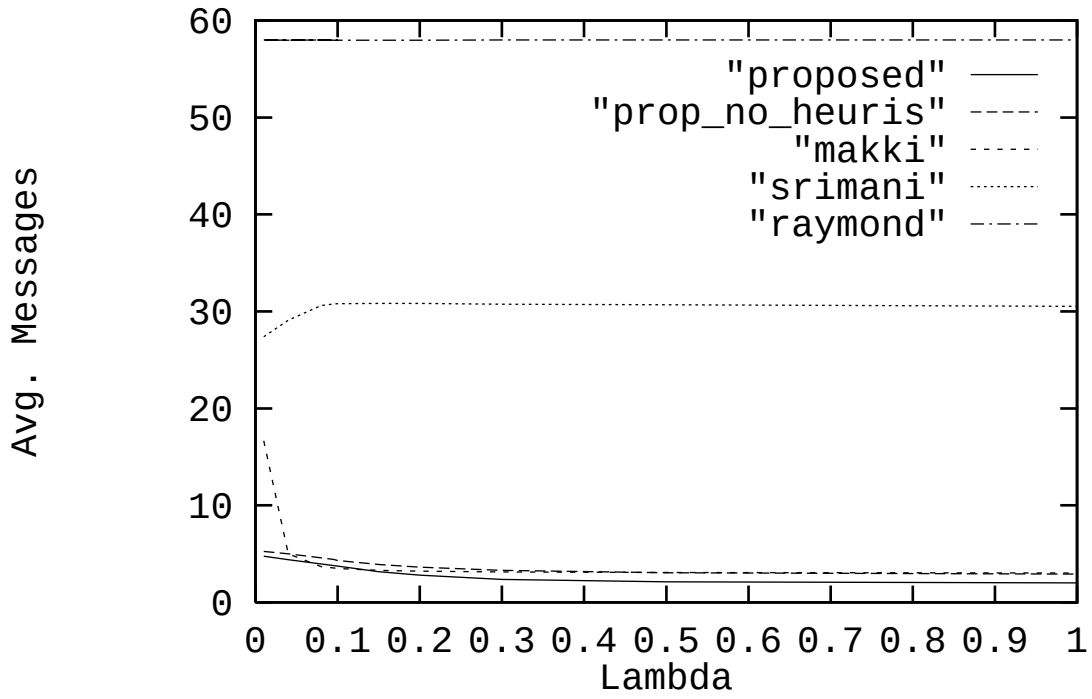


Fig. 14. Average # of messages per critical section entry for $T_r = 0.1$, $T_s = 0.1$ and $T_t = 0.8$

has the lowest number of messages compared to the other algorithms. The number of messages is comparatively high, around five messages, at light loads (i.e. small λ). This is to be expected because at light load the request is forwarded more often before it reaches a token. However, at heavy loads, the possibility of a request reaching a

node that has also requested for the same token increases. Hence, the forward propagation of these requests is reduced resulting in fewer number of messages. At very high loads, on an average, approximately two messages are needed. Also, applying the heuristics for choosing a token has reduced the number of messages.

Raymond's algorithm has a lower bound of $2N - K - 1$ on number of messages required and an upper bound of $2 * (N - 1)$, where N is the number of nodes in the system and K is the number of critical section entries allowed at a time. With the simulation values of $N = 30$ and $K = 3$, the lower bound is 56 and the upper bound is 58. The graph shows that this is indeed true and the number of messages average around 57 messages per critical section entry.

Srimani's algorithm has an upper bound of $N + K - 1$ messages and a lower bound of zero messages, where N is the number of nodes in the system and K is the number of critical section entries allowed at a time. The analysis suggests that the average number of messages per critical section entry is close to $(N - 1)$ [20]. With the simulation values of N and K, the upper bound turns out to be 32 messages per critical section entry. The graph shows that the average number of messages needed is around 31 messages agreeing with the analysis.

Makki's algorithm suggests that [21] at lower λ rates, the number of messages required is quite large and that the performance of the algorithm improves as the number of requests for the token increase, with only three messages being required in the heavy load case. This can be seen in the graph of Figure 14 where the average number of messages required per critical section entry reduce to three. Although the number of messages required is small, as seen before, with large λ, Makki's algorithm results in longer delays.

Doing the measurements for the average number of messages for $T_r = 0.05$, $T_s = 0.05$ and $T_t = 0.9$ and also with $T_r = 0$, $T_s = 0$ and $T_t = 1.0$ it was found that

the number of messages is practically the same for all the cases. This implies that the average number of messages per critical section entry is not affected by the sending and receiving time. This is shown in Figures 15 and 16.

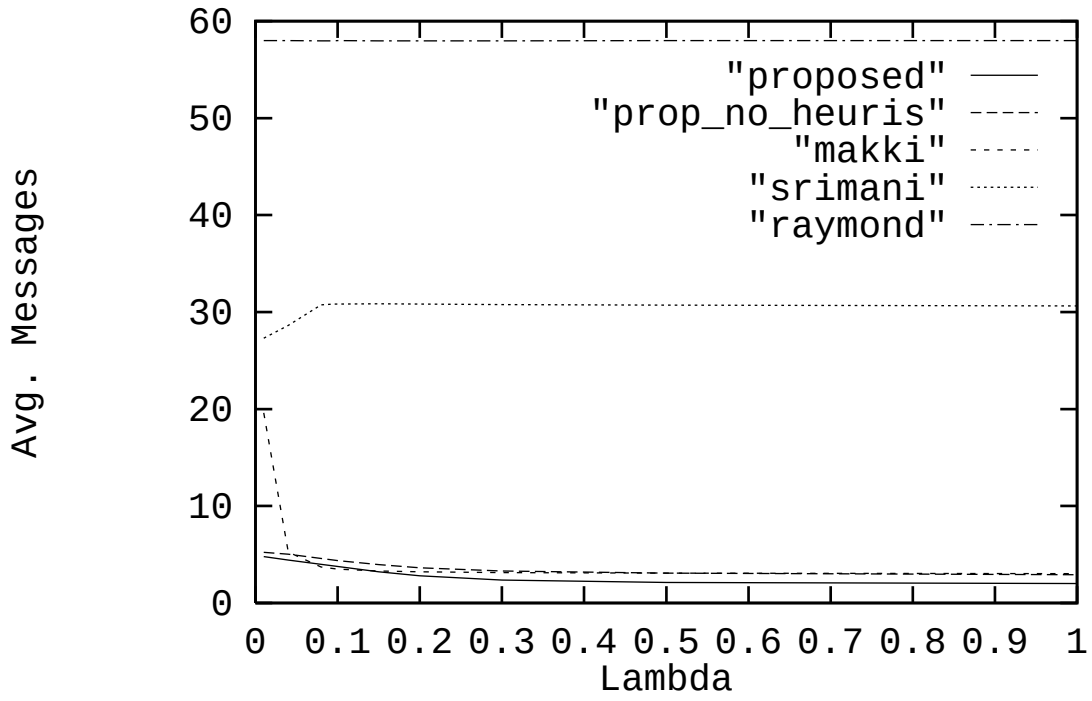


Fig. 15. Average # of messages per critical section entry for $T_r = 0.05$, $T_s = 0.05$ and $T_t = 0.9$

The *average information* that is passed in the messages is about 9 words for our algorithm for $\lambda = 1$ as we need to pass the token-queue and the tags along with the token. In case of Raymond's algorithm, 4 words are needed for all values of λ, since here only request and reply messages are sent. In Srimani's algorithm about 6.5 words are transferred on an average. This is because, here again, the token-queue is sent along with the token. Makki's algorithm required about 8 words as the token semaphore and token is sent with the token. This is illustrated in Figures 17, 18

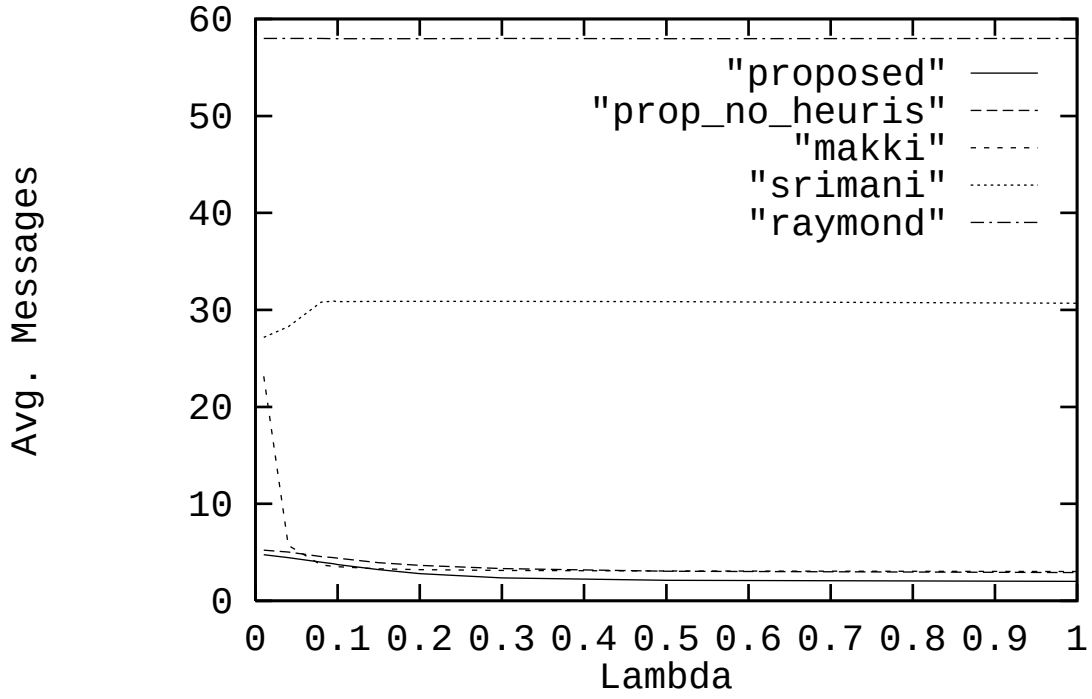


Fig. 16. Average # of messages per critical section entry for $T_r = 0$, $T_s = 0$ and $T_t = 1.0$

and 19. Note that the average information content is insensitive to the values of T_r

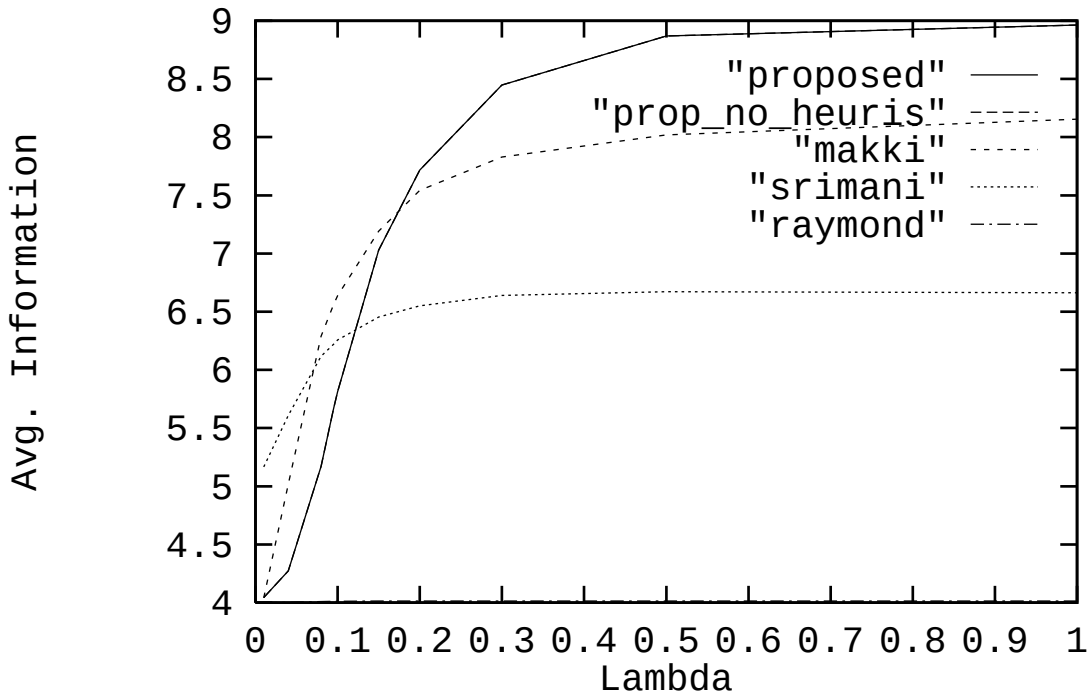


Fig. 17. Average information (in words) per message for $T_r = 0.1$, $T_s = 0.1$ and $T_t = 0.8$

and T_s.

Thus, for almost the same amount of information content that is transferred between nodes, our algorithm has the least average time to enter the critical section and the least average number of messages to enter the critical section (when T_r and T_s are non-zero).

Experiments were also carried out for $E = 1$. The results are similar to that for $E = 0.0002$ discussed above. Figures 20, 21, 22 plot the average delay and Figures 23, 24 and 25 plot the average number of messages, for various values of T_r, T_s and T_t.

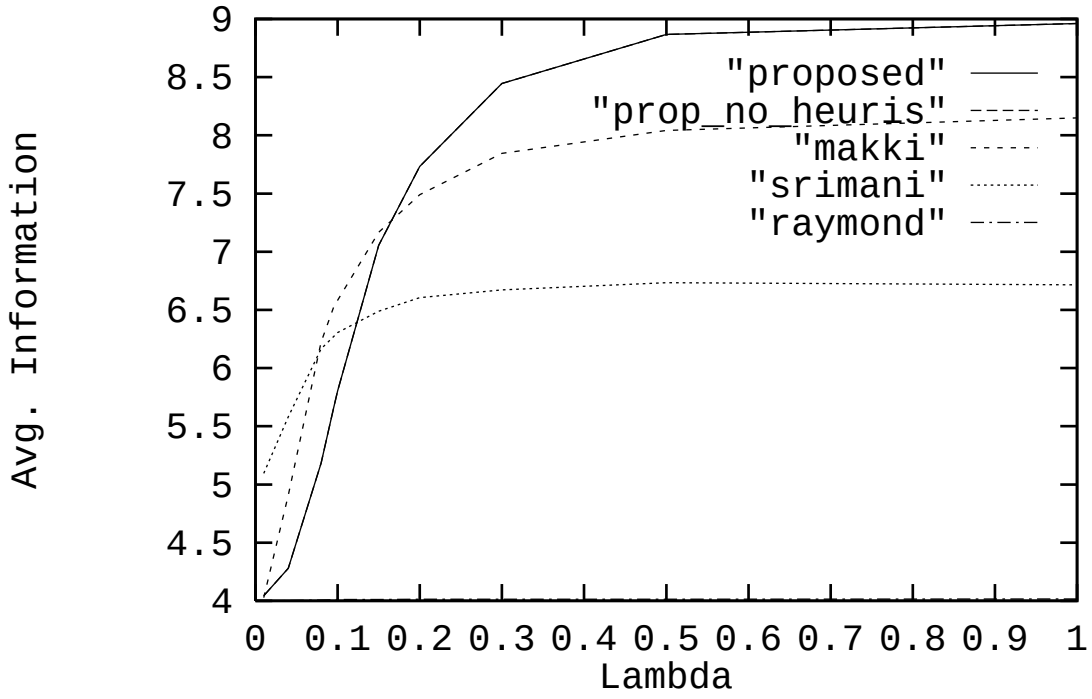


Fig. 18. Average information (in words) per message for $T_r = 0.05$, $T_s = 0.05$ and $T_t = 0.9$

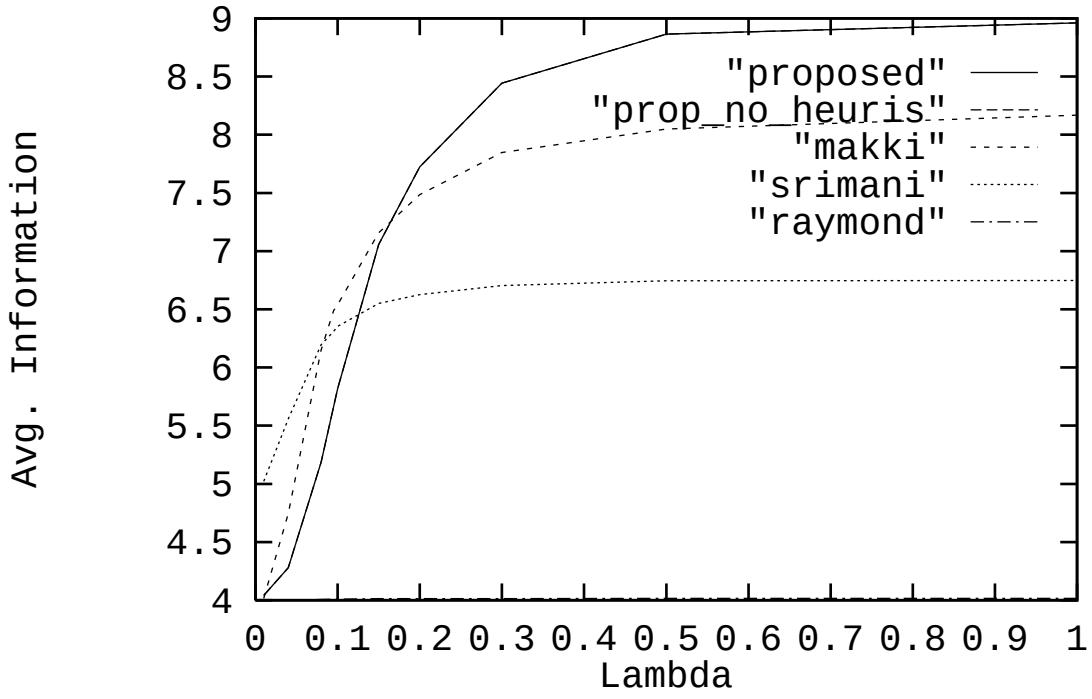


Fig. 19. Average information (in words) per message for $T_r = 0$, $T_s = 0$ and $T_t = 1.0$

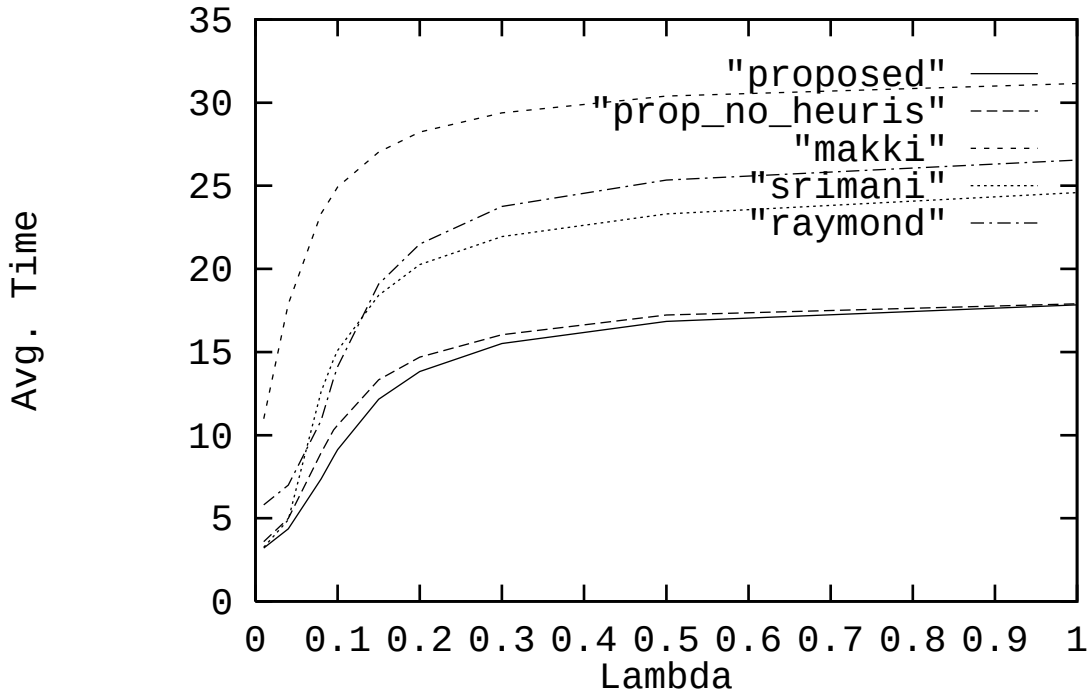


Fig. 20. Average time to enter the critical section for $T_r = 0.1$, $T_s = 0.1$ and $T_t = 0.8$ and $E = 1$

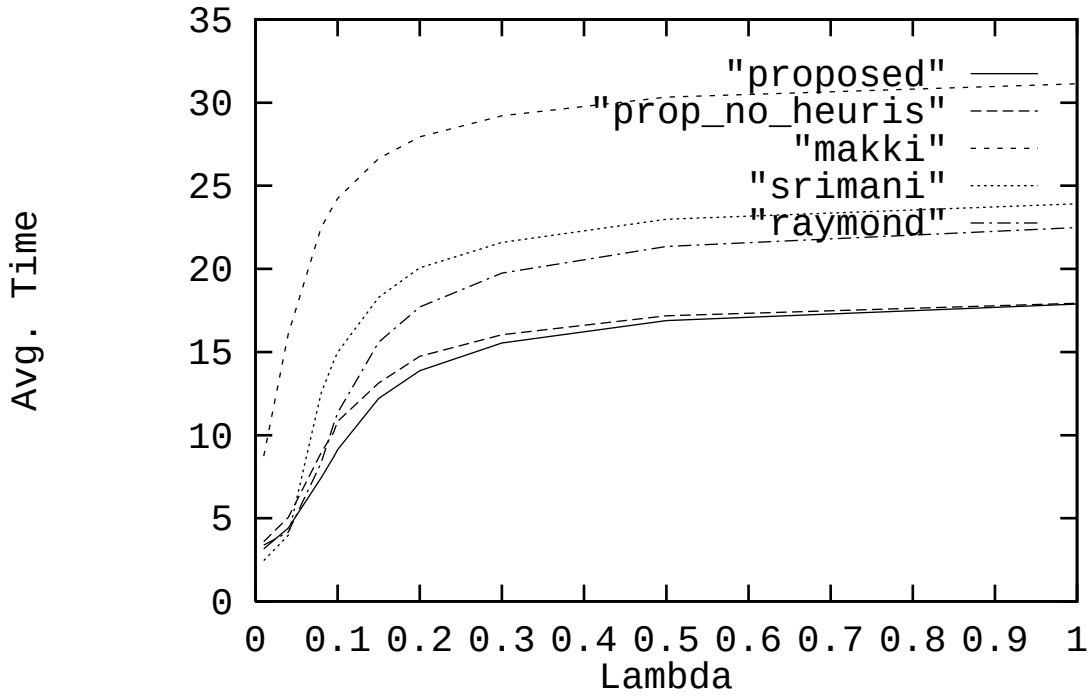


Fig. 21. Average time to enter the critical section for $T_r = 0.05$, $T_s = 0.05$ and $T_t = 0.9$ and $E = 1$

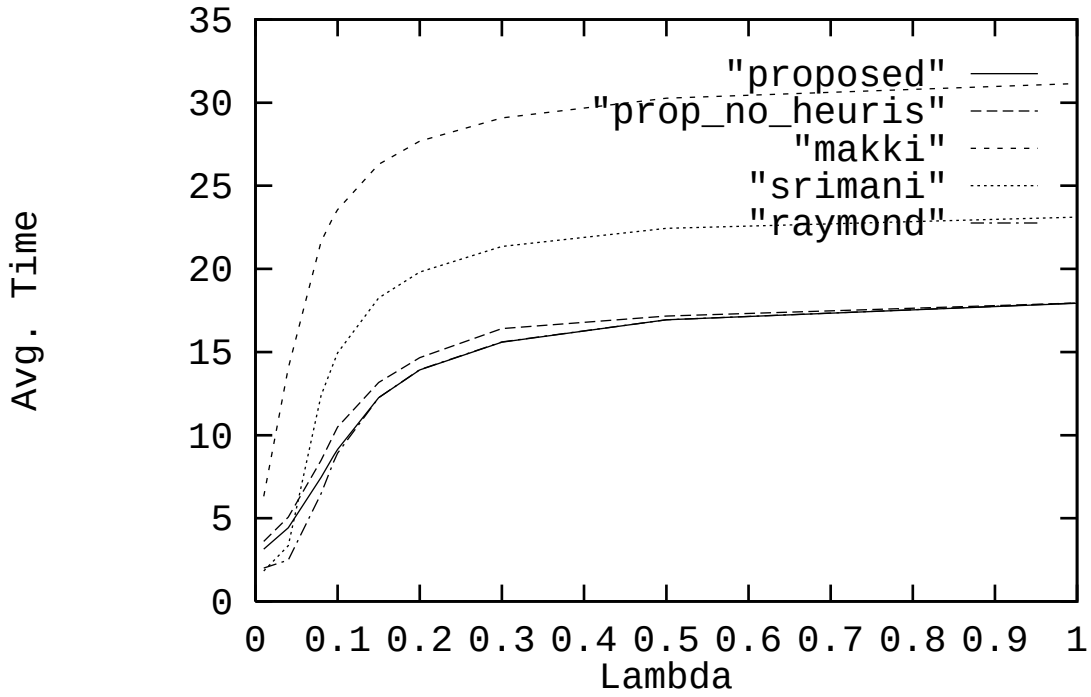


Fig. 22. Average time to enter the critical section for $T_r = 0$, $T_s = 0$ and $T_t = 1.0$ and $E = 1$

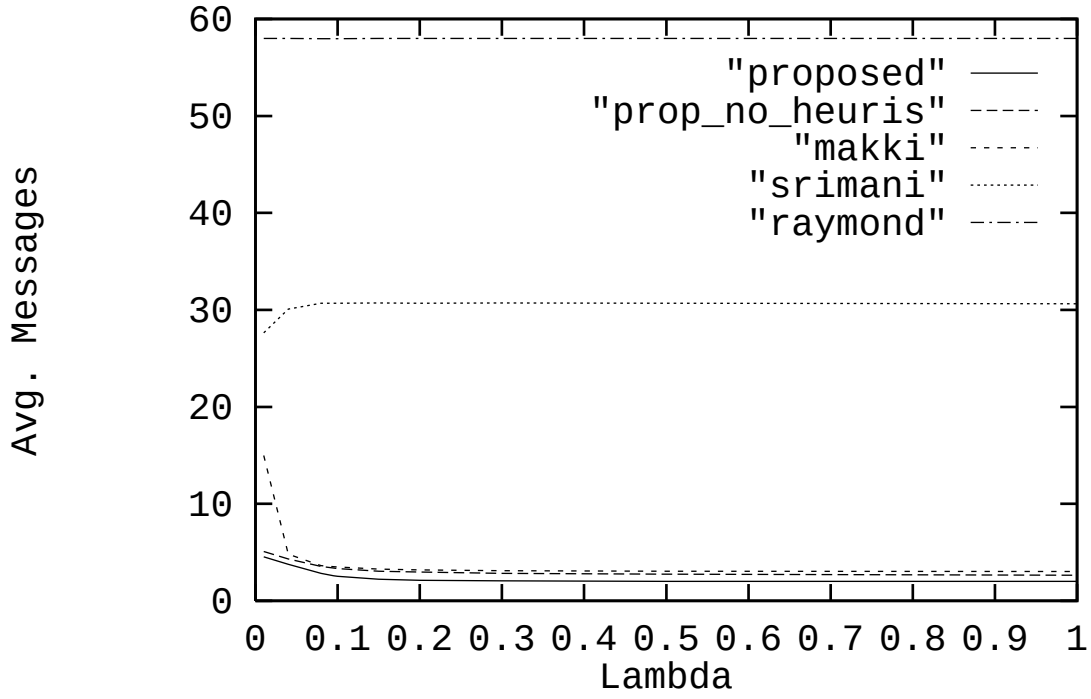


Fig. 23. Average # of messages per critical section entry for $T_r = 0.1$, $T_s = 0.1$ and $T_t = 0.8$ and $E = 1$

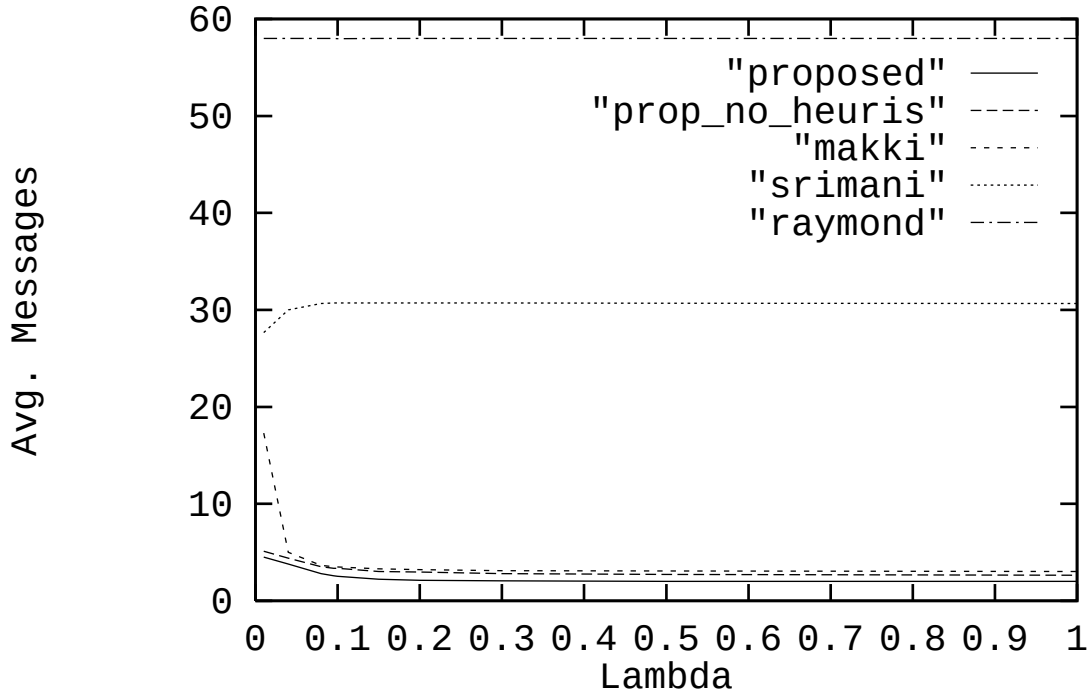


Fig. 24. Average # of messages per critical section entry for $T_r = 0.05$, $T_s = 0.05$ and $T_t = 0.9$ and $E = 1$

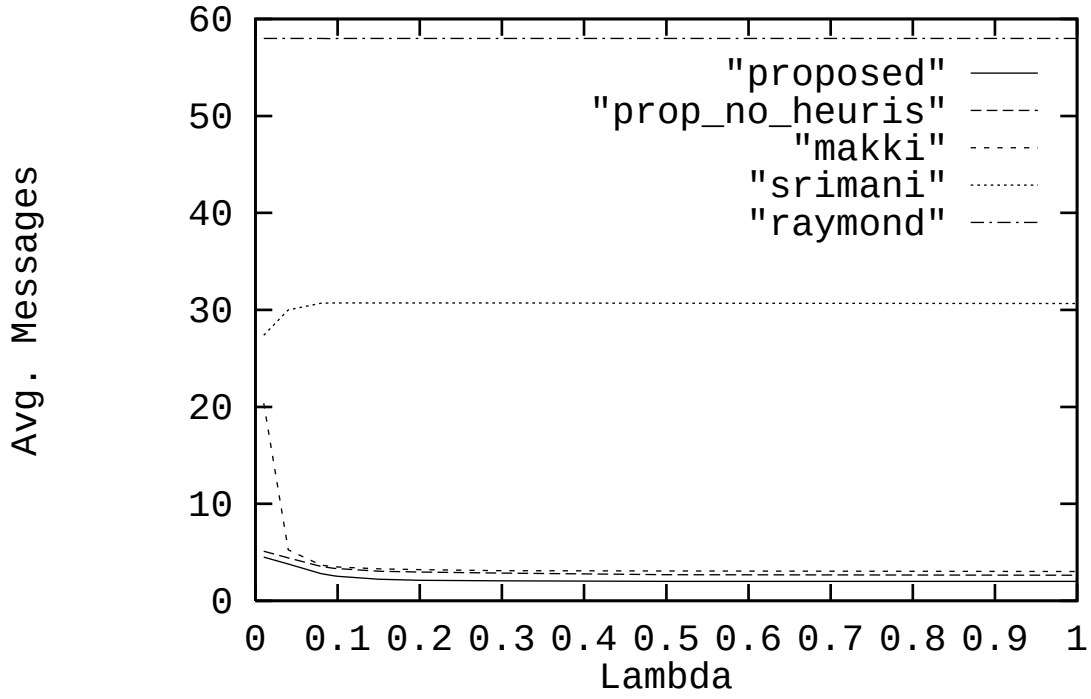


Fig. 25. Average # of messages per critical section entry for $T_r = 0$, $T_s = 0$ and $T_t = 1.0$ and $E = 1$

One part of the *sanity checks* of our simulations was to verify that the average number of messages that the simulation results showed matched approximately with that obtained by an analysis of some of the algorithms. The other check to verify the correctness of the simulations was to see that all the nodes in the system entered the critical section for roughly equal number of times and the waiting time of every node was not beyond reasonable limits.

Simulations were also carried out by *partitioning the system* of thirty nodes and three tokens into three systems of ten nodes and one token each. Makki's, Raymond's and Srimani's algorithms all perform better under partitioning. The time to enter the critical section reduces in a partitioned system with these algorithms. This is interesting because it implies that the existing algorithms are not very suitable for K-mutual exclusion. The average time to enter the critical section for our algorithm in a partitioned system is the same as for the unpartitioned scheme. The performance of our algorithm is, however, comparable to or better than the other algorithms. This can be seen in Figure 26. The scheme with heuristics performs the same as that without heuristics because there is only one token in the system of ten node. Figures 27 and 28 illustrate that the effect on various values of T_r, T_s and T_t on the curves is the same as that in the unpartitioned scheme.

The average number of messages in the partitioned scheme for our algorithm has remained the same as that of the unpartitioned scheme. Since the other algorithms are sensitive to the number of nodes in the system, the average number of messages have reduced proportionately as can be seen in Figure 29. Changing the parameters T_r, T_s and T_t has insignificant effect on the average number of messages.

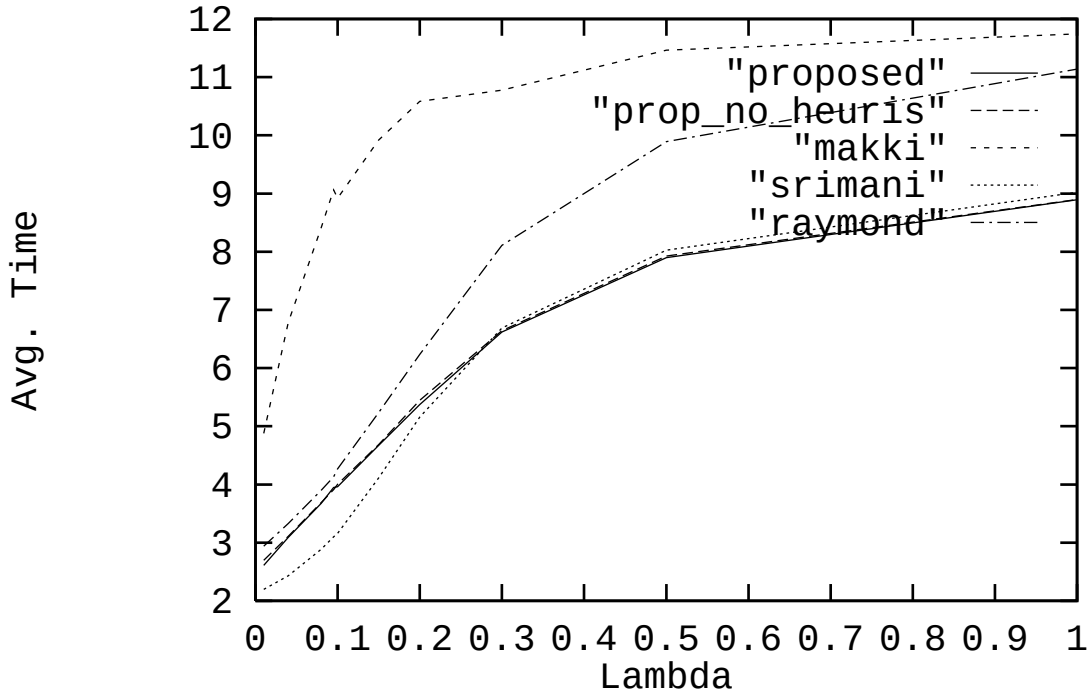


Fig. 26. Partitioning scheme: average time to enter the critical section for $T_r = 0.1$, $T_s = 0.1$ and $T_t = 0.8$

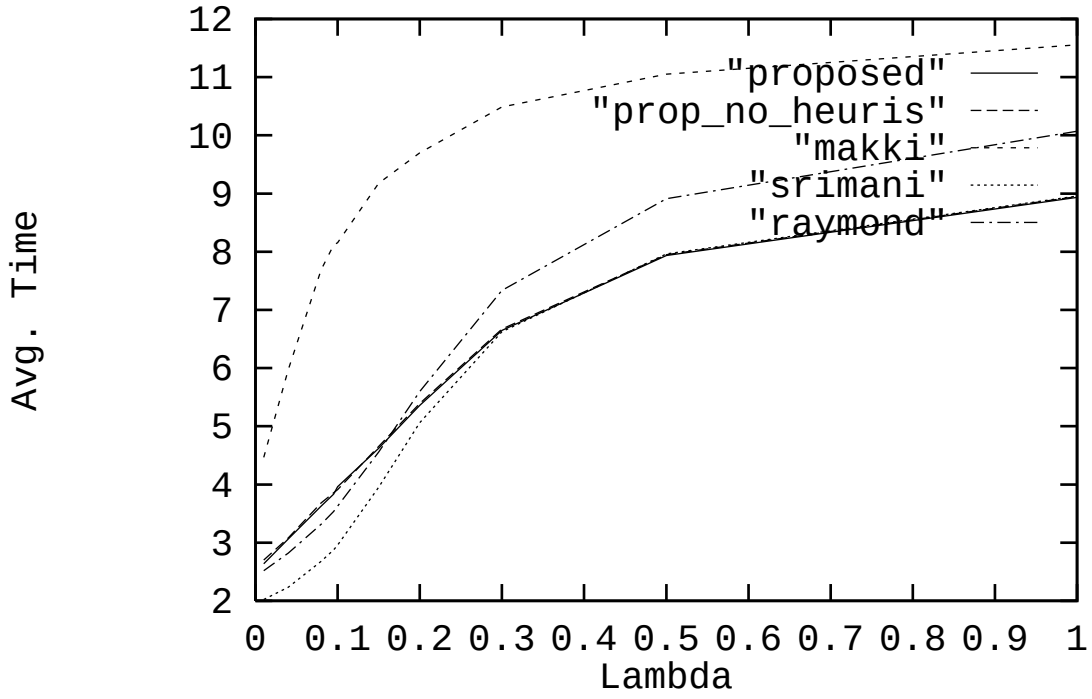


Fig. 27. Partitioning scheme: average time to enter the critical section for $T_r = 0.05$, $T_s = 0.05$ and $T_t = 0.9$

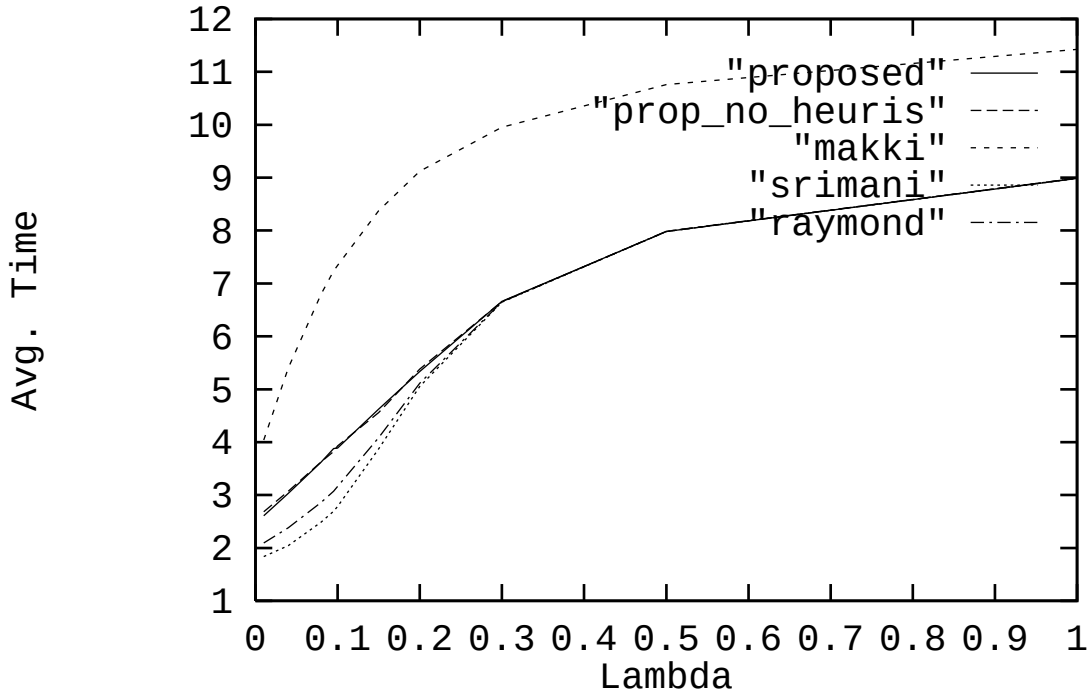


Fig. 28. Partitioning scheme: average time to enter the critical section for $T_r = 0$, $T_s = 0$ and $T_t = 1.0$

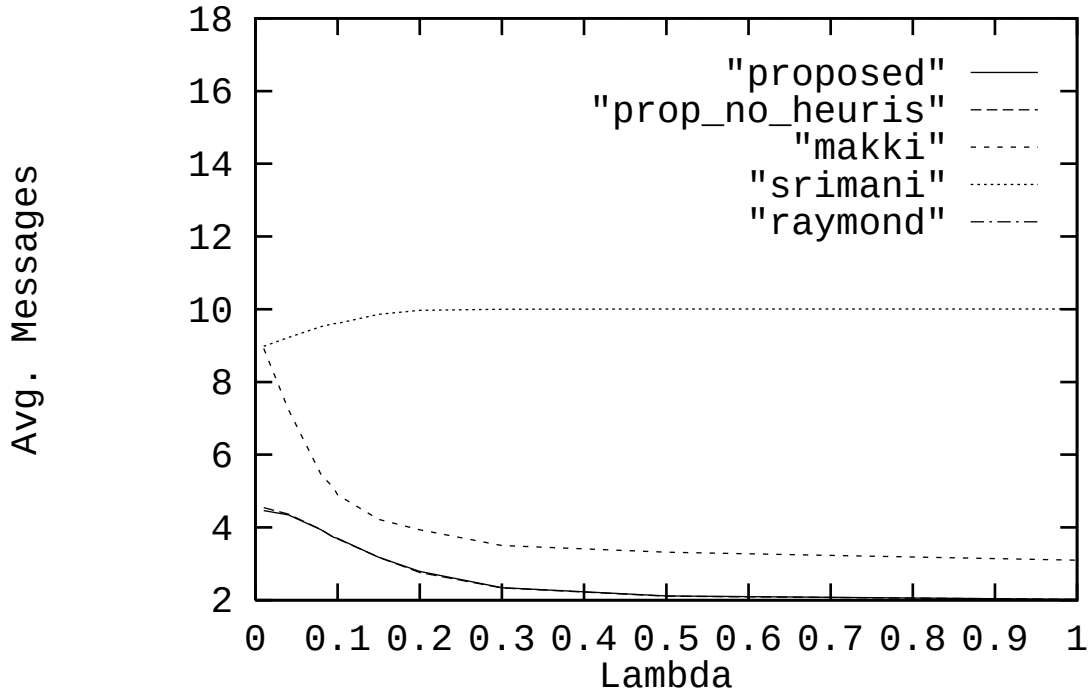


Fig. 29. Partitioning scheme: average # of messages per critical section entry for $T_r = 0.1$, $T_s = 0.1$ and $T_t = 0.8$

CHAPTER VI

IMPLEMENTATION

We implemented our algorithm and Raymond's algorithm on a UNIX Sun Workstation environment with a network of nodes connected by ethernet. TCP/IP protocol was used for communication between nodes. Each node is logically connected with every other node in the system and can send messages to any node in the system.

A. Details of Implementation

Ten Sun Workstations on the ethernet were first connected using the TCP/IP protocol. Once the connections were established, each node executed the same copy of the various procedures of the algorithm. The processes on the nodes have a simple structure in which they repeatedly alternate between computations outside a critical section and computations inside a critical section. The procedures are the same as those used in the simulations, with some minor modifications to handle the messages using the TCP/IP send and receive message protocol. Message arrivals can occur at any time and are handled serially as they arrive.

The time spent in the critical section, E is taken to be $2000\mu s$ and the number of nodes ν to which the inform messages are sent is fixed to two. K was assumed to be 3. The critical section request rate is varied over a range of values. The measurements are taken for 200 entries into the critical section for each node. To achieve a degree of confidence in the measurements, the standard deviation of the average number of messages per critical section entry and average delay is calculated over the nine samples produced by the different processes.

The parameters such as average time to enter the critical section and the average number of messages per critical section entry are measured. One of the ten nodes

was a dummy node to collect statistics from all the other nodes and to calculate the average number of messages and average time to enter the critical section. So basically there were nine nodes in the system that were performing the mutual exclusion.

B. Results

The results of the implementation, showed that our algorithm performed similar to the simulations. The number of messages required was about three to four messages at light load and at heavy loads, the number of messages averaged to two messages per entry into the critical section, just as it did for the simulations. This is seen in Figure 30.

For Raymond's algorithm too the number of messages in the critical section averaged to about 15 messages, well within the upper bound of 16 messages for a system of nine nodes (calculated from $2*(N-1)$). The lower bound for the messages in a system of nine nodes was calculated to be 14 using the formula $2N - K - 1$. Figure 30 shows the measurements of the average number of messages for Raymond's algorithm. The time measurements of the implementation cannot be compared with that of the simulations because the model assumed for simulation was different from that for the experimental system.

We calculated the round trip delay of the system for sending and receiving the messages. This turned out to be around $13ms$ on the average. So the average message delay time from one node to another was around $6.5ms$.

The measurements showed that at high loads, our algorithm required $25ms$ for a node to get the token as seen in Figure 31. This is approximately equivalent to four message hops. This is about what one might expect for a system of nine nodes and three tokens.

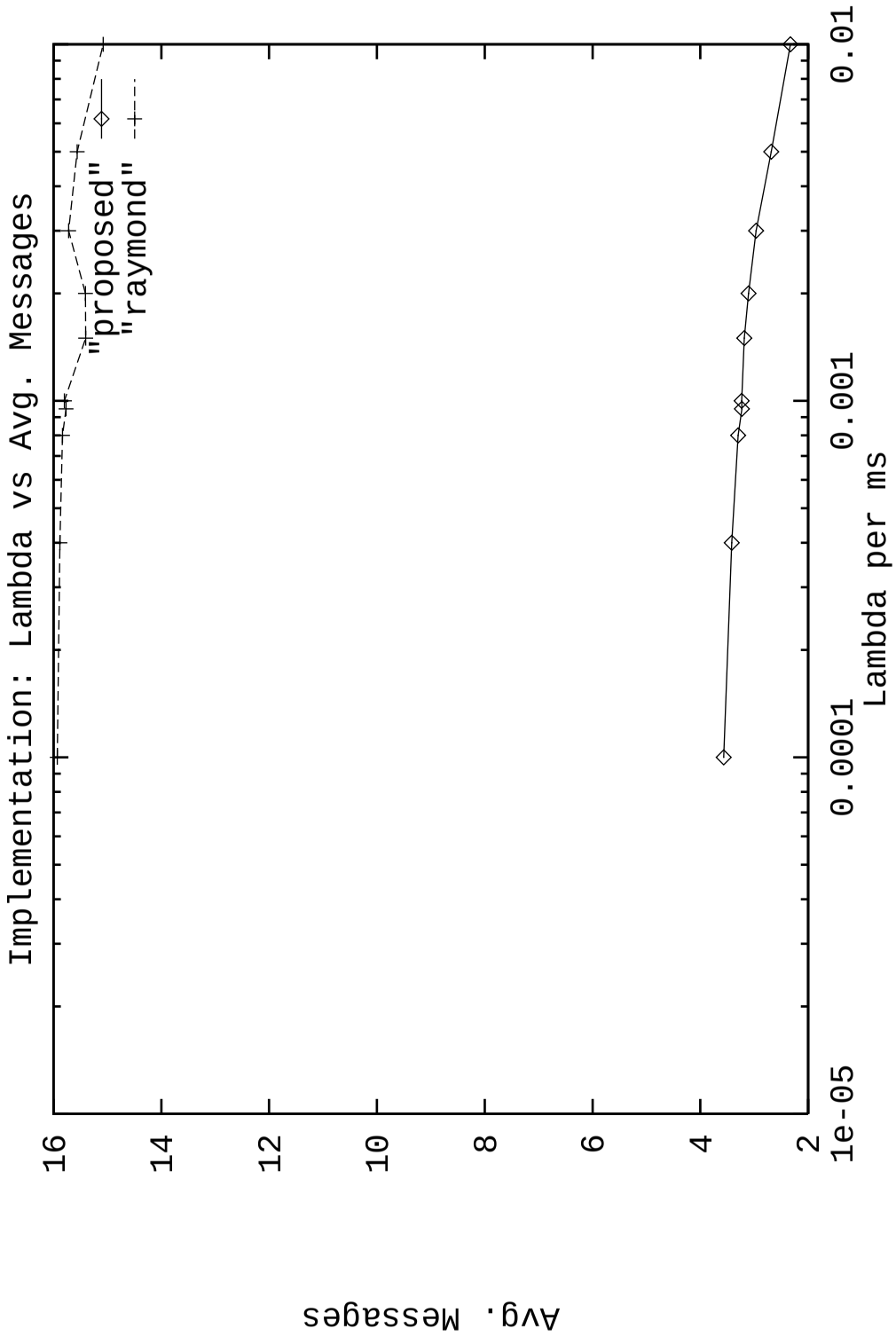


Fig. 30. Implementation: average # of messages per critical section entry

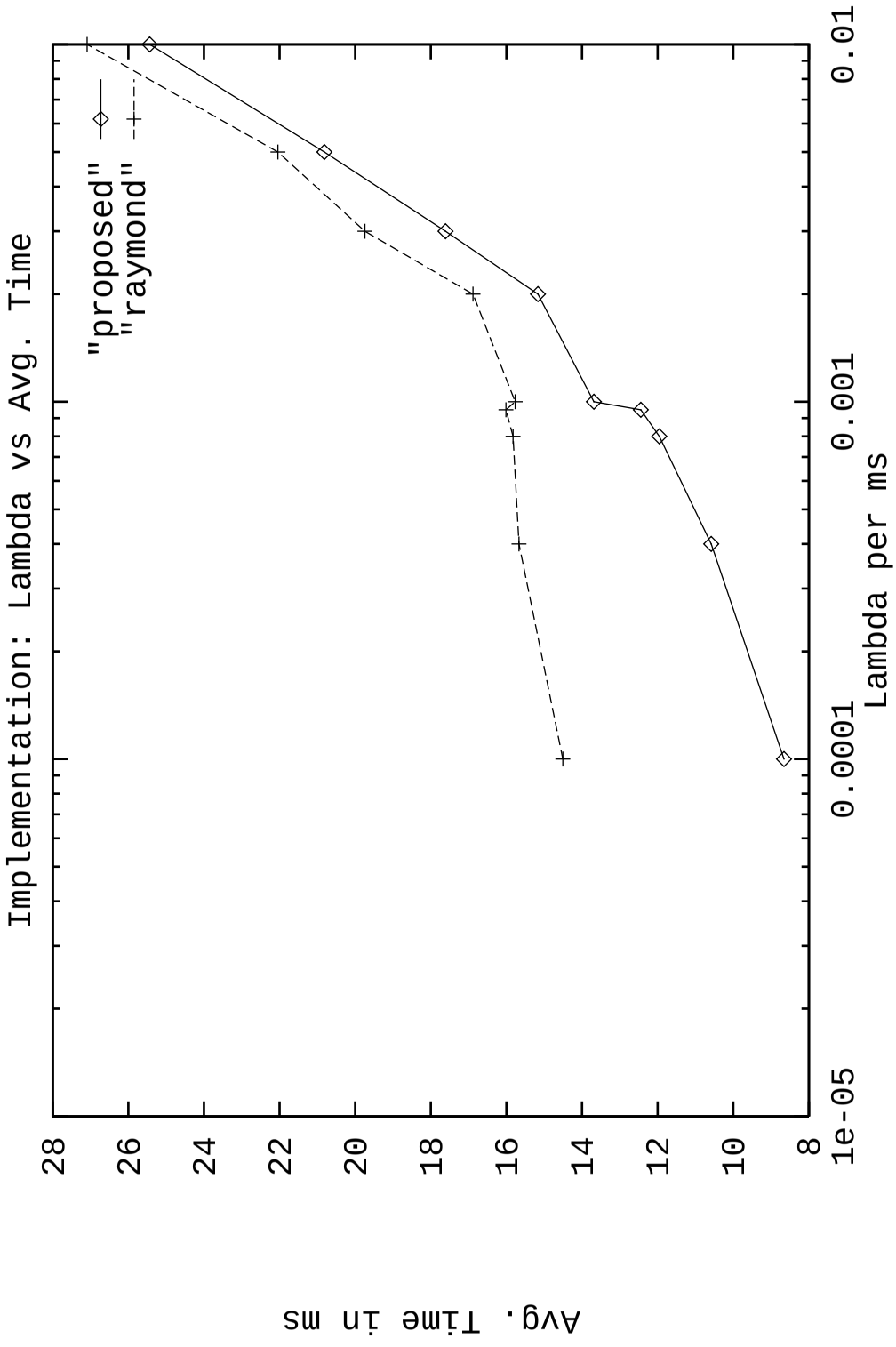


Fig. 31. Implementation: average time to enter the critical section

Table I. Standard deviations for the proposed algorithm

λ in ms^{-1}	S.D. for average # of messages	S.D. for average delay in ms
0.000100	0.1628299	0.51004291
0.000400	0.1362799	0.54537131
0.000800	0.1646376	0.45759222
0.000950	0.1689692	0.39158795
0.001000	0.1181891	0.41565752
0.002000	0.1621289	0.43096136
0.003000	0.1355544	0.55824485
0.005000	0.1662327	0.41734271
0.010000	0.1260511	0.52181130

Measurements of the implementation of Raymond's algorithm showed that a node got the permission to enter the critical section in about four to five message hops at heavy loads, with the actual time for getting the token being about $27ms$. Figure 31 shows the time measurements for the Raymond's algorithm.

The curves in the graphs are not smooth due to the effect of the varying network load on the measurements. The trend though is obvious. The standard deviations (S.D.) for the average number of messages and average delay per critical section entry are tabulated in the Table I for our algorithm. The standard deviations were evaluated over the nine samples provided by the nine processors in our experiment. The standard deviations for Raymond's algorithm is shown in Table II. The standard deviations are quite small compared to the averages, giving us enough confidence in our measurements for 200 critical section entries per node.

Table II. Standard deviations for Raymond's algorithm

λ in ms^{-1}	S.D. for average # of messages	S.D. for average delay in ms
0.000100	0.0394014	0.55298836
0.000400	0.0793700	0.47895460
0.000800	0.0958523	0.57512490
0.000950	0.1300878	0.49831504
0.001000	0.1229825	0.56785351
0.002000	0.3321572	0.42134403
0.003000	0.1820960	0.52645571
0.005000	0.2610555	0.51134079
0.010000	0.4250359	0.43649802

CHAPTER VII

CONCLUSIONS AND FUTURE WORK

In this report we have presented an algorithm for K-Mutual Exclusion in a distributed system, where sites do not share a global memory and communicate solely by passing messages over a communication network.

Our algorithm is token based, with K tokens to implement the K-mutual exclusion. A token-queue and a *tag* associated with each node on the token-queue, form the data structure of the token. When a node wants to enter the Critical Section (CS), it enters the CS immediately if it has a token; else requests a token from another node. The nodes maintain information about the likely path to reach each token and send the request messages along this path. This path defines a forest structure for each token, with a node holding the token forming a root of the tree in the forest. Hence, for a system of K tokens, there exists K forest structures in the system. Requests are put on a token-queue, if the request has reached a site in CS or on a "node-queue" if it has reached a node that has requested the same token. Holding the requests in the node-queue helps to prevent unnecessary messages from being forwarded, and this reduces the average number of messages considerably.

The algorithm sends INFORM messages to attempt to minimize the distance of a node from a token. A node dynamically adapts to the load traffic by determining whether or not to send the INFORM messages. Under high load conditions, INFORM messages are not needed and therefore not sent. Under light load conditions, the INFORM messages help to reduce the distance from the token. Hence a node with an unused token will inform a few other nodes about the availability of a token. This reduces the average number of messages required per CS entry.

The simulation results have shown that our algorithm performs better under light

and heavy loads, both in terms of the average number of messages per CS entry and the average time to enter CS compared to Srimani, Raymond or Makki's algorithms. At high loads the average number of messages is approximately 2 messages per CS entry and the average delay is about N/K, where N is the number of nodes in the system and K is the number of tokens. The algorithm is implemented on a network of UNIX Sun Workstations. The implementation results support the simulation results and prove that the algorithm performs better from the standpoint of both message traffic and delay.

Our algorithm could be improved further by choosing a good heuristics to determine the nodes to which the INFORM messages are sent. This could improve the performance of the algorithm at light loads. It would also be interesting to evaluate the performance of our algorithm on parallel message passing machines such as nCube and Maspar.

REFERENCES

[1] M. Raynal, *Algorithms for Mutual Exclusion*. Cambridge, MA: MIT Press, 1st ed., 1986.

[2] G. Ricart and A. K. Agrawala, "An optimal algorithm for mutual exclusion in computer networks," *Comm. ACM*, vol. 24, no. 1, pp. 9–17, January 1981.

[3] M. Maekawa, "A $\sqrt{N}$ algorithm for mutual exclusion in decentralised systems," *ACM Trans. Comp. Syst.*, vol. 3, no. 2, pp. 145–159, May 1985.

[4] B. A. Sanders, "The information structure of distributed mutual exclusion algorithms," *ACM Trans. Comp. Syst.*, vol. 5, no. 3, pp. 284–299, August 1987.

[5] M. Singhal, "A dynamic information-structure mutual exclusion algorithm for distributed systems," in *International Conf. Distributed Computing Systems*, Newport Beach, CA, pp. 70–78, June 1989.

[6] I. Suzuki and T. Kasami, "A distributed mutual exclusion algorithm," *ACM Trans. Comp. Syst.*, vol. 3, no. 4, pp. 344–349, November 1985.

[7] G. Ricart and A. K. Agrawala, "Author's response to 'On mutual exclusion in computer networks' by Carvalho and Roucairol," *Comm. ACM*, vol. 26, no. 2, pp. 147–148, 1983.

[8] M. Singhal, "A heuristically-aided algorithm for mutual exclusion in distributed systems," *IEEE Trans. Computers*, vol. 38, no. 5, pp. 651–662, May 1989.

[9] M. Mizuno, M. L. Neilsen, and R. Rao, "A token based distributed mutual exclusion algorithm based on quorum agreements," in *International Conf. Distributed Computing Systems*, Arlington, TX, pp. 361–368, 1991.

[10] K. Makki, N. Pissinou, and Y. Yesha, "A new token based distributed mutual exclusion algorithm," in *International Conf. Distributed Computing Systems*, Pittsburgh, Pa, pp. 164–169, 1993.

[11] M. Trehel and M. Naimi, "A distributed algorithm for mutual exclusion based on data structures and fault tolerance," in *6th Annual International Phoenix Conference on Computers and Communications*, Scottdale, AZ, pp. 35–39, 1987.

[12] J. M. Bernabeu-Auban and M. Ahamad, "Applying path compression techniques to obtain an efficient distributed mutual exclusion algorithm," in *Lecture Notes in Computer Science*, vol. 392, pp. 33–44, 1989.

[13] D. Ginat, D. D. Sleator, and R. E. Tarjan, "A tight amortized bound for path reversal," *Information Processing Letters*, vol. 31, pp. 3–5, April 1989.

[14] K. Raymond, "A tree-based algorithm for distributed mutual exclusion," *ACM Trans. Comp. Syst.*, vol. 7, no. 1, pp. 61–77, February 1989.

[15] M. L. Neilsen and M. Mizuno, "A dag-based algorithm for distributed mutual exclusion," in *International Conf. Distributed Computing Systems*, Arlington, TX, pp. 354–360, 1991.

[16] T. Woo and R. Newman-Wolfe, "Huffman trees as a basis for a dynamic mutual exclusion algorithm for distributed systems," in *International Conf. Distributed Computing Systems*, Yokohama, Japan, pp. 126–133, June 1992.

[17] H. Koch, "An efficient replication protocol exploiting logical tree structure," in *Digest of papers: The* 23^{rd} *Int. Symp. Fault-Tolerant Comp.*, Toulouse, France, pp. 382–391, June 1993.

[18] S. Huang, J. Jiang, and Y. Kuo, "k-coteries for fault-tolerant k entries to a critical section," in *International Conf. Distributed Computing Systems*, Pittsburgh, Pa, pp. 74–81, 1993.

[19] K. Raymond, "A distributed algorithm for multiple entries to a critical section," *Information Processing Letters*, vol. 30, no. 4, pp. 189–193, February 1989.

[20] P. K. Srimani and R. L. Reddy, "Another distributed algorithm for multiple entries to a critical section," *Information Processing Letters*, vol. 41, no. 1, pp. 51–57, January 1992.

[21] K. Makki, P. Banta, K. Been, N. Pissinou, and E. Park, "A token based distributed k mutual exclusion algorithm," in *IEEE Proceedings of the Symposium on Parallel and Distributed Processing*, Arlington, TX, pp. 408–411, December 1992.

APPENDIX A

MODIFIED SRIMANI AND REDDY'S ALGORITHM

In order to bound the sequence numbers to any fixed positive integer L, Srimani and Reddy's algorithm used REPLY messages and PRIV_REPLY messages. Since the integer size is 32 bits, the sequence numbers will not overflow and reset to the starting value for the duration of our simulation. Hence, we did not use these messages. Consequently we do not need the RT and the LT arrays used in the original algorithm [20]. Some other changes were made to improve the algorithm. These changes are noted in the pseudo-code for the modified algorithm presented below. The original algorithm is presented in [20].

Procedure P1;

begin

requesting := TRUE;

if privilege_count = 0 then

begin

for all j in [1,2...,N] - [I] do

send REQUEST(I,RN[I]) to node j;

wait until privilege_count $\geq$ 1;

end;

P5;

PLN[I] := RN[I];

/**** not present in original algorithm *******/

/**** node's PLN array updated so that when it receives privilege message again, it can pass this most current information along with the token to the next

node on the token-queue*******/

critical section;

LN[I]:= RN[I];

P6;

if Q $\neq$ empty then

send PRIVILEGE(tail(Q), LN) to node head(Q)

else privilege_count := privilege_count +1;

end;

Procedure P2; (* Request(j,n) received; P2 is indivisible *)

begin

RN[j] := max(RN[j],n);

if PLN[j] + 1 ¡ RN[j] then PLN[j] := RN[j] - 1;

if (RN[j] = (LN[j] + 1) and (privilege_count $\geq$ 2

or (privilege_count = 1 and not(requesting))) then

begin

privilege_count := privilege_count - 1;

for all j in [1,2,...,N]-[I] do

LN[j] := max(LN[j], PLN[j]);

/****** The original code was LN[j] := PLN[j]; which was changed to the above to always make LN[j] the larger of the two values ********/

for all k in [1,2,...,N]-[I] do

if (in(Q,k) and RN[k] = LN[k] then

Q := delete(Q,k);

```
        send PRIVILEGE(Q,LN) to node j;
        PLN[j] := RN[j];
            /**** not present in original algorithm *******/
            /**** node's PLN array updated to reflect the most current information *******/
    end;
end;
```

Procedure P4; (* PRIVILEGE(Q,LN) is received; P4 is indivisible *)

```
begin
    if requesting and privilege_count = 0 then
        privilege_count := privilege_count + 1;
    else
    begin
        LN[I] := RN[I];
            /**** not present in original algorithm *******/
            /**** LN array of the token received is updated with this information before passing on the token to the next node on the token-queue *******/
        P6;
        if Q ≠ empty then
            send PRIVILEGE(tail(Q,LN)) to node head(Q);
            PLN[node head(Q)] := RN[node head(Q)];
              /**** not present in original algorithm *******/
              /**** node's PLN array updated to reflect the most current information *******/
        else privilege_count := privilege_count + 1;
```

```
    end;
end;

Procedure P5; (* Change requesting status; P5 is indivisible *)
begin
    privilege_count := privilege_count - 1;
    requesting := FALSE;
end;

Procedure P6; (* Update privilege messages; P6 is indivisible *)
begin
    for all j in [1,2,....,N]-[I] do
        begin
            if (LN[j] ≠ RN[j] - 1 and LN[j] ≠ RN[j])
            or (PLN[j] = RN[j])
                then LN[j] := PLN[j];
            else PLN[j] := LN[j];
            if not(in(Q,j)) and (RN[j] = (LN[j] + 1))
                then Q := append(Q,j);
            if (in(Q,j) and RN[j] = LN[j])
                then Q := delete(Q,j);
        end;
end;
```

VITA

Shailaja Gurupad Bulgannawar received her B.E. in Electronics in 1990 from University Visvesvaraya College of Engineering, Bangalore, India. She worked for a period of one year and six months as R&D Engineer with Center for Development of Telematics (C-DOT), India. She can be reached through Mr. G. N. Bulgannawar, 299, 100 Feet Road, 1st Stage, Indiranagar, Bangalore-560038, India.

The typist for this thesis was Shailaja Gurupad Bulgannawar.