A Distributed K-Mutual Exclusion Algorithm

Shailaja Bulgannawar Nitin H. Vaidya
Department of Computer Science
Texas A&M University
College Station, TX 77843-3112

Abstract

This paper 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 expected to minimize the number of messages and also the delay in entering the critical section, at low as well as high loads.

The paper 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 significant increase in the size of the messages.

1 Introduction

This paper 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 (CS). Although there has been extensive research on distributed 1-mutual exclusion (e.g., [7, 3, 4, 5, 6]), research on distributed K-mutual exclusion ($K > 1$) is limited [9, 10, 11, 12]. The objective of this paper is to motivate further research on K-mutual exclusion, by demonstrating existence of algorithms that can perform better than the existing ones.

Our approach for K-mutual exclusion is derived by extending the 1-mutual exclusion algorithm by Trehel and Naimi [6]. Simulation results for the proposed algorithm are compared with three other distributed K-mutual exclusion algorithms [10, 11, 12]. Using simulations, we show that the proposed algorithm performs better than the existing algorithms under heavy as well as light load.

In Section 5 we briefly discuss a "partitioning" approach, for K-mutual exclusion using 1-mutual exclusion algorithms, that performs better than the existing K-mutual exclusion algorithms.

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. The nodes are assumed to be reliable and fully connected. The network is reliable and delivers 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 *temporarily* violated by the formation of a cycle (as explained later). However, these cycles do not affect the correctness of the algorithm. Therefore, we will continue to use the term *forest* for the structure defined by the `pointers`.

`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. Each node maintains the following data structures:

- *token_id*: indicates the identifier of the token if a token is present at the node, NULL otherwise.
- *holding*: *boolean*; TRUE if the node is in the CS, FALSE otherwise.
- *waiting_for_token*: *integer* or NULL; 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 request of node B for token t, then identifier B is stored in the node-queue of node A.

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: *integer* or *NULL* (−); 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.

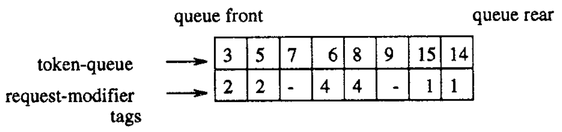


Figure 1: An example of a token-queue

For future reference, define `last-unmodified` node in a non-empty token-queue as follows: If the token-queue contains at least one node whose request-modifier is null, then the last node in the queue whose request-modifier is null is the `last-unmodified` node. Else, the first node on the token-queue is defined to be the `last-unmodified` node. For instance, in Figure 1, the `last-unmodified` node is node 9. If, in Figure 1, request-modifier tags for nodes 7 and 9 were not null, then the `last-unmodified` node would have been 3.

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.

Pseudo-code for the algorithm in a C-like language is presented first, followed by a verbal explanation of 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 */. 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.

```
Procedure Entry_CS:
{
    if (token_id ≠ NULL) then /* node I has a token */
        holding := TRUE
    else {
        Choose token t using some heuristics;
        send REQUEST(I, t) to pointer[t];
        waiting_for_token := t;
        wait until a TOKEN is received;
    }
    Enter Critical Section
}
```

```
Procedure Exit_CS:
/* Let t be the token possessed by this node */
{
holding := FALSE;
if token-queue of token t is not empty {
    token_id := NULL;
    pointer[t] := last-unmodified node on token-queue
    send TOKEN(t) to the first node on token-queue;
}
else send INFORM(I, t) message to ν nodes;
}
```

```
Procedure Handle_REQUEST(Y,t):
    /* the request for token t originated at node Y */
{
if (token_id ≠ NULL) then
{ /* node I has a token */
```

```
        enqueue Y into the token-queue;
        if (token_id $\neq$ t) then
            request-modifier tag of Y := I;
        else
            request-modifier tag of Y := NULL;
        if (holding = FALSE) then
        { /* node I has a token but is not in CS */
            send TOKEN(token_id) to Y;
            pointer[token_id] := Y;
            token_id := NULL;
        }
    }
    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):

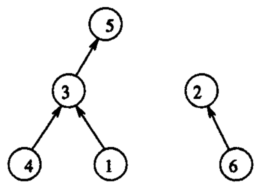
```
{
Append node-queue to token-queue of token t;
if (waiting_for_token $\neq$ t) then
{ /* node I had requested some other token */
    pointer[waiting_for_token] := request-modifier for I;
    For all the nodes that were in the node-queue,
        set their request-modifier tags equal to
        the request-modifier for I;
}
else
{ /* node I had requested 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_id := t;
pointer[t] := I;
holding := TRUE;
}
```

Procedure Handle_INFORM(Y,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).

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.



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

Exit_CS: Assume that node I has token t. If the token-queue of token t is empty, then node I continues to possess token t, but sends INFORM(I, t) messages to ν nodes (where ν is a design parameter). INFORM messages can be useful to reduce the distance of a node from a token.

If the token-queue is not empty when node I exits the critical section, I sends the token to the node at the head of token t's *token-queue*. The pointer[t] of node I is set to `last-unmodified` node to ensure that the forest structure is maintained. For 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.

Handle_INFORM(Y, t): The INFORM message indicates 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 can help reduce the distance of a node from a token.

Handle_TOKEN(t): This procedure is executed when a node receives token t. 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 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).

Example: If node 3 receives token 2 with token-queue in Figure 1, but had requested token 1, node 3 sets pointer[1] to 2, which is its tag in the token-queue. *Case 2:* 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 appends its node-queue to t's token-queue, setting their tags equal to NULL.

Handle_REQUEST(Y,t): This procedure is invoked when node I receives a request for a token. Y is the node that is requesting token t.
Case 1: node I possesses a token u (u may or may not be equal to t). In this case, node I adds Y to the token-queue of token u. The request-modifier tag for Y is set to I if $t \neq u$, and NULL otherwise. If node I is not in the critical section, then it sets pointer[t]=Y, and sends the token to node Y.
Case 2: node I does not possess a token and has requested token t. In this case, Y is stored in the node-queue.
Case 3: node I does not possess a token and has not requested token t. Node I forwards the request to pointer[t] and sets pointer[t]=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.

A proof of correctness is sketched in [2].

Temporary cycles: 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.

Comparison with Trehel and Naimi [6] The above algorithm is based on [6], but differs in three ways: (1) Proposed algorithm maintains explicit queues, instead of implicit distributed queues in [6]. (2) Proposed algorithm allows multiple tokens. A node may request any token, and may possibly receive a token different from that requested. (3) The algorithm allows for INFORM messages.

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 paper, we present simulation results rather than analysis.

Under light load (i.e., small request rate), there is a good chance that the token-queue will be empty when a node, say I, exits from the critical section. Whenever the token-queue is empty, the Exit_CS procedure informs ν nodes that node I has a token, say t. This may reduce the average distance of a node from token t (at the cost of ν extra messages). The net effect could be a reduction in the average number of messages required per CS entry.

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.

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 [6]. Hence, unnecessary message transmission is avoided. This reduces the average number of message, at the cost of an increase in the message size.

Decision Mechanisms

The performance of the above algorithm depends on three decision mechanisms for choosing:

- the token requested in Entry_CS.

- the number (ν) of INFORM messages sent in Exit_CS.
- the destination nodes for the INFORM messages.

In our simulations, somewhat arbitrarily, ν is fixed at 2. This is not necessarily optimal (e.g., $\nu = 0$ may be optimal in some cases). Also, instead of fixing ν, one may want to vary ν dynamically to adapt the algorithm to a given system. In our simulations, the destination of INFORM messages are chosen randomly. A good heuristic for choosing these destinations can potentially improve performance.

Heuristics for choosing a token 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 either the node recently received token t, or the node recently received an INFORM message from a node possessing token t. If a node I remembers that it had last seen the token t and makes a request for that token, there may be a better chance of node I's request reaching the token t with a small number of hops. Other heuristics for choosing a token can also be conceived.

Choice of Decision Mechanisms

The algorithm performance is dependent on the choice of three decision mechanisms discussed above. Our intent in this paper is to show that there *exists* an instantiation of our algorithm that performs better than other algorithms – no attempt is made to determine the optimal instantiation. The choice of decision mechanisms used in this paper is essentially arbitrary.

For instance, INFORM messages are useful only if sending ν INFORM messages will reduce the number of other messages by at least ν. This paper assumes $\nu = 2$, however, $\nu = 2$ is not necessarily optimal. It is quite possible that, in some systems, INFORM messages will not help to reduce the total number of messages – in such a case ν should be chosen to be 0. Although we believe that INFORM messages may be useful to reduce the aggregate number of message (and, possibly, synchronization delay), this paper does not explore the impact of INFORM messages on algorithm performance.

4 Simulation

The simulation model used here is a refinement of the model presented by Singhal [5]. 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 denoted by E. Each node spends T_s time units when sending a message (time spent in the network layer software). 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, a cost of T_s is encountered for each message copy. This assumption should apply to a system that does not provide a hardware multicast facility.

The simulation model presented by Singhal [5] 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, with high-speed networks, the time spent in the network layer may not be negligible as compared to the transmission delay, therefore, T_s and T_r cannot be ignored.

We simulated our algorithm and compared it with three other K-mutual exclusion algorithms proposed by Raymond [10], Srimani and Reddy [11] and Makki et al. [12]. The algorithm by Srimani and Reddy [11] assumes finite counters for numbering requests. We removed this restriction by allowing infinite counters, somewhat reducing the number of messages required by their algorithm. For T_r, $T_s \neq 0$, Makki's algorithm [12] 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 [12] that will work correctly for non-zero T_r and T_s is expected to 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 ($K = 3$). ν 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 [5]. (Results for $E = 1$ are also similar [1].) For comparison, some results for $T_s = T_r = 0$, $T_t = 1$ are also presented. (This set of parameters implies that all the message communication delay is encountered in transmission alone.).

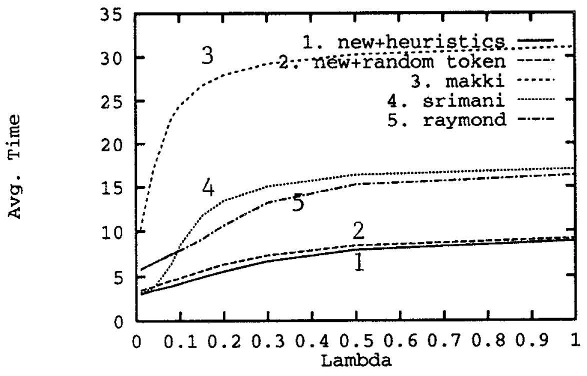
The simulations were performed for 5000 critical section entries. This number was chosen because we observed that the results of the simulation converged by 5000 entries into the critical section.

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+heuristics' 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. Our algorithm performed better than 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 curves gradually flatten out for greater values of λ ("saturation").

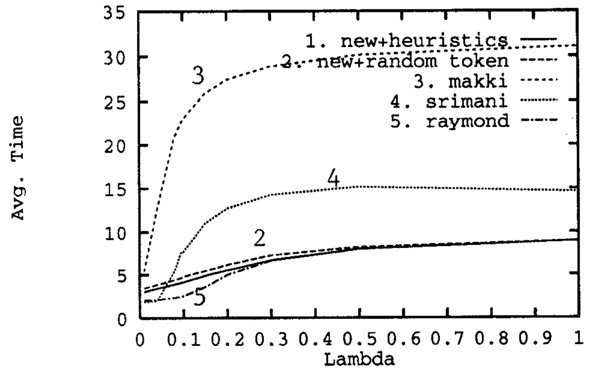
Figure 3(b) shows the *average time* taken to enter the critical section, for $T_r = T_s = 0$ and $T_t = 1.0$. Observe that here Raymond's algorithm [10] performs better than us for small λ and equally well for large λ. When a node wants to enter CS, Raymond's algorithm sends multiple request messages 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's algorithm performs well when $T_s = 0$, but performs poorly with the realistic assumption that $T_s \neq 0$.

The time to enter the critical section is maximum for Makki's algorithm [12]. This algorithm uses a token message to maintain the correctness of the algorithm. At high load, the token 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. (A simple modification to this algorithm can improve its performance [1].)

Figure 4 plots the average number of messages required per CS entry versus λ. Our algorithm requires smaller number of messages compared to the other



(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

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 per CS entry, and an upper bound of $2 * (N - 1)$ [10]. Srimani's analysis suggests that, for their algorithm, the average number of messages per critical section entry is close to $(N - 1)$ [11]. Our simulation results agree with these analytical results.

For Makki's algorithm [12] 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

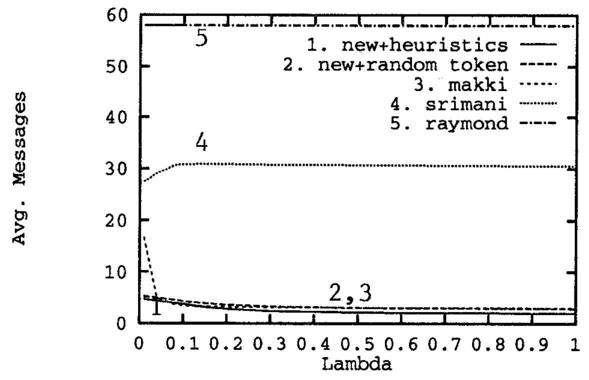


Figure 4: Average number of messages per critical section entry for $T_r = 0.1$, $T_s = 0.1$ and $T_t = 0.8$

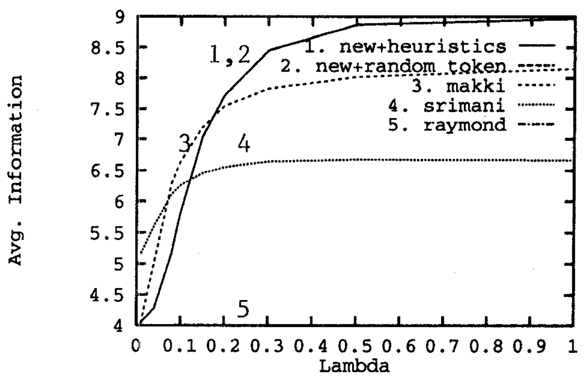


Figure 5: Average information (in words) per message for $T_r = 0.1$, $T_s = 0.1$ and $T_t = 0.8$

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.)

Measurements for the average number of messages for $E = 1$, and also for $T_r = 0.05$, $T_s = 0.05$, $T_t = 0.9$ and $T_r = 0$, $T_s = 0$, $T_t = 1.0$, indicate that the number of messages is practically the same for all the cases [1]. This suggests that the average number of messages per critical section entry is not affected significantly by T_s, T_r and E.

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. (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. All messages, except TOKEN, are fixed size. The size of a TOKEN message can increase at most linearly with the number of nodes. Therefore, we expect that the increased average message size for our algorithm (as compared to other algorithms) will not be a serious limitation for systems with a larger number of nodes.

Another interesting performance metric is the "average information transmitted" per CS entry. This metric would be a measure of "bandwidth" consumed by the algorithm for each CS entry. Due to lack of space, we do not present measurements of average information transmitted per CS entry.

5 System Partitioning

Consider a K-mutual exclusion algorithm A_k. One way to achieve K-mutual exclusion is to use algorithm A_k for the N nodes. Another possibility is to partition the system into K clusters containing approximately N/K nodes each, and executing algorithm A_1 (i.e., A_K with $K = 1$) within each cluster. While such a "partitioning" approach may appear inefficient at first, surprisingly, the partioning approach performs quite well for the three previous algorithms. For our algorithm, partitioning approach does not perform better than the K-mutual exclusion algorithm [1].

For instance, the algorithm in [11] requires approximately $N-1$ messages per CS entry (for K-mutual exclusion). Clearly, if the partitioning approach is used, then the number of messages per CS entry will reduce to approximately $(N/K) - 1$, a significant improvement (for $K > 1$). (This improvement is often accompanied by an improvement in the average time to enter the CS [1].) Similarly, the average number of messages (per CS entry) for the K-mutual exclusion algorithm in [10] is approximately $2(N - 1)$. Again, the average number can be reduced to approximately $2(N/K - 1)$ by using the partitioning scheme.

These results suggest that superiority of a K-mutual exclusion algorithm should be demonstrated by comparing it not only with other K-mutual exclusion algorithms, but also with the "partitioning" scheme. This paper, however, neglects to present such a comparison, as the K-mutual exclusion algorithm is *not* yet optimized. Optimization requires proper choice of the three decision mechanisms described earlier. This problem is a subject of future work.

6 Summary

The previous research on distributed K-mutual exclusion is limited. The objective of this paper is to motivate further research on K-mutual exclusion, by demonstrating existence of algorithms that can perform better than the existing ones.

The paper presents a token-based K-mutual exclusion algorithm and compares simulation results for the proposed algorithm with three other algorithms. Unlike previous work, the 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. Future work includes algorithm optimization, and evaluation of the algorithm in "non-homogeneous" environments where all nodes are not identical.

Acknowledgements

We thank the referees for helpful comments.

References

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

[2] S. Bulgannawar and N. H. Vaidya, "A distributed k-mutual exclusion algorithm," Tech. Rep. 94-066, Computer Science Department, Texas A&M University, College Station, November 1994.

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

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

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

[6] 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*, pp. 35–39, 1987.

[7] 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.

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

[9] 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.

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

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

[12] 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*, pp. 408–411, December 1992.