A New Token Passing Distributed Mutual Exclusion Algorithm

Sujata Banerjee
Telecommunications Program
University of Pittsburgh
Pittsburgh, PA 15260

*Panos K. Chrysanthis**
Computer Science Department
University of Pittsburgh
Pittsburgh, PA 15260

Abstract

Eliminating interference between concurrently executing activities through mutual exclusion is one of the most fundamental problems in computer systems. The problem of mutual exclusion in a distributed system is especially interesting owing to the lack of global knowledge in the presence of variable communication delays. In this paper, a new token-based distributed mutual exclusion algorithm is proposed. The algorithm incurs approximately three messages at high loads, irrespective of the number of nodes N *in the system. At low loads, it requires approximately* N *messages. The paper also addresses failure recovery issues, such as token loss.*

1 Introduction

It is not uncommon multiple activities to require access to a single shared resource in a system. In order to avoid interference among these activities, access to the shared resource needs to be controlled. In operating systems, access to a section of code called the critical section (CS) is typically controlled using mutual exclusion algorithms. The problem of mutual exclusion in a distributed system is especially interesting owing to the lack of global knowledge in the presence of variable communication delays. There has been a significant amount of research on the distributed mutual exclusion problem [1, 3–13, 16], and a very good taxonomy of these algorithms is available in [14].

The performance of most distributed mutual exclusion algorithms has been traditionally evaluated by the number of messages generated per critical section invocation. Also, a useful mutual exclusion algorithm is characterized as fair to all nodes in the distributed system, being starvation-free and deadlock-free [2, 14].

In this paper, a new distributed mutual exclusion algorithm is proposed. The algorithm is token-based and has the flavor of a "reverse" Suzuki-Kasami [16] algorithm. However, the number of messages incurred is much lower than in other algorithms. In fact, at high loads, the algorithm on the average incurs less than 3 messages per critical section invocation, performing better than Raymond's tree-based algorithm [9], which is known to have the best performance, requiring approximately 4 messages at high loads.

In the next section, the basic algorithm is described and its execution is depicted through an example. Further, the correctness of the algorithm and its relation with other existing mutual exclusion algorithms is discussed. In Section 3, the performance analysis of this algorithm is presented, and then validated by simulation. In Section 4, the issue of starvation is discussed and a starvation-free variant of the proposed algorithm is described. Also, some additional important performance issues are presented, such as fairness with respect to scheduling of critical regions and load balancing in Section 5. A solution to the problem of token recovery in the presence of node and communication failures is presented in Section 6. Finally, Section 7 concludes the paper with a summary.

*This research is supported in part by the National Science Foundation under grant No. IRI-9210588.

2 The New Algorithm

2.1 Description of the Basic Algorithm

The new mutual exclusion algorithm, proposed here is token-based, and supports all the basic assumptions about token-based mutual exclusion algorithms: There is only one PRIVILEGE message (or *token*) in the system at any given moment in time. The token is passed from node to node, and only the token-holder is permitted to enter its critical section. No assumptions are made with respect to the network topology and the communication medium.

In this new algorithm, the token contains an *ordered list* or *queue* (referred to as the Q-list) of all the nodes that have been scheduled to execute their critical section. The token is passed from node to node in the order specified by the Q-list. The node which executes the critical section is the current node at the head of the Q-list.

The Q-list is created by a node currently designated as the *arbiter* of the system. Initially, a specific node (say, node 1) is assigned to be the arbiter by the system. As will become clear very soon, the responsibility of the arbiter node is shared by all the nodes in the system. Every node in the system keeps track of the arbiter by setting a parameter called ARBITER. An arbiter node executes the following two phases: *Request Collection Phase* and *Request Forwarding Phase*.

During the *request collection phase*, as the name suggests, the arbiter node collects all the requests from the nodes that are seeking access to their critical sections, and creates the ordered Q-list. At the end of the request collection phase, when the arbiter gets the token in its possession, the token is updated with the newly constructed Q-list and transmitted to the node that is at the head of Q-list, along with the contents of Q-list. Further, the arbiter node declares the node of the last request in the Q-list as the new arbiter, by sending a broadcast message to all the nodes in the system.

A node, on receipt of the message electing it as the new arbiter, enters the request collection phase. The token,

1063-6927/96 $5.00 © 1996 IEEE
Proceedings of the 16th ICDCS

Authorized licensed use limited to: University of Pittsburgh Library System. Downloaded on November 12,2025 at 14:54:05 UTC from IEEE Xplore. Restrictions apply.

which is passed from node to node according to the Q-list, finally reaches the new arbiter node. Thus, there are the following three kinds of messages used in this algorithm.

- PRIVILEGE message, which is of the form PRIVILEGE(Q), where Q is an ordered list of nodes which will be granted access to the CS one after another. The last node in Q is always the next arbiter node.
- REQUEST message is of the form REQUEST(j), when node j is requesting access to the CS.
- NEW-ARBITER message is of the form NEW-ARBITER(j), when node j is the new arbiter. The Q-list is sent as part of the NEW-ARBITER message.

Nodes requesting access to the critical section send their requests to the arbiter. It is possible however, that a node sends its request to the previous arbiter before a message informing it of the current arbiter arrives. In this case, the request message needs to be forwarded to the current arbiter. Thus, an arbiter enters the *request forwarding phase* after the request collection phase in order to forward requests that did not arrive during the request collection phase, but were transmitted before the identity of the current arbiter was received at the requesting nodes. Any requests arriving after the end of the forwarding phase, are dropped. The durations of the collection and forwarding phases are parameters that can be tuned for the best performance. This issue is discussed further in sections 3 and 4. The algorithm is summarized in Figure 1. An illustrative example follows.

2.2 Example

Consider a distributed system with 5 nodes, numbered 1 through 5 respectively and assume that the message transmission time, request collection and forwarding durations, and the execution time per critical section, are each equal to 1 unit of time.

Initially, node 1 is assigned to be the arbiter and enters the request collection phase immediately, whereas nodes 2, 4 and 5 require to be in their critical sections, and send their requests to node 1. Let us assume further that the request from node 2, REQUEST(2), and from node 5, REQUEST(5) arrive at node 1 within the request collection phase one after the other (see Figure 2). At the end of the request collection phase, node 1 assigns the node at the tail position of the Q-list – Tail(Q), viz., node 5 as the current arbiter, and broadcasts a NEW-ARBITER(5) message to all the nodes in the system. At the same time, the PRIVILEGE(Q) is transmitted to node 2 – Head(Q), where Q is the ordered list:{2,5}. After this, node 1 enters the request forwarding phase. REQUEST(4) arrives during the request forwarding phase, and node 1 forwards it to the current arbiter (node 5).

On receipt of the NEW-ARBITER(5) message, node 5 starts collecting requests from other nodes to access their critical sections. On receiving the PRIVILEGE(Q) message, node 2 enters its critical section. After executing its critical section, node 2 removes itself from the head position of Q, and sends the PRIVILEGE(Q) message to the node currently at the Head(Q), viz., node 5.

When node 5 subsequently receives the token, it executes its critical section, while still collecting requests in the background. After executing its critical section, node 5

```
function all-nodes(void);
        { ARBITER = 1;
          repeat {
             wait for NEW-ARBITER(node-id);
             ARBITER = node-id;
             if (ARBITER = self-node-id) {
                arbiter-node;
             }
          }
        }
function all-requesting-nodes(void);
        { send REQUEST(self-node-id) to ARBITER;
          wait for PRIVILEGE(Q);
          HavePrivilege = true;
          Execute CS;
          Q = Remove(Q,HEAD(Q));
          send PRIVILEGE(Q) to HEAD(Q);
          HavePrivilege = False;
        }
function request-collection(void);
        { t = 0;
          while (t <= REQUEST-COLLECTION-TIME) {
             Add incoming REQUEST(node-id) to Q;
             t = t + 1;
          }
        }
function request-forwarding(void);
        { t = 0;
          while (t <= REQUEST-FORWARDING-TIME) {
             Send incoming REQUEST(node-id) to
                                        ARBITER;
             t = t + 1;
          }
        }
function arbiter-node(void);
        { q = empty;
          while (not HavePrivilege) {
             Add incoming REQUEST(node-id) to q
          }
          if (HEAD(Q) = self-node-id){
             Execute CS while still monitoring
                    incoming requests;
          }
          Q = q;
          request-collection;
          if (Q is not empty) {
             send PRIVILEGE(Q) to HEAD(Q);
             broadcast NEW-ARBITER(Tail(Q));
             request-forwarding;
          }
          else {
             request-collection;
          }
        }
```

Figure 1: Mutual Exclusion Algorithm

Authorized licensed use limited to: University of Pittsburgh Library System. Downloaded on November 12,2025 at 14:54:05 UTC from IEEE Xplore. Restrictions apply.

removes itself from the head position, and enters its request collection phase, during which a request from node 3 (REQUEST(3)) arrives. At the end of the request collection phase, Q is the ordered list: {4,3}. Node 5 declares node 3 as the current arbiter by broadcasting the NEW-ARBITER(3) message, and the entire process is repeated.

2.3 Proof of Correctness

The proof of correctness of the proposed distributed mutual exclusion algorithm is similar to the proofs for most token-based mutual exclusion algorithms. An informal outline of the proof follows.

Assuming that the token is not lost, or replicated by the network, there is only one token in the system. The main point is that at any given time, only one node possesses the token, and hence is in its critical section, and only the same node can pass the token when it exits the critical section. In the proposed algorithm, the token is passed to the node at the head position of the ordered list of requesting nodes. Since the head position can be occupied by only one node at any given time, (and only the current head can send the token to the new Head(Q)) the token is never sent to more than one node, ensuring that no two nodes can be in the critical section simultaneously. In Section 5, we will discuss how the token is correctly recovered after a failure by the arbiter.

2.4 Related Work

The work most closely related to ours is reported in [7, 8]. While there are similarities in the basic idea underlying our algorithm and theirs, we have described a different realization of this independently discovered idea and have presented a different analysis and considered the failure recovery aspects of the proposed mutual exclusion algorithm.

As already indicated above, a number of distributed mutual exclusion algorithms have been proposed and classified into token-based and non-token-based [14]. The algorithm with the best performance is non-token based and was proposed by Raymond [9]. Of the token-based techniques, our algorithm is similar to the Suzuki-Kasami algorithm [16]. However, it is more efficient than even the tree-based algorithm proposed by Raymond, as shown in the next section.

Note that unlike the Suzuki-Kasami algorithm [16], where the request messages are sent to all the nodes, in our algorithm, the request message is sent to only one node. The reason for this is that all the nodes in the system can keep track of the identity of the arbiter node from the NEW-ARBITER messages. New arbiters are selected only at the end of a request collection phase, and hence broadcasting this information to all the nodes is done only at the end of a request collection phase, and not after every execution of the critical section. The main savings in the number of messages comes from these features. The nodes receiving the token remove their entry in the ordered list Q, and forward the token to the node at the head of the new list.

It has been implicitly assumed that the requests are ordered according to their arrival times at the queue. A fairer method may be to use sequence numbers, in which case, exactly as in the Suzuki-Kasami algorithm, the REQUEST messages will take the form REQUEST(j,n), when node j is requesting its $(n+1)^{\text{th}}$ critical section. The token will take the form PRIVILEGE(Q,L), where L is an array containing the sequence number of the last request granted for each node. The NEW-ARBITER messages will remain the same.

3 Performance Analysis

In this section, the performance of the proposed algorithm is analyzed. Only the theoretical bounds on the number of messages are presented here. The distributed system is assumed to have N nodes. The average number of messages ($\overline{M}$) and average service time per critical section ($\overline{X}$) for two extreme loading situations (very low and very high loads) is theoretically calculated. Later in this section, simulation results for a specific system are presented in order to study the performance at intermediate loads. The following simplifying assumptions are made.

- The request forwarding phase is ignored.
- The message delay between any two nodes is a constant T_{msg}.
- The execution time per critical section is a constant T_{exec}.
- The request collection time for all arbiters is a constant T_{req}.

3.1 Light load

Under situations of extremely light loading, there will be at most one out of N nodes requesting access to the critical section at any given time. Further, it is assumed that each of the N nodes is equally likely to be the current arbiter. Under such conditions, the number of messages incurred by each invocation of the critical section is 0 if the current arbiter is the requesting node itself. If the current arbiter is not the requesting node, then the total number of messages incurred are 1 REQUEST message, (N-1) NEW-ARBITER messages, and 1 token. It should be noted that in this case, the NEW-ARBITER message need not be sent to the new arbiter, since the token containing only its own request is proof that it is the new arbiter. Hence the average number of messages incurred $\overline{M}$, under low load situations is:

$$\overline{M} = (1 - \frac{1}{N})(1 + N - 1 + 1) = \frac{(N^2 - 1)}{N} \qquad (1)$$

From the above, it follows that in a large distributed system (big N), the average number of messages incurred per critical section invocation tends to N.

$$\overline{M} \rightarrow N, \quad \text{for} N \gg 1 \qquad (2)$$

Another important performance metric is the average service time for each critical section $\overline{X}$. This parameter is similar to the response time, as defined in [14], except that $\overline{X}$ includes the execution time for the critical section as well. From the arguments made above, at low loads, the average delay to execute the critical section after arriving at the head(Q) is as given below.

$$\begin{aligned} \overline{X} &= (1 - \frac{1}{N})\, T_{msg}(1+1) + T_{req} + T_{exec} \\ &= (1 - \frac{1}{N})\, 2\, T_{msg} + T_{req} + T_{exec} \end{aligned} \qquad (3)$$

Authorized licensed use limited to: University of Pittsburgh Library System. Downloaded on November 12,2025 at 14:54:05 UTC from IEEE Xplore. Restrictions apply.

3.2 Heavy load

Under situations of heavy loading, all nodes will have at least one pending request to enter the critical section. Hence, at any given time, there will be N requests in Q. With N requests in the Q, the PRIVILEGE(Q) message will have to be sent (N-1) times, and there will be (N-1) NEW-ARBITER messages generated for every N executions of the critical section. Hence the average number of messages incurred under heavy load situations is:

$$\overline{M} = (1 - \frac{1}{N}) + \frac{N + (N-1)}{N} = 3 - \frac{2}{N} \quad (4)$$

From the above, it follows that when the number of nodes is large, the average number of messages generated per critical section is approximately 3 messages.

$$\overline{M} \rightarrow 3, \text{ for} N \gg 1 \quad (5)$$

The average service time to execute a critical section after arriving at the head of Q position at a node, under heavy load is given by the following.

$$\overline{X} = (1 - \frac{1}{N})\,\mathrm{T_{msg}} + \mathrm{T_{req}} + (\frac{N}{2} + 1)\,(\mathrm{T_{msg}} + \mathrm{T_{exec}}) \quad (6)$$

3.3 Simulation Results

In the above analysis, the request forwarding process has not been taken into account. Hence a simulation of this algorithm was performed to evaluate the effect of the request forwarding process on performance. The simulation was event-driven with multiple runs. Each simulation run processed a total of a million requests for the critical section from 10 nodes. Three parameters were calculated, viz., the average number of messages generated per invocation of the critical section, the percentage of forwarded messages, and the average delay per critical section. Each parameter was calculated for a range of loads. For the sake of simplicity, each of the nodes generated requests using a Poisson probability distribution with the same arrival rate λ requests/second. Further, all the algorithm parameters, such as the request collection and request forwarding durations are the same at all the nodes. The execution time per critical section at each node is a constant.

In Figures 3 and 4, the average number of messages generated is plotted against the arrival rate of requests for critical section. 95% confidence intervals were computed and plotted for each parameter. The confidence intervals are too small to be noticed on the graphs in most cases. The message transmission time, request forwarding time and critical section execution time is set to the value of 0.1 units. Two cases are studied with the request collection phase duration set to 0.1 units (continuous curve) and 0.2 units (dotted curve) respectively. As expected, it is observed that with a longer request collection phase, the average number of messages incurred is lower, but the average delay per critical section is higher. In Figure 5, the fraction of forwarded messages is shown for the two cases. At very high loads, the fraction of forwarded messages becomes negligible. Again, as expected, the fraction of forwarded messages is lower when the request collection phase is longer, enabling most requests to arrive during the collection phase.

From Figure 3, at high loads, the average number of messages incurred reduces to approximately 3 messages, which is better than Raymond's tree algorithm [9] which incurs 4 messages at high loads. In Figure 6, our algorithm is compared to the Ricart-Agrawala [10] and the dynamic mutual exclusion algorithm presented in [13]. Thus we compare our scheme to two important classes of mutual exclusion algorithms, viz., static and dynamic. A comparison with Raymond's algorithm [9] is not made in order to keep the comparison platform reasonably general and not topology-dependent. As is to be expected, the scheme proposed here performs better than the Ricart-Agrawala algorithm at all loads. Except at very low loads, it does better than the dynamic algorithm as well.

4 The Starvation Issue

In the context of mutual exclusion algorithms, *starvation* is said to occur if one or more nodes do not receive permission to enter the critical section for excessively long periods of time, and sometimes never at all. In the proposed algorithm, starvation may occur only if request messages continuously get forwarded, without the request being registered at the arbiter, or if request messages are continuously dropped because of their arrival at the arbiter after the request collection and forwarding phases are over. Although only a maximum of 4% of messages were forwarded, in the simulation, this is an important problem. Hence this problem is termed as the *indefinite forwarding problem* in the context of this paper.

It should be apparent that the probability of indefinite forwarding is much higher at low loads than at high loads. This is because at higher loads, there are a large number of requesting nodes, causing the token to arrive at the current arbiter only after all the requesting nodes have completed executing their critical section. This gives ample time for the forwarded message to arrive at the current arbiter before its request collection phase is over. The worst possible scenario occurs, when only a single request message from node i is received at the current arbiter l during the request collection phase. Node i is assigned to be the current arbiter. Further, a new request from node j arrives at the end of the request forwarding phase of l, and is immediately forwarded to i. For the request from node j to not be forwarded again, the following equation must be satisfied. It is assumed here that the request forwarding phase duration is set to the time it takes to broadcast the NEW-ARBITER message to all nodes in the distributed system plus the time for a request message to arrive at the arbiter.

$$\mathrm{T_{privilege}} + \mathrm{T_{exec}} + \mathrm{T_{req}} > \mathrm{T_{fwd}} + \mathrm{T_{fwd\text{-}req}} \quad (7)$$

Thus, indefinite forwarding can be avoided by assigning appropriate values to the request collection and forwarding durations. However, this would work only if execution times, and message transmission times are deterministic. In reality, execution times depend on the current processor load. Message transmission times depend on the current network and processor loads. Hence, an alternate simple method of avoiding indefinite forwarding and hence starvation is outlined below.

Authorized licensed use limited to: University of Pittsburgh Library System. Downloaded on November 12,2025 at 14:54:05 UTC from IEEE Xplore. Restrictions apply.

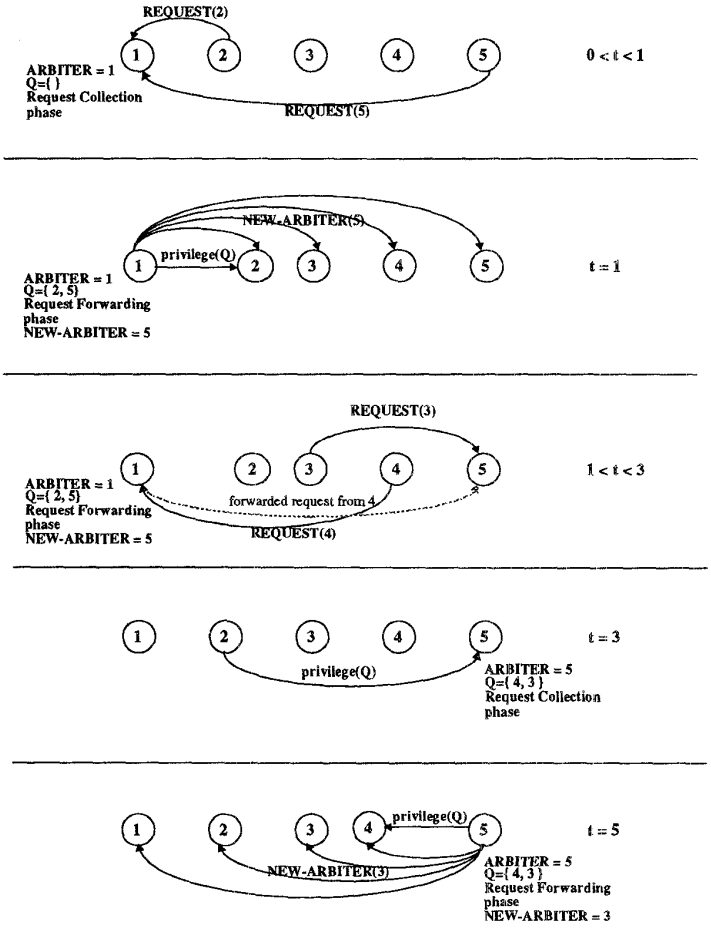

Figure 2: Illustrative Example

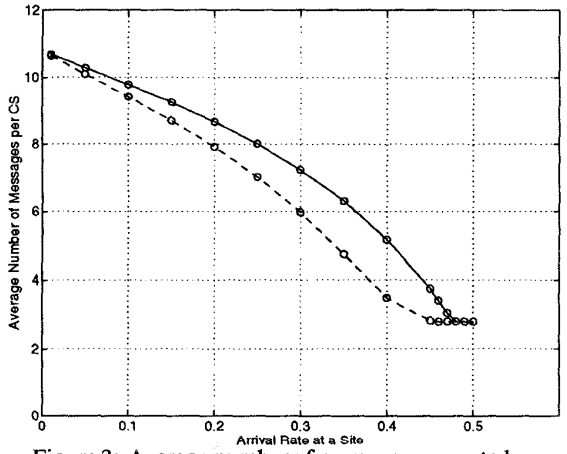

Figure 3: Average number of messages generated

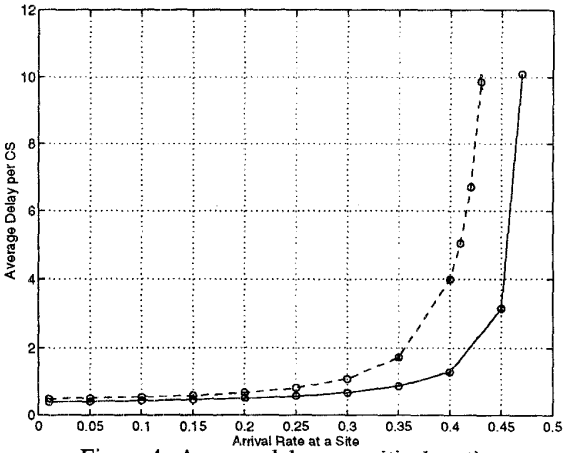

Figure 4: Average delay per critical section

Authorized licensed use limited to: University of Pittsburgh Library System. Downloaded on November 12,2025 at 14:54:05 UTC from IEEE Xplore. Restrictions apply.

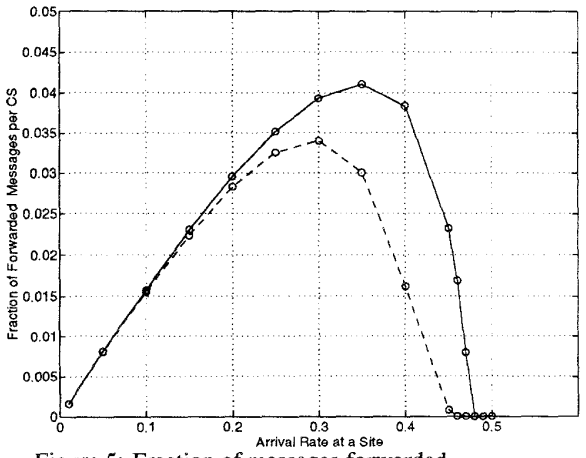

Figure 5: Fraction of messages forwarded

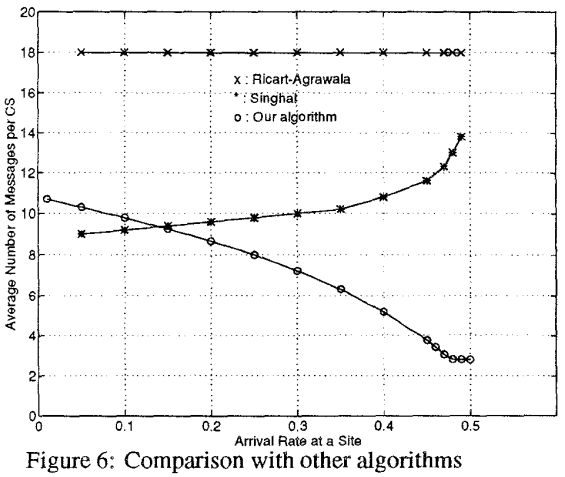

Figure 6: Comparison with other algorithms

4.1 Description of the Starvation-Free Algorithm

The basic scheme described in Section 2 remains the same, except for the following important addition. One node is assigned to be a *monitor* node, whose identity is known to all other nodes. Request messages that have been forwarded more number of times than a given *threshold* τ, are dropped by an arbiter even if they arrive within the request forwarding phase. A requesting node resubmits the request to the monitor after failing to see its request in the Q-list in τ consecutive NEW-ARBITER messages, which is an indication that its request has been dropped.

The problem of starvation is eliminated by passing the token to the monitor node periodically with a specific *period*. The monitor does not forward any request messages, but stores them until the token arrives and the requests can be appended to the Q-list. Thus, the monitor maintains a set of requests that could potentially have become victims of the indefinite forwarding phenomenon. When an arbiter determines that the token must be sent to the monitor node, it sends the token without broadcasting a NEW-ARBITER message, which is now broadcast by the monitor node. When the monitor node receives the token, it acts as an arbiter in that it augments the Q-list in the token with the set of requests it has received, broadcasts a NEW-ARBITER message, forwards the token and discards its set of requests.

The period to include the monitor node should depend on the current load on the system. The reason being that at low loads, the probability of forwarding is high and hence the period should be short, whereas at high loads, the probability of forwarding is low, and hence the period can be long. We propose an adaptive period based on the average Q-list size. Specifically, each node keeps track of the size of the Q-list by observing the NEW-ARBITER messages and computes the average size of the Q-list within a moving window. Further, the NEW-ARBITER messages are augmented with a counter which is incremented every time a NEW-ARBITER message is sent out. An arbiter decides to send the token to the monitor node when its NEW-ARBITER message counter equals the ceiling of its computed average size of the Q-list. The counter is then set to zero by the monitor node when it broadcasts the NEW-ARBITER message. Thus at high loads, the queue size will be high, causing the period to be long, and vice versa.

The above modifications will cause the average number of messages to increase by incurring 1 extra message (one token message to the monitor) in every period. Assuming that at extremely low loads, in every period only 1 CS is executed, the average number of messages incurred will be 1 more message than in the basic algorithm. At high loads, since several CS executions will be done in every period, the average number of messages per CS will not be affected significantly by this modification.

5 Other Performance Issues

5.1 Fairness

In evaluating fairness properties of a mutual exclusion algorithm, there have been two main criteria.

- In the absence of prioritized access, the mutual exclusion algorithm must not favor one node over another.
- The work involved in executing the mutual access algorithm must be fairly divided among the nodes.

The proposed mutual exclusion algorithm is fair in both the above aspects, as argued below. Since the requests for access to the critical section are processed on a first-come-first-served (FCFS) basis, every node is guaranteed to access the critical section in a fair manner. In order to satisfy a more strict notion of fair access, a scheme similar to Suzuki-Kasami's algorithm may be implemented. In the Suzuki-Kasami algorithm, access to the critical section is granted based on the number of times a node has entered its critical section previously. The node that has accessed the critical section the least number of times is given priority. A similar scheme may be implemented in this algorithm by simply passing a sequence number (the number of times that node is requesting access to the critical section), as discussed in Section 2.4.

The proposed algorithm is fair with respect to the load balancing criteria as well. In fact, it is fair in a stricter sense than most other algorithms that have been proposed earlier. Most mutual exclusion algorithms assign some work load

Authorized licensed use limited to: University of Pittsburgh Library System. Downloaded on November 12,2025 at 14:54:05 UTC from IEEE Xplore. Restrictions apply.

on all nodes, irrespective of whether a particular node is creating any load (requesting access to the critical section) or not. For instance, the Ricart-Agarwala algorithm [10] causes request messages from a node to be sent to all other nodes in the system, which then have to reply to the requesting node. In the Maekawa algorithm [6], the workload is divided equitably only when every node has the same request rate. A fairer criterion to balance the load would be to divide the workload only among the nodes that are generating the load. In the proposed algorithm, at a given time, a single node is given the responsibility of executing the mutual exclusion algorithm, and different nodes take turns to act as the arbiter. However, only the nodes that request for the critical section are likely to be assigned the responsibility of being an arbiter in the future. The higher the number of requests generated by a node, the higher is the probability of being designated to be the arbiter.

The extension to the basic algorithm to eliminate starvation requires one node to be the monitor node, thereby loading one node more than the others. However, the role of the monitor node can also be shared by all the nodes by rotating (in a round-robin fashion) the functions of the monitor node. In that case, the NEW-ARBITER message broadcast by the monitor will include the identity of the new monitor node.

5.2 Prioritized Access

Starvation becomes an especially important problem when access to the critical section is prioritized. For instance, nodes are assigned priorities. In such circumstances, it is possible that nodes with low priorities may be preempted constantly by higher priority nodes, and hence starve. Usually, priorities are assigned dynamically, for instance, the least recently used (LRU) criteria is a dynamic priority assignment scheme. However, in the case of a static priority system, starvation is a major problem.

In the proposed algorithm, in the absence of node priorities, starvation can never occur since all nodes are granted permission in the FCFS order. In the following discussion, it is attempted to show that even in the presence of a static assignment of priorities, the starvation problem is not encountered. In the proposed algorithm, the arbiter node collects the requests to enter critical section from all the nodes in the system. In the presence of priorities, the arbiter will order the requests in the order of the node priorities. Since the lower priority nodes will end up towards the end of this ordered list, there will be a higher probability that one of the lower priority nodes will be elected to be the next arbiter, and will indeed get a chance to execute their critical section. Thus, the starvation problem is non-existent. However, two points should be noted.

- The proposed algorithm implements only an incremental priority-based system. That is, priorities are implemented every time a new arbiter is selected, and not in between the arbiter selection process. Thus, it is possible that some higher priority requests that arrive after the request collection phase and before the current arbiter has executed its critical section, will be serviced after some lower priority requests.

- The algorithm is unfair with respect to load balancing to the lower priority nodes, since it is these nodes that have a high probability of being selected as an arbiter.

6 Recovery from Failures

The proposed mutual exclusion algorithm is non-blocking to node failures as long as the node currently in possession of the token does not fail. The failure of nodes that are not scheduled to receive the token does not impede the successful execution of the mutual exclusion algorithm. Further, the mutual exclusion algorithm can proceed at all times if the node that currently has the token (executing its critical section), the current arbiter node (collecting requests) and the previous arbiter node (forwarding requests) are operational, even if all the other nodes have failed. Thus, this algorithm has the potential of running with just three operational nodes, and is thus, highly resilient to node failures. Below, we consider the possibility that the message containing a request or the token are lost.

- Lost Request: If a request message is lost, the requesting node can detect it relatively easily by monitoring the NEW-ARBITER message that is broadcast each time a new arbiter is selected. As mentioned before, the NEW-ARBITER message contains the Q-list as well. Thus, if a requesting node does not find its request in the NEW-ARBITER message, it can detect the loss of its request, and retransmit the request. Thus the NEW-ARBITER messages act as an implicit acknowledgment that a request has been received and scheduled. Further, appropriate timeouts may also be used to retransmit a request, in case the token is not received within the timeout period.

  Note that a lost request due to communication errors, a delayed request due to queuing delays in the communication network, and a dropped packet due to the arrival of the request after the forwarding phase, will have the same effect, viz., the request will not be in the Q-list broadcast in the NEW-ARBITER messages. With the increasing quality of emerging communication networks, loss or delay of requests will be minimized. Thus resubmitting the request to the monitor node after noticing the absence of its request in τ consecutive NEW-ARBITER messages may suffice.

- Lost Token: The lost token problem is more problematic and requires the following detailed solution. A token may be lost either because the node currently holding the token fails or because the PRIVILEGE message was dropped. The failure of previous arbiter is also considered since the previous arbiter may fail after broadcasting the NEW-ARBITER message and before sending out the token. The proposed solution is timeout-based and in this sense is similar to the timeout-based solutions proposed for other token-based systems, including token ring networks [15].

  After confirming that its request has been received and scheduled from the NEW-ARBITER message, every requesting node (including the current arbiter) selects an appropriate timeout to receive the token. When a node times out, it sends a WARNING message to the current arbiter. When the arbiter receives a WARNING message or times out, it starts a *two-phase token invalidation* protocol.

  - *Phase 1:* The arbiter sends an ENQUIRY message to all the nodes on the Q-list (including the

Authorized licensed use limited to: University of Pittsburgh Library System. Downloaded on November 12,2025 at 14:54:05 UTC from IEEE Xplore. Restrictions apply.

previous arbiter). When nodes receive the ENQUIRY message, they respond with one of the following messages. Further, if they possess the token, they suspend their CS execution, and do not forward the token to the next node.

  * I had the token, and have executed my CS.
  * I have the token.
  * I am waiting for the token.

- *Phase 2:* If a single node responds with a "I have the token" message, then the arbiter responds with RESUME messages, which causes regular operations to proceed.

  After either all nodes have responded or the arbiter times out, if no node has the token, then the arbiter sends out INVALIDATE messages to all the nodes that are waiting for the token and adds them on the front of its Q-list. Also, nodes that do not respond are assumed to have failed, and are not included on the Q-list.

- *Failed Arbiter node:* To prevent the system to be blocked by the failure of the arbiter, the following method is proposed. The current arbiter is monitored by the previous arbiter. If the previous arbiter does not receive a NEW-ARBITER message within a timeout period, it probes the current arbiter. If the probe is unanswered, the previous arbiter sends a NEW-ARBITER message proclaiming itself the current arbiter. If the requesting nodes do not find their outstanding requests in the Q-list, they can retransmit their request to the current arbiter. The failure of the monitor node may also be handled as above.

7 Summary and Conclusions

In this paper, a token passing distributed mutual exclusion algorithm is proposed, and analyzed. The algorithm performs well with respect to the number of messages incurred. Under heavy load, the number of messages incurred is approximately three. There are two parameters: the request collection phase and the request forwarding phase durations that may be adjusted to obtain the best performance. Further, a starvation-free variant of the basic algorithm has been proposed. The failure recovery problem has also been discussed, owing to its importance in token based systems.

The distributed mutual exclusion algorithm proposed in this paper is not strictly distributed as defined by Ricart and Agarwala in [10], in that not every node participates in the decision to grant permission to access the critical section. This is the case for Raymond's tree-based distributed mutual exclusion algorithm [9] as well as the one proposed by Maekawa [6]. The Suzuki-Kasami algorithm [16] is not strictly distributed either since, only the node holding the token decides the next node to receive the token and hence access the critical section. Thus, the process of deciding the next token-holder is not distributed, but the task of decision-making is distributed to each node in turn. The proposed algorithm also is distributed in a similar manner. Finally, the algorithm is fair with respect to load balancing and scheduling of critical section, while it can support partial prioritized access. Future work involves a more detailed study of the performance in the face of failures, as well as comparisons with a larger number of algorithms.

Acknowledgments

We thank the anonymous reviewers for their comments, that helped to improve this paper. We also thank Dr. Kia Makki and Dr. Niki Pissinou for bringing related papers [7, 8] to our attention.

References

[1] D. Agrawal and A. E. Abbadi. An Efficient and Fault-Tolerant Solution for Distributed Mutual Exclusion. *ACM Trans. on Computer Systems*, 9(1):1–20, 1991.

[2] A. Bernstein and P. Lewis. *Concurrency in Programming and Database Systems*. Jones and Bartlett, 1993.

[3] P. Chaudhuri. Optimal Algorithm for Mutual Exclusion in Mesh-Connected Computer Networks. *Computer Communications*, 14(10):627–632, 1991.

[4] L. Lamport. The Mutual Exclusion Problem: Part-I - A Theory of Interprocess Communication. *Journal of the ACM*, 33(2):313–326, 1986.

[5] L. Lamport. The Mutual Exclusion Problem: Part-II - Statement and Solutions. *Journal of the ACM*, 33(2):327–348, 1986.

[6] M. Maekawa. A $\sqrt{N}$ Algorithm for Mutual Exclusion in Decentralized Systems. *ACM Trans. on Computer Systems*, 3(2):145–159, 1985.

[7] K. Makki, P. Banta, K. Been, and R. Ogawa. Two Algorithms for Mutual Exclusion in a Distributed System. In *Proceedings of the International Conference on Parallel Processing*, pages 460–466, 1991.

[8] K. Makki, K. Been, and N. Pissinou. A Simulation Study of Token-Based Mutual Exclusion Algorithms in Distributed Systems. *International Journal in Computer Simulation*, 4(1), 1994.

[9] K. Raymond. A Tree-Based Algorithm for Distributed Mutual Exclusion. *ACM Trans. on Computer Systems*, 7(1):61–77, 1989.

[10] G. Ricart and A. K. Agrawala. An Optimal Algorithm for Mutual Exclusion in Computer Networks. *Communications of the ACM*, 24(1):9–17, 1981.

[11] R. Satyanarayanan and C. R. Muthukrishnan. Message-efficient Distributed Mutual Exclusion Incorporating the 'Least Recently Used' Fairness Criterion. *IEE Proceedings-E*, 139(6):501–504, 1992.

[12] R. Satyanarayanan and D. R. Muthukrishnan. A Note on Raymond's Tree Based Algorithm for Distributed Mutual Exclusion. *Information Processing Letters*, 43(5):249–255, 1992.

[13] M. Singhal. A Dynamic Information-Structure Mutual Exclusion Algorithm for Distributed Systems. *IEEE Trans. on Parallel and Distributed Systems*, 3(1):121–125, 1992.

[14] M. Singhal. A Taxonomy of Distributed Mutual Exclusion. *Journal of Parallel and Distributed Computing*, 18(1):94–101, 1993.

[15] W. Stallings. *Data and Computer Communications*. Macmillan, 1991.

[16] I. Suzuki and T. Kasami. A Distributed Mutual Exclusion Algorithm. *ACM Trans. on Computer Systems*, 3(4):344–349, 1985.

Authorized licensed use limited to: University of Pittsburgh Library System. Downloaded on November 12,2025 at 14:54:05 UTC from IEEE Xplore. Restrictions apply.