A Distributed K-Mutual Exclusion Algorithm [*]

Shailaja Bulgannawar Nitin H. Vaidya[†]
Department of Computer Science
Texas A&M University
College Station, TX 77843-3112
E-mail: vaidya@cs.tamu.edu

Technical Report 94-066
November 1994

Abstract

This report presents a token-based K-mutual exclusion algorithm. The algorithm uses K tokens and a dynamic forest structure for each token. This structure is used to forward token requests. The algorithm is designed to minimize the number of messages and also the delay in entering the critical section, at low as well as high loads.

The report presents simulation results for the proposed algorithm and compares them with three other algorithms. Unlike previous work, our simulation model assumes that a finite (non-zero) overhead is encountered when a message is sent or received. The simulation results show that, as compared to other algorithms, the proposed algorithm achieves lower delay in entering critical section as well as lower number of messages, without a serious increase in the size of the messages.

[*]This report is an abbreviated version of [1]
[†]Direct all correspondence to Nitin Vaidya.

1 Introduction

This report presents a token-based algorithm for K-mutual exclusion in a distributed environment wherein different nodes (processes) communicate via message passing. The problem requires that at most K nodes be in a *critical section* (CS) at any given time. The proposed algorithm achieves this using K tokens; only a process in possession of a token may enter the critical section. Although there has been extensive research on distributed 1-mutual exclusion [13, 18, 4, 11, 10, 16, 15, 2, 3, 5, 6, 9, 7, 12, 17], research on distributed K-mutual exclusion ($K > 1$) is limited [19, 20, 21, 22].

Our approach for K-mutual exclusion is derived by improving and extending the 1-mutual exclusion algorithm by Trehel and Naimi [12]. The proposed algorithm is compared with three other distributed K-mutual exclusion algorithms [20, 21, 22]. It is shown that the proposed algorithm performs better than the existing algorithms under heavy as well as light load.

The report is organized as follows. Section 2 presents our algorithm for distributed K-mutual exclusion. Section 3 discusses the performance parameters. Section 4 presents a simulation model and Section 5 presents simulation results. The report concludes with Section 6.

2 Proposed Algorithm

The nodes (or processes) in the system are numbered 1 through N. There are K tokens in the system, numbered 1 through K. 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 first-in-first-out (FIFO) order on each channel. Initially, token t is possessed by node t, $1 \leq t \leq K$.

Each node maintains a `pointer` array with one entry for each token. These pointers define K forests corresponding to the K tokens, a forest being a collection of trees. By "t-th forest" we refer to the forest corresponding to token t formed by `pointer`[t] at each node. In each forest, the out-degree of a node is at most one, but the in-degree can be larger than one.

Note: Actually, the forest structure is sometimes violated by the formation of a cycle. However, as explained later, the cycles are formed temporarily under very specific circumstances, and they do not affect the correctness of the algorithm. Therefore, we will continue to use the term *forest* for the structure defined by the `pointer`s.

`pointer`[t] of node j contains identifier of the *parent* of node j in the t-th forest; `pointer`[t] at node j being equal to j means that node j is at the root of a tree in the t-th forest. Initially, `pointer`[t] for each node is set equal to t, $1 \leq t \leq K$. Thus, initially, each forest contains just one tree, with node t being at the root of the tree for token t. In general, for token t, the nodes waiting to receive token t and the node holding token t are at roots of the trees in the forest for token t.

Data structures maintained at each node:

- *token*: *boolean*; TRUE if the node has some token, FALSE otherwise.
- *token_id*: *integer*; indicates the identifier of the token, if a token is present at the node.
- *holding*: *boolean*; TRUE if the node is in the CS, FALSE otherwise.
- *waiting_for_token*: *integer*; indicates the identifier of the token that the node is waiting for.
- *pointer*: *array[1..K] of integer*; pointer[i] indicates the path towards token i.
- *node-queue*: *FIFO queue*; if a node A waiting for token t receives a a request of node B for token t, then identifier B is stored in the node-queue of node A.

Data structures associated with each token

Every token is associated with a data structure that is always sent with the token. 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.
- *request-modifier* tags: A tag is attached to each entry in the *token-queue*. The tag may often be NULL ($-$). The use of these tags will be clearer later. Figure 1 shows a typical token-queue and its associated tags.

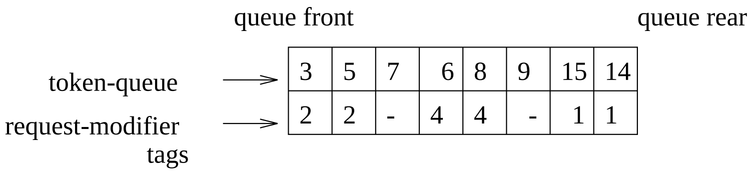


Figure 1: Token-queue for token 2 and associated request-modifier tags

Message types: Three types of messages are used by the proposed algorithm.

- REQUEST(Y, t) message: Indicates that node Y has requested token t. The request of a node for a token typically gets forwarded through a few nodes, the REQUEST message is used for this purpose. Thus, Y is not necessarily the sender of the REQUEST(Y, t) message, Y is the originator of the request.
- TOKEN(t) message: This message is used to send a token t and its associated data structures.
- INFORM(X, t) message: A node X sends this message to another node to inform them that X has token t.

2.1 Proposed algorithm

Pseudo-code for the algorithm is presented first, followed by a verbal explanation of the five procedures in the algorithm. Note that in the pseudo-code below, 'I' denotes identifier of the node executing the procedures. In the pseudo-code the comments are presented as /* comment */.

Procedure Entry_CS:

```
{
    if (token = TRUE) then
        holding = TRUE
    else
    {
        Choose token t using some heuristics;
        send REQUEST(I, t) to pointer[t];
        waiting_for_token := t;
        wait while holding ≠TRUE; /* procedure Handle_TOKEN sets holding = TRUE */
    }
    Enter Critical Section
}
```

Procedure Exit_CS:

```
/* Let t be the token possessed by this node */
{
    holding := FALSE;
    dest := First node on the token-queue;
    if (dest ≠ NULL) then
    {
        token := FALSE;
        if request-modifier of any node in token t's token-queue is NULL
        then pointer[t] := the last node on the token-queue whose
                   request-modifier tag is NULL.
        else pointer[t] := first node on the token-queue

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

Procedure Handle_REQUEST(Y,t):

```
            /* Y denotes the node where the request for token t originated */
{
    if (token = TRUE) and (holding = TRUE) then /* node I is in CS */
    {
```

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

Procedure Handle_TOKEN(t):

```
{
    if (waiting_for_token ≠ t) then
    {
        pointer[waiting_for_token] := tag for node I on token-queue;
        Append the node-queue to the token-queue of token t;
        For all the nodes that were in the node-queue, set their request-modifier
           tags equal to pointer[waiting_for_token];
    }
    else
    {
        Append the node-queue to the token-queue of token t;
        For all the nodes that were in the node-queue, set their tags equal to NULL;
    }
    de-queue node I and its tag from the token-queue;
    waiting_for_token := NULL;
    token := TRUE;
    token_id := t;
```

```
    pointer[t] := I;
    holding := TRUE;
}
```

Procedure Handle_INFORM(Y,t):

```
{
    if (waiting_for_token ≠ t)
        pointer[t] := Y;
}
```

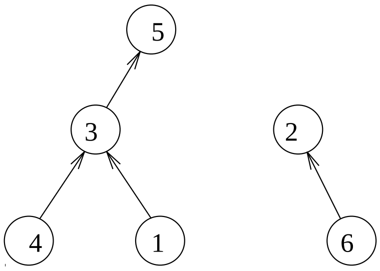
The procedures are explained below. To aid in the explanation, we first elaborate on the significance of the *request-modifier* tags associated with the token-queue entries. Ordinarily, a node that has requested token t eventually receives token t. However, if the request of a node, say A, arrives at some node B that possesses token u ($u \neq t$), then node B adds A's request to the token-queue of token u. This essentially modifies node A's request for token t into a request for token u. The fact that node B modified the request is recorded by setting the *request-modifier* tag for node A's entry in the token-queue equal to B. This information is used by node A (when it receives token u) to maintain the forest structure for token t (i.e., to avoid creation of cycles in the t-th forest).

There are five procedures in all. Entry_CS and Exit_CS are called when a node wants to enter or exit the critical section, respectively. The remaining three procedures are message handlers for the three types of messages.

Entry_CS: This procedure is invoked by node I when it wants to enter the critical section (CS). If node I has a token then it can enter CS without any delay. Otherwise, using some heuristic, it chooses a token t ($1 \leq t \leq K$), and sends a request for token t to its parent in the t-th forest (i.e., pointer[t]). In Figure 2, if node 4 wants to request token 1, it sends REQUEST(4,1) message to node 3.

Exit_CS: This procedure is executed when a node exits the CS. Assume that node I has token t.

If the token-queue is empty, then node I continues to possess token t, but sends INFORM(I, t) messages to any ν nodes (where ν is a design parameter). INFORM messages



node 1: pointer[1] = 3
node 2: pointer[1] = 2
node 3: pointer[1] = 5
node 4: pointer[1] = 3
node 5: pointer[1] = 5
node 6: pointer[1] = 2

Figure 2: Example: Forest structure for token 1

are useful to reduce the distance of a node from a token (in its forest).

If the token-queue is not empty when node I exits the critical section, I sends the token to the node, say A, at the head of token t's *token-queue*. The pointer[t] of node I is modified appropriately to ensure that the forest structure is maintained. Specifically, pointer[t] 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 A, which is at the head of the token-queue.

Example: If node I has token 2 with the token-queue shown in Figure 1, then it sends the token 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 have set pointer[2] equal to 3 (the node at the head of the token-queue).

Handle_INFORM(Y, t): This procedure is executed when the node receives an INFORM message from some node Y. This message implies that node Y possessed token t at the time the message was sent. In response to this message, pointer[t] is set equal to Y. INFORM messages help reduce the distance of a node from a token.

Handle_TOKEN(t): This procedure is executed when a node receives a token t, i.e., receives TOKEN(t) message. 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: Node I had requested token t: (In this case, the tag of node I in the token-queue of token t is guaranteed to be NULL.) Node I sets pointer[t] equal to I to indicate that I

is now the root of a tree, and appends the requests in its node-queue to t's token-queue, setting their tags equal to NULL.

Case 2: Node I had requested token u ($u \neq t$): Here, the *request-modifier* tag (say A) of node I in token t's token-queue indicates that the request of node I was modified by node A. In this case, the requests waiting in the node-queue of node I are also considered to have been effectively modified by node A. Therefore, in this case, node I appends its node-queue to token t's token-queue, and sets the tags (for the newly added nodes) equal to A.

To maintain the forest structures for token t, node I sets pointer[t] equal to A (the node that modified I's request).

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

Handle_REQUEST(Y, t): When a node receives a request for a token this procedure is invoked. Y is the node that is requesting token t. The REQUEST message may be sent by Y, or may have been forwarded by some other node. 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(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 2, if node 3 receives a request from node 4 for token 1, node 3 forwards the request to node 5 (as at node 3, `pointer`[1] = 5) and changes `pointer`[1] to 4.

Case 3: node I possesses a token u (u may or may not be equal to t) and is not in the CS. This implies that the token-queue of u is empty.

Node I sets pointer[t] to Y and adds node Y to the token-queue. The *request-modifier* tag of Y is set equal to I if $t \neq u$, else the tag is set to NULL.

Case 4: node I possesses a token and is in the CS. Let the token possessed by I be u.

Node Y is added to the token-queue of token u. The *request-modifier* tag for node Y is set to I if $t \neq u$, else set to NULL.

Proof of Correctness

We present a brief sketch of the proof here [1].

K-Mutual exclusion

A node can be in critical section only if it has a token. As there are only K tokens, at most K nodes can be in the critical section at a time. Hence, K-mutual exclusion is achieved.

Deadlock free

The proof of freedom from deadlock is in two parts: First, we show that for a deadlock to occur, it should be possible for a request from some node X to travels back to node X. Next, we show that this situation is impossible with the given algorithm.

Part I: We say that node A is *blocked* at node B, if the request of node A for some token is in the node-queue of node B. Consider a graph formed by drawing an arc from node A to node B, if node A is blocked at node B. For a deadlock to occur, this graph must contain a cycle. (Note that each node can have only one outgoing arc, as a node can have at most one outstanding request at any time.) Assume that nodes X1, X2, ... , Xm form a cycle, such that X1 is blocked at X2, X2 is blocked at X3, ..., and Xm is blocked at X1. Note that at node X1, some nodes other than Xm may also be blocked. Similarly, some nodes may be blocked at X2, X3, ... , Xm. Consider the set S of nodes containing X1, ... , Xm and all the nodes blocked at X1, ..., Xm. Consider a scenario that is similar to the deadlocked scenario, except that nodes in set S$-\{$X1$\}$ do not make their last request that is blocked in the deadlocked scenario. In this modified scenario, the REQUEST from X1 will reach X2 as before. However, now instead of blocking the request, X2 will forward the request, the REQUEST will then reach X3 (possibly through some intermediate nodes). Applying this argument repetitively, the REQUEST will reach nodes X4, X5,... , Xm and subsequently back to X1.

The above implies that, for a deadlock to occur, the algorithm must allow a request from some node X1 to return to itself.

Part II: For the above REQUEST to return to its originator, a cycle must exist in the structure formed by the `pointer`s. The K-mutual exclusion algorithm modifies the `pointer`s at many places. It can be easily shown that only one modification made to the `pointer`s can create a cycle. We will discuss only this modification to the `pointer`s.

Specifically, when the request of a node A for token t is modified by some node B by adding A to the token-queue of token u ($u \neq t$), a cycle can be created in the structure formed by pointer[u]. Originally, paths may exist from node A to node B in the t-th forest as well as the u-th forest. The request from A for token t travels the links in the t-th forest. Node A's request is added to u's token-queue at node B. If node B, on exiting from its critical section, finds that no node on its token-queue has its request-modifier tag NULL and node A is the first node on the token-queue, then node B will send token u to node A and set `pointer`[u] = A. As a path already exists from A to B in the u-th forest, a cycle is now formed. This cycle is broken as soon as token u reaches node A.

Now observe that the REQUEST from node A above cannot return to node A itself because the request is added by node B to the token-queue of token u. Thus, the condition necessary for deadlock (as stated in part I above) cannot occur. Therefore, deadlock cannot occur.

Starvation free

When a request from a node Y for token t is forwarded by some node Z, node Z sets its `pointer`[t] equal to Y. This implies that, in the absence of a cycle, a request can visit a node at most once. When a cycle, as described above, is formed, a request may visit a node at most twice. This property, along with the deadlock free property, guarantees that a node that has sent a request will eventually receive a token.

3 Performance

The performance parameters of interest 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 existing papers on K-mutual exclusion typically present an analytical estimate of the average number of messages required per CS entry. The average number of messages is inadequate to measure the algorithm performance, because (as shown later) an algorithm that requires small number of messages may result in large delays in entering the CS. In this report, we present simulation results rather than analysis.

Under light load (i.e., small λ), there is a good chance that the token-queue will be empty when a node, say A, exits from the critical section. Whenever the token-queue is empty, the Exit_CS procedure informs ν nodes that node A has a token, say t. This reduces the average delay in entering the critical section at the cost of ν extra messages. This also reduces the average distance of a node from from token t, which in turn results in a reduction in the average number of messages required per CS entry. The net effect of the INFORM messages is often to reduce the average number of messages required.

Under high 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 does not send the INFORM messages. Thus the algorithm improves the performance by sending INFORM messages only when beneficial.

Updates made to *pointer*s in Exit_CS and Handle_REQUEST are also designed to reduce the distance between the nodes and the tokens. This in turn results in smaller CS entry delays and smaller number of messages.

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 the 1-mutual exclusion algorithm by Trehel and Naimi [12].

Hence, unnecessary message transmission is avoided. This reduces the average number of messages.

Heuristics for choosing a token in Entry_CS

Performance of the algorithm is dependent on the decision mechanism used by each node to decide which token to send the request for (in Entry_CS). One possibility is to choose the token randomly. The other possibility is to use a heuristic to choose a token that is likely to be reached with a small number of hops. The 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.

4 Simulation Model

The simulation model used here is a refinement of the model presented by Singhal [9]. 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 called the *rate* of arrival of CS requests. 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. If the same message is sent simultaneously to multiple destinations (multicast), a cost of T_s is encountered for each message copy. This assumption holds on many present implementations.

The simulation model presented by Singhal [9] 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 significantly and cannot be neglected.

5 Simulation Results

We simulated our algorithm and compared it with three other K-mutual exclusion algorithms proposed by Raymond [20], Srimani and Reddy [21] and Makki et al. [22].

We modified the algorithm by Srimani and Reddy [21] to improve its performance. The original algorithm assumes finite counters. By removing that restriction, we reduce the number of messages required by their algorithm.

For T_r, $T_s \neq 0$, Makki's algorithm [22] does not work correctly as such. We simulated a slightly modified version that yields optimistic results for Makki's algorithm when T_r, $T_s \neq 0$. In particular, Makki's algorithm assumes that time required for a message to reach its destination and to receive the response takes $2T_t$ time units. This is true when $T_r = T_s = 0$, and not valid when T_r and T_s are non-zero. In such situations, we "accelerate" the response messages to reach within $2T_t$, resulting in optimistic estimates of CS entry delay and number of messages. Any adaptation of [22] that will work correctly for non-zero T_r and T_s will perform worse than what our results indicate. The results presented for $T_s = T_r = 0$ are obtained by simulating the original algorithm by Makki.

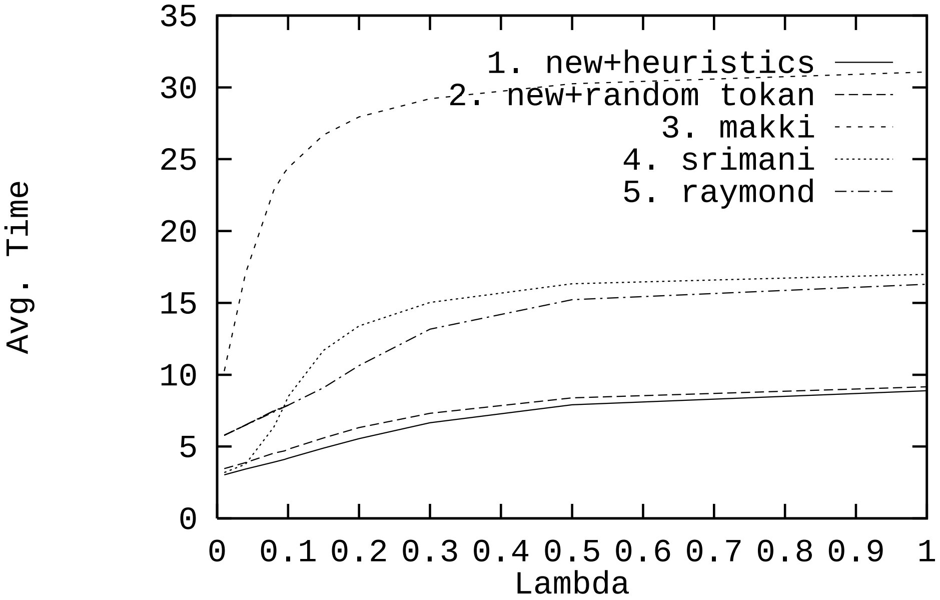
Simulations were carried out for a system of thirty nodes ($N = 30$) and three tokens ($K = 3$), for various values of λ. The number of nodes ν to which INFORM messages are sent was fixed at 2. We simulated using various values of T_s, T_t, T_r and E. For various non-zero T_s and T_r, the result trends were similar, therefore we present only one set of results. Similarly, result trends for different values of E were similar, so we present results only for one value of E. Specifically, results are presented for $T_s = T_r = 0.1$, $T_t = 0.8$, $E = 0.0002$.

$E = 0.0002$ is identical to that used by Singhal [9]. (Results for larger E are also similar [1].) For comparison, some results for $T_s = T_r = 0$, $T_t = 1$, $E = 0.0002$ are also presented. (This set of parameters implies that all the message communication delay is encountered in transmission alone.).

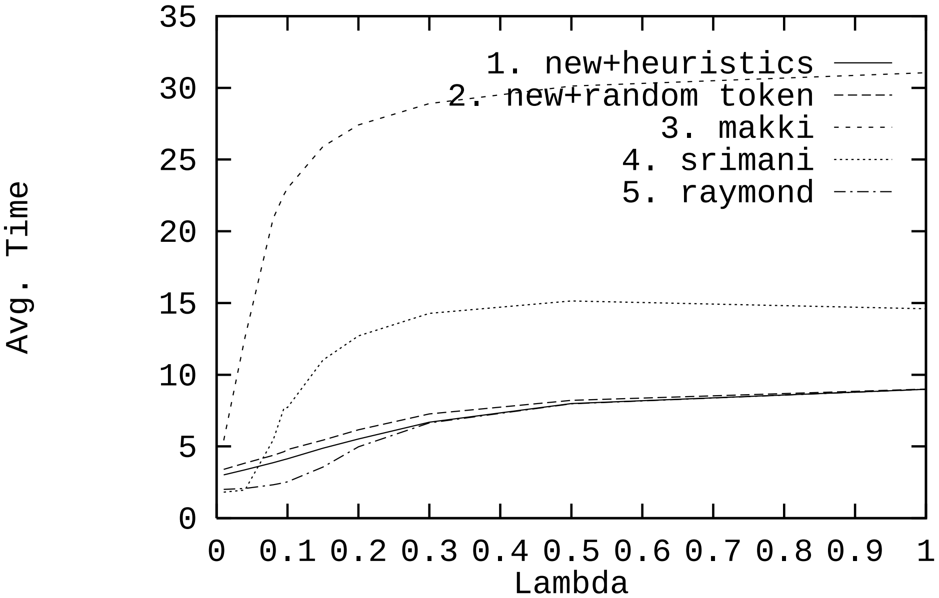
The simulations were performed for 5000 critical section entries. This number was chosen because we observed that the results of the simulation converged by 5000 entries into the critical section.

Figure 3(a) shows the *average time* taken to enter the critical section, by the four different algorithms for $T_r = T_s = 0.1$ and $T_t = 0.8$. In the graph, 'new+heuristic' refers to our algorithm with the heuristic in the previous section and 'new+random token' refers to our algorithm where a token is chosen randomly (in Entry_CS procedure). Our algorithm performed better than the other algorithms for most values of λ. (The heuristic has improved the performance by only a small amount.) 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. The curve gradually flatten out for greater values of λ. The intuitive reasoning is as follows: The rate at which each node enters CS is identical (on average). The message communication delay is 1 unit time ($T_s + T_t + T_r$), therefore, sending the token from one node to another requires 1 unit time. Therefore, the maximum rate at which a node can enter CS (with $K = 3$ and $N = 30$) is upper bounded by $1/10$, independent of the value of λ. The delay in entering the CS is determined by the rate at which the nodes enter CS (which is almost independent of λ when λ is large). Therefore, for large λ, the curve flattens ("system saturation").

Figure 3(b) shows the *average time* taken to enter the critical section, for $T_r = T_s = 0$ and $T_t = 0.8$. Observe that here Raymond's algorithm [20] performs better than us for small λ and equally well for large λ. When a node wants to enter CS, Raymond's algorithm sends multiple request messages in parallel to other nodes. When $T_s = 0$, the overhead of sending all these messages is zero (for the sender). When T_s is non-zero, the overhead of sending multiple messages can be substantial. Therefore, Raymond algorithm performs well when $T_s = 0$, but performs poorly with the realistic assumption that $T_s \neq 0$.



(a) $T_r = T_s = 0.1$ and $T_t = 0.8$



(b) $T_r = T_s = 0$ and $T_t = 1.0$

Figure 3: Average time to enter the critical section

The time to enter the critical section is maximum for Makki's algorithm [22]. This algorithm uses a RELEASE message to maintain the correctness of the algorithm. At high load, the RELEASE message is propagated through all other nodes before reaching the same node again. This causes the system to behave similar to a system with a single token, resulting in significant delays.

Figure 4 plots the average number of messages required per CS entry versus λ. Our

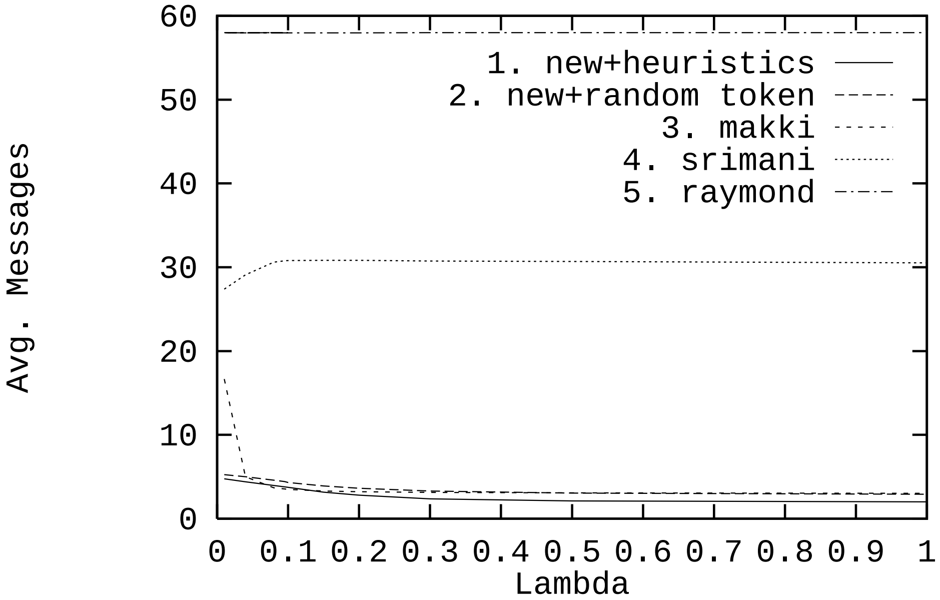


Figure 4: Average number of messages per critical section entry for $T_r = 0.1$, $T_s = 0.1$ and $T_t = 0.8$

algorithm requires smaller number of messages compared to the other algorithms, for most values of λ. Applying the heuristics for choosing a token has reduced the number of messages at high load.

Raymond's algorithm has a lower bound of $2N - K - 1$ on number of messages required and an upper bound of $2 * (N - 1)$ [20]. 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. Their analysis suggests that the average number of messages per critical section entry is close to $(N - 1)$ [21]. 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.

For Makki's algorithm [22] at low load, the number of messages required is quite large. The number of messages required becomes smaller with increasing λ, with only three messages being required at heavy load. Although the number of messages required is small, as seen before, with large λ, Makki's algorithm results in longer delays. (This shows that number of messages, by itself, is inadequate to evaluate algorithm performance.)

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 suggests that the average number of messages per critical section entry is not affected by the sending and receiving times (T_s and T_r).

Figure 5 plots the *average information* that is passed in the messages by various algorithms. The information content of a messages was calculated by taking into account all the fields of the message. For example, the TOKEN message contains the token identifier, token-queue, request-modifier tags, and message source. As in most implementations, each message, by default, contains message source, destination and message type. The *average information* for our algorithm is same with and without the heuristic. When $\lambda = 1$, the average information is about 9 words for our algorithm, 4 words for Raymond's algorithm, 6.5 words for Srimani's algorithm, and 8 words for Makki's algorithm. All messages in Raymond's algorithm are of the same size (4 words), therefore, that curve is simply a horizontal line. For our algorithms the average message size is about 1.5 to 2 times larger than the other algorithms. By sending more information in each message, our algorithm reduces the number of messages. As the messages are still quite small, the overhead is proportional to the number of messages, and quite independent of the size of the message.

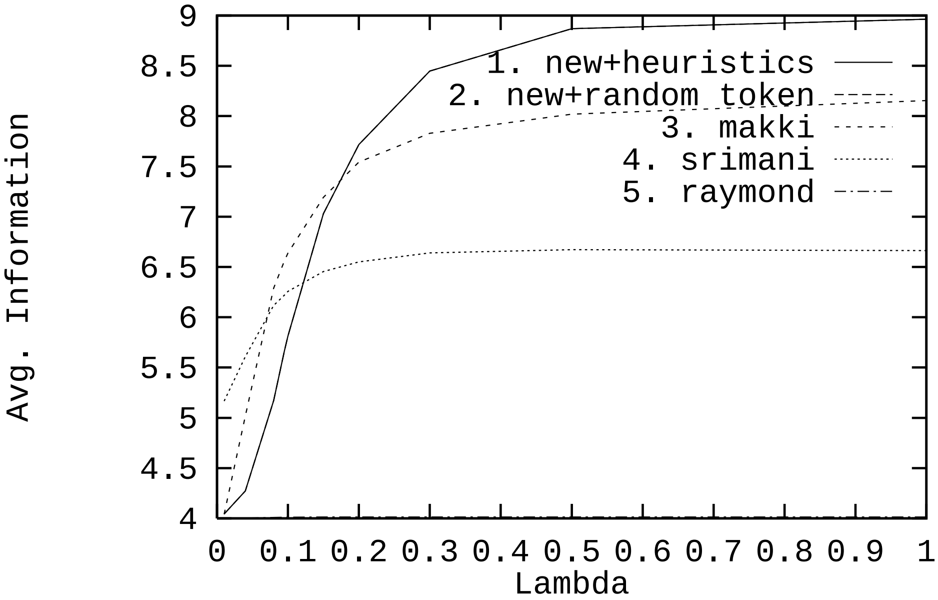


Figure 5: Average information (in words) per message for $T_r = 0.1$, $T_s = 0.1$ and $T_t = 0.8$

Sanity Checks: One *sanity check* of our simulations was to verify that the average number of messages from the simulation results matched approximately with that obtained by an analysis of some of the algorithms. Another check was to verify that all the nodes in the system entered the critical section roughly equal number of times and the average delay for entering CS for every node was comparable.

6 Conclusions

This report presents a token-based K-mutual exclusion algorithm. The algorithm uses K tokens and a dynamic forest structure for each token. This structure is used to forward token requests. The algorithm is designed to minimize the number of messages and also the delay in entering the critical section, at low as well as high loads.

The report presented simulation results for our algorithm and compared them with three other algorithms. Unlike previous work, our simulation model assumes that a finite (non-zero) overhead is encountered when a message is sent or received. The simulation results show that, as compared to other algorithms, the proposed algorithm achieves lower delay in entering CS as well as lower number of messages, without a serious increase in the size of the messages.

References

[1] S. Bulgannawar, *A Distributed K-Mutual Exclusion Algorithm*, M. S. Thesis, Dept. of Electrical Eng., Texas A&M University, August 1994.

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

[3] G. Ricart and A. K. Agrawala, "An optimal algorithm for mutual exclusion in computer networks," *Comm. ACM*, vol. 24, pp. 9–17, January 1981.

[4] M. Maekawa, "A $\sqrt{N}$ algorithm for mutual exclusion in decentralised systems," *ACM Trans. Comp. Syst.*, vol. 3, pp. 145–159, May 1985.

[5] B. A. Sanders, "The information structure of distributed mutual exclusion algorithms," *ACM Trans. Comp. Syst.*, vol. 5, pp. 284–299, August 1987.

[6] 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.

[7] I. Suzuki and T. Kasami, "A distributed mutual exclusion algorithm," *ACM Trans. Comp. Syst.*, vol. 3, pp. 344–349, November 1985.

[8] 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.

[9] M. Singhal, "A heuristically-aided algorithm for mutual exclusion in distributed systems," *IEEE Trans. Computers*, vol. 38, pp. 651–662, May 1989.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] K. Raymond, "A tree-based algorithm for distributed mutual exclusion," *ACM Trans. Comp. Syst.*, vol. 7, pp. 61–77, February 1989.

[16] 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.

[17] T. K. 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.

[18] 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.

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

[20] K. Raymond, "A distributed algorithm for multiple enteries to a critical section," *Information Processing Letters*, vol. 30, pp. 189–193, February 1989.

[21] P. K. Srimani and R. L. Reddy, "Another distributed algorithm for multiple enteries to a critical section," *Information Processing Letters*, vol. 41, pp. 51–57, January 1992.

[22] 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.