A Centralized Token-Based Algorithm for Distributed Mutual Exclusion *

Edward W. Felten   Michael Rabinovich

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

In this paper, we present a new algorithm for distributed mutual exclusion that combines the advantages of existing centralized and token-based algorithms: it takes only two messages to acquire the lock when it is not held by another process, and only one message to pass the lock from one process to another. At the same time, our algorithm generates significantly less message traffic than non-centralized token-based algorithms. We also show that, within our approach, the performance of the algorithm can be further improved by making use of information on the pattern of lock requests, if available. Finally, we outline a combined protocol that uses our efficient algorithm most of the time and switches dynamically without service interruption to an existing non-centralized token-based algorithm during periods when the central node in our algorithm becomes temprorarily overloaded.

1 Introduction

Mutual exclusion is one of the basic problems in concurrent computing. It arises whenever conflicting parallel processes must share a common resource so that only one process may access the resource at any given time. A process that has the exclusive right to access a resource is said to hold the lock for this resource. To be correct, a mutual exclusion algorithm must ensure that (1) at any time, at most one process can hold the lock; and (2) any request for the lock will be granted in finite time.

Distributed mutual exclusion is a particular case of the general mutual exclusion problem; in the distributed case, processes reside on separate nodes and can communicate only by sending messages. The most obvious solution to the distributed mutual exclusion problem is a centralized algorithm. In such an

*This research was supported in part by the National Science Foundation (Grants No. CCR-8907666 and CCR-8619663), Digital Equipment Corporation, Apple Computer, and the Washington Technology Center. In addition, Felten's work was supported in part by an AT&T Ph.D. Scholarship and a Mercury Seven Fellowship.

algorithm, one node is dedicated to service the requests for mutual exclusion from all nodes in the system. A requesting node sends a `REQUEST` message to the dedicated (central) node and obtains the lock by receiving a `PRIVILEGE` message in response. A node releases the lock by sending a `RELEASE` message to the central node. This algorithm is simple and requires a very small number of messages per lock acquisition. However, it has two problems. First, the algorithm is vulnerable to failure or disconnection of the central node. Second, two sequential messages (`RELEASE` and `PRIVILEGE`) are required to pass the lock from one process to another.

To address the first problem, quorum-based algorithms have been proposed (see, e.g., [13, 1, 14]). In these algorithms, each node maintains its own local lock that can be given to one process at a time. In order to attain mutual exclusion, a process must obtain local locks on a set of nodes called a *quorum* [6, 7]. Quorums are defined in such a way that any two of them have at least one node in common, which ensures that at most one process can succeed in obtaining local locks from a quorum. These algorithms allow a process to attain mutual exclusion as long as some quorum of nodes is operational and accessible, while still requiring two sequential messages for the (global) lock to be passed from one process to another.

The second problem of the centralized algorithm is addressed by token-based and permission-based algorithms (see, e.g., [15, 3, 16, 20, 4, 9, 11, 18, 19]). It appears that every known permission-based algorithm has a corresponding token-based algorithm generating less message traffic [4]. For this reason, we limit our discussion here to the token-based algorithms. In these algorithms, the mutual exclusion privilege is associated with possession of the *token*. The algorithms provide that there is always only one valid token in the system, and hence, mutual exclusion is enforced. The requests for mutual exclusion are put into a (possibly distributed) token queue, so that the process possessing the token knows immediately which (if any) request is next in line. Therefore, when a process releases the lock, it passes the token (and hence, the lock) directly to the next process using one message. As in the case of the centralized solution, these algorithms guarantee service only if all nodes in the system are operational and connected — they may stop providing service during failures.

Both quorum-based and token-based algorithms achieve their goals at the expense of increased message traffic in the system. In the quorum-based algorithms, a process must request and receive the privilege from at least a quorum of nodes. In the token-based algorithms, a process contacts a group of nodes in search of the token. Another disadvantage of many of these algorithms as compared to the centralized solution is that they require a multicast facility for the network, so that a node can send a message to a group of other nodes in parallel and almost as cheaply as to just one node. If such a facility is not provided (and many of the newer networks do not provide one [17, 2]), then these protocols become prohibitively expensive.

In this paper, we present a centralized algorithm that, like the existing token-based algorithms, requires only one message to pass the lock from one process to another and, at the same time, generates significantly less message traffic than non-centralized algorithms. In fact, under heavy load, i.e., when message traffic is of most concern, our algorithm generates less traffic than any existing algorithm we know, including the centralized ones.

We also discuss the issues of load sharing and fault-tolerance, two common concerns about centralized algorithms. We show that our protocol exhibits the same fault tolerance as non-centralized token based protocols and outline the way that would allow the system to switch dynamically without service interruption from our centralized to a distributed token-based algorithm to deal with rare periods when the central node gets overloaded.

Existing research into token-based algorithms for distributed mutual exclusion has concentrated on non-centralized algorithms only. This paper shows that the long neglected centralized approach has much unresearched potential, and in many cases provides a better alternative.

In section 2 of the paper, we consider measures of performance commonly used to compare different solutions for the distributed mutual exclusion problem, and discuss the existing token-based algorithms. We present our new algorithm in section 3, and examine its properties in section 4. In particular, we analyze the performance and fault tolerance of our algorithm and compare our protocol with the existing solutions. Section 5 outlines ways to further improve our algorithm which we intend to explore more in the future. In section 6, we summarize the results of the paper.

2 Background

2.1 Performance Measures

Several performance parameters are used to compare different solutions to the distributed mutual exclusion problem. *Number of messages per lock acquisition* is the total number of messages required to obtain and release the lock. This parameter measures overall network traffic generated by an algorithm. *High-load synchronization delay* is the number of *sequential* messages required to pass the lock from one process to another. The lock is passed from one process to another when, by the time one process releases the lock, there is another process requesting it. This happens often if the rate of requests is high — hence, the name of the parameter. *Light-load synchronization delay* is the number of sequential messages required for a process to obtain the lock if it is not held by any other process. Unlike the number of messages per lock acquisition that includes both sequential and parallel messages, synchronization delays are concerned with sequential messages only and thus measure inherent latencies of an algorithm.

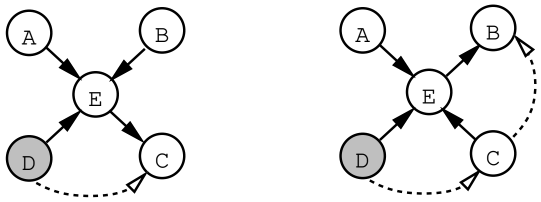
2.2 Related Work

In token-based algorithms [4, 19, 16, 20], each node counts the requests for mutual exclusion it issues and includes the request number in the request message. Each node i also maintains an array L_i of length N, where N is the number of nodes in the system. An element $L_i[j]$ stores the highest request number from node j of which node i is aware. The token has an array H of length N such that $H[i]$ contains the number of times that node i actually acquired the lock. So, if node i holds the token and for some j, $H[j] < L_i[j]$, node i knows that j is waiting for the lock. Thus, when site i releases the lock, it increments $H[i]$ by one to reflect its latest possession of the lock, applies some fair procedure to choose a node j_{next} among all nodes j such that $H[j] < L_i[j]$, and sends the token to j_{next}. The difference among these algorithms is in how they determine to which nodes the request message must be sent. In one of the first such algorithms [16, 20], the request is sent to all nodes. Later algorithms [19, 4] use various ways to determine more accurately the possible token-holders to which the request must be sent. Still, all these algorithms require $O(N)$ messages per lock acquisition. Since the node releasing the lock sends the token directly to the next node, the high-load synchronization delay is 1. The light-load synchronization delay is 2 for a node not holding the token (it must send out a request message and receive the token) and 0 for the node holding the token.

In [9], a way of locating the token based on quorum agreements was proposed. Given a set of all nodes in the system U, a quorum agreement is a pair of sets of subsets of U, (Q, Q^{-1}), such that every member of Q intersects with every member of Q^{-1}. In this algorithm, a node sends a request to all nodes from some set $R \in Q$, while a node possessing the token sends a notification to all nodes from some set $A \in Q^{-1}$. By the definition of quorum agreements, there is always a node among the ones to which the request is sent that receives the notification from the token-holder and, hence, knows where the token is. Thus, the request can always be forwarded to the node holding the token. This algorithm cuts significantly the number of messages per lock acquisition. While the particular quorum agreement used in [9] brings this number to $O(\sqrt{N})$, the best performance is achieved when both Q and Q^{-1} are singletons containing the same subset, which in turn contains the same single node: $Q = Q^{-1} = \{\{M\}\}$ for some node M. In this case, the number of messages per lock acquisition is 3 or 4, depending on the arrival times of requests and messages in the system. (In the best case, several requests can be combined into one forward message, while in the worst case, a separate forward message is required for each request.) The disadvantage, however, is that the upper bound for the light load synchronization delay increases to 3 due to the extra level of indirection in finding the token.

Finally, several algorithms use pointers to track the whereabouts of the token and to represent the token queue as a distributed linked list. In the most recent of these algorithms [11], the pointers maintain a DAG topology of the network in which every node is on a path to the last node in the token queue (as for example in Figure 1a). When a node (say, `B`) wants to obtain the lock, it sends the request message along the path. The request message contains the identity of the requesting node, so that when the message reaches the sink node `C` (which is the last node in the token queue), the latter can enqueue the requesting node into the token queue by linking the requesting node to the tail of the corresponding distributed linked list. While the request message travels along the path towards the sink, all arcs along the way are reversed. Therefore, after the requesting node is enqueued, all paths in the system will again lead to the (new) last node in the token queue (Figure 1b). The authors note that the best topology for this DAG-based algorithm is a star topology, as in Figure 1, in which case the number of messages per lock acquisition is 3 as in the centralized algorithm and the heavy-load synchronization delay is 1 as in the token-based algorithms considered above. However, the light-load synchronization delay is 3 if the requesting node does not have the token.

If the star topology is used in the DAG-based algorithm, or the singleton quorum agreement is utilized in the quorum-agreement-based algorithm (i.e., when these algorithms achieve their best performance), they



a. Node D holds the token; the token queue is (D,C).

b. D holds the token; the token queue is (D,C,B).

Figure 1: The network structure in the DAG-based algorithm. The dotted lines represent the token queue; the solid lines show the paths to the last node in the token queue.

can be considered centralized algorithms. Indeed, in the DAG-based algorithm, the same node is always situated in the center of the star, and in the quorum-agreement-based algorithm, the same node is always at the crossroads of all request and notification messages. Therefore, these algorithms are in fact the first step in the direction argued for in the present paper. Our algorithm goes further in this direction by making the centralized approach explicit and thus doing away with the necessity to support general cases. This has several benefits. First, the light load synchronization delay in our algorithm is 2. Second, under heavy load, i.e., when the volume of message traffic is especially important, our algorithm generates about 30% fewer messages per lock acquisition than the DAG-based algorithm, and fares even better when compared with the quorum-agreement-based algorithm. Finally, our algorithm is more flexible and can provide policies which allow the application to make use of the information available to it in order to improve the performance of the algorithm. We discuss these policies later in the paper.

3 The Algorithm

Figure 2 shows the algorithm run by the master node, and figure 3 shows the algorithm run by the client nodes. (Note that the master may be co-located with one of the clients.) A client requests the token by sending a *TokenRequest* message to the master. The master maintains two data structures: the *activeTokenQ* contains a list of nodes that have requested the token, and which will receive the token without further action by the master; the *passiveTokenQ* is a list of nodes that have requested the token, but whose requests will not be satisfied until the master sends further messages.

A client node receives the token by the arrival of a *TokenGrant* message. A *TokenGrant* message

```
procedure Master() {
  const
    WarningLength : Integer;
  var
    requester : NodeIdentifier;
    activeTokenQ : SEQUENCE OF NodeIdentifier;
    passiveTokenQ : SEQUENCE OF NodeIdentifier;
    batchNum : Integer;

  activeTokenQ := ∅;
  passiveTokenQ := ∅;
  batchNum := 1;
  forever loop {
    receive message;
    if(message matches TokenRequest(requester)){
      passiveTokenQ = append(passiveTokenQ, {requester});
    }else if(message matches DoneReport(C)){
      while(head(activeTokenQ)! = C){
        activeTokenQ := removeFirstItem(activeTokenQ);
      }
      activeTokenQ := removeFirstItem(activeTokenQ);
    }
    if( (activeTokenQ = ∅) AND (passiveTokenQ! = ∅) ){
      activeTokenQ := passiveTokenQ;
      passiveTokenQ := ∅;
      send TokenGrant(batchNum, activeTokenQ, True) to head(activeTokenQ);
      batchNum := batchNum + 1;
    }else if( (length(activeTokenQ) < WarningLength) AND (passiveTokenQ! = ∅) ){
      send ForwardToken(batchNum, passiveTokenQ) to tail(activeTokenQ);
      batchNum := batchNum + 1;
      activeTokenQ := append(activeTokenQ, passiveTokenQ);
      passiveTokenQ := ∅;
    }
  }
}
```

Figure 2: The algorithm run by the master node.

```
const
   MyNodeID : NodeIdentifier;
   WarningLength : Integer;
var
   holding : Boolean;
   afterMe : SEQUENCE OF NodeIdentifier;
   shouldReport : Boolean;
   lastBatch : Integer;
   forwardQ : SEQUENCE OF NodeIdentifier;
   localLock : Lock;

procedure initialize() {
   holding := False;
   forwardQ := ∅;
   lastBatch := 0;
   forwarding := False;
}

procedure AcquireSharedLock() {
   send TokenRequest(MyNodeID) to Master;
   receive TokenGrant(Bnum, Q, ShouldRep);
   holding := True;
   lastBatch := Bnum;
   shouldReport := Should;
   afterMe := removeFirstItem(Q);
}

procedure ReleaseSharedLock() {
   acquire localLock;
   holding := False;
   if(forwardQ! = ∅){
      afterMe := append(afterMe, forwardQ);
      forwardQ := ∅;
      lastBatch := lastBatch + 1;
      shouldReport := True;
   }
   if( (afterMe = ∅) OR ((length(afterMe) < WarningLength) AND shouldReport) ){
      send DoneReport(MyNodeID) to Master;
      shouldReport := False;
   }
   if(afterMe! = ∅){
      send TokenGrant(lastBatch, afterMe, shouldReport) to head(afterMe);
   }
   release localLock;
}

procedure ForwardTokenHandler(Bnum, Q) {
   acquire localLock;
   if( (Bnum = (lastBatch + 1)) AND (NOT holding) ){
      send TokenGrant(Bnum, Q, True) to head(Q);
   }else if(Bnum > lastBatch){
      forwardQ := Q;
   }
   release localLock;
}
```

Figure 3: The algorithm run by the client nodes. Arrival of a message matching $ForwardToken(G, Q)$ triggers execution of the procedure ForwardTokenHandler.

contains a queue of nodes that have requested the token. When a client receives a *TokenGrant* message, it removes its own ID from the head of the queue, and stores the remainder of the queue in its local $afterMe$ variable. The client then executes its critical section. On exit from the critical section, if the client's $afterMe$ queue is nonempty, it sends a *TokenGrant* message directly to the first node listed in the $afterMe$ queue; this *TokenGrant* message contains the contents of the sending node's $afterMe$ queue. When a client leaves its critical section, it also informs the master of this fact by sending a *DoneReport* message. On receiving this message, the master will remove the sending node from its *activeTokenQ*.

If the master has nodes in its *passiveTokenQ*, and the master's *activeTokenQ* is less than some predefined length *WarningLength*, the master sends a *ForwardToken* message to the node at the tail of the *activeTokenQ*. This message contains a list of nodes in the *passiveTokenQ*; when the recipient's current request is satisfied, it will pass the token directly to the first of the nodes in this message. After sending a *ForwardToken* message, the master moves the nodes in its *passiveTokenQ* into its *activeTokenQ*.

Each *ForwardToken* or *TokenGrant* message sent by the master is labelled with a "batch number" which uniquely identifies the corresponding group of requests. The batch number is passed along with the token, and each client remembers the highest batch number it has seen. This allows a client receiving a *ForwardToken* message to determine whether the *ForwardToken* applies to its previous request or its current request.

As an optimization, a client finishing with the token need not always send a *DoneReport* to the master. Correctness of the algorithm requires only that a *DoneReport* be sent if the client's $afterMe$ queue is empty; other *DoneReport* messages are optional and may be omitted. This provides some flexibility to optimize the performance of the algorithm. The code in figure 3 attaches a flag called *shouldReport* to the token; *shouldReport* is set when a client receives a *TokenGrant* or *ForwardToken* message from the master, and cleared when a client sends a *DoneReport*. A client sends a *DoneReport* if either

- the client's $afterMe$ queue is empty, or
- $shouldReport = True$ and the length of the client's $afterMe$ queue is less than *WarningLength*.

This policy improves the number of messages per lock acquisition in the heavy-load case.

4 Properties of the Algorithm

4.1 Correctness

In this section, we sketch proofs that the algorithm ensures mutual exclusion, and is free from deadlock and starvation.

Mutual Exclusion The algorithm ensures mutual exclusion because no client can be in its critical section without possessing the token, and the token can be held by at most one client node at a time. We say the master has the token if

1. the master's *activeTokenQ* and *passiveTokenQ* are both empty, or
2. the master's *passiveTokenQ* is empty, and a *DoneReport* message is enroute to the master from the node at the tail of the master's *activeTokenQ*.

We say a client N has the token if

1. N's variable *holding* is true, or
2. a *TokenGrant* message is enroute to N, or
3. a *DoneReport* message is enroute from N to the master, and a *ForwardToken* message is enroute from the master to N.

An exhaustive case analysis shows that at all times, exactly one node has the token. Since any node in its critical section must be a client whose *holding* variable is true, no node is ever in its critical section unless it has the token. Thus mutual exclusion is ensured.

Freedom from Deadlock Deadlock freedom requires that at all times, either no node has requested the token, or some progress is being made. At any time during execution, the system is in one of the following states:

- all requests that have reached the master have been satisfied; or
- some node is in its critical section; or
- a *TokenGrant* message is enroute to some node that has requested the token; or
- a *ForwardToken* message is enroute to some node.

In the first case, there are no outstanding requests for the token. In the other three cases, progress is being made — either a useful message is in transit, or some node is executing its critical section.

Freedom from Starvation No node can be starved because all requests are satisfied in the order in which the master receives them. Let the *tokenQ* be the result of appending the master's *activeTokenQ* and its *passiveTokenQ*. When the master receives a request for the token, the requester is appended to the tail of the *tokenQ*. Nodes receive the token in the order in which they appear in the *tokenQ*, so all requests are satisfied in the order in which the master received them.

4.2 Behavior Under Light Load

When requests are rare, the algorithm behaves like the traditional centralized scheme. When a client requests the token, two sequential messages are required to satisfy the request — the client sends a *TokenRequest* message to the master, and the master replies with a *TokenGrant*. When a client finishes the critical section, another message is required — the client sends a *DoneReport* to the master. Thus, the light-load synchronization delay is two, and the number of messages per lock acquisition in this case is three.

4.3 Behavior Under Heavy Load

Under heavy load, there are usually many outstanding requests at a time. The master groups these requests into batches; the master grants the token to the first client in the batch, and the token is passed directly from node to node within the batch. While one batch is being serviced, the master accumulates a new batch of requests in its *passiveTokenQ*. When the number of unserviced requests in the previous batch drops below *WarningLength*, the master schedules service for the new batch by sending a *ForwardToken* message to the last node in the previous batch.

Under heavy load, the token is passed directly from one client to the next, since the *ForwardToken* message normally reaches the last client in a batch before that client finishes with the token (*WarningLength* is chosen sufficiently large that this is the case; if all messages have the same latency, $WarningLength = 3$ is large enough.) Therefore the heavy-load synchronization delay is 1.

A batch of size $b > WarningLength$ requires a total of $2b + 2$ messages — one request for each client in the batch, one *TokenGrant* for each client, one *ForwardToken* from the master to the last client, and one *DoneReport*. (The last client will not send a *DoneReport*, since it will have received the *ForwardToken* containing the next batch.) The number of messages per lock acquisition in this case is $2 + 2/b$.

4.4 Comparison to Other Algorithms

Table 1 compares the performance of our algorithm and several other algorithms from the literature. The other algorithms are

- the "traditional" centralized algorithm (not published, but part of the folklore),
- the DAG-based algorithm of [11], using the centralized star topology (the topology that gives best performance)
- the quorum agreement algorithm of [9], using the centralized quorum agreement $Q = Q^{-1} = \{\{M\}\}$, where M is a single node (the quorum agreement that gives best performance), and
- the DAG-based algorithm of [11], using a non-centralized topology,
- the quorum agreement algorithm of [9], using a non-centralized quorum agreement.
- the non-centralized token-based algorithms of [4, 18, 16, 20]

As the table shows, our algorithm dominates all existing non-centralized algorithms — it is better than every one in some category, and at least as good in all the other categories. Our algorithm is the only one that achieves light-load synchronization delay 2 and heavy-load synchronization delay 1, while still sending a constant number of messages per lock acquisition. Our algorithm also uses the fewest messages in the heavy-load case.

algorithm	Light-load sync. delay	Heavy-load sync. delay	Light-load messages/ lock acq.	Heavy-load messages/ lock acq.	Worst-case messages/ lock acq.
our centralized	2	1	3	$2+2/b$	4
traditional centralized	2	2	3	3	3
DAG-based (centralized)	3	1	3	3	3
quorum-agreement-based (centralized)	3	1	4	$3+1/b$	4
DAG-based (non-centralized)	3	1	≥ 3	≥ 3	≥ 4
quorum-agreement-based (non-centralized)	3	1	> 4	> 4	> 4
non-centr. token-based alg. of [4, 18, 16, 20]	2	1	$O(N)$	$O(N)$	$O(N)$

Table 1: Performance measures for our algorithm and several existing algorithms.

4.5 Fault Tolerance

In our protocol, failures may cause the token, or some outstanding token requests, to be lost. Losing token requests is not harmful since processes can always repeat their requests if the previous requests have not been satisfied. As to the lost token, it can be regenerated in our protocol using a scheme proposed in [12], which is equally applicable to our centralized as well as to non-centrilized algorithms. Therefore, the fault-tolerance properties of our protocol are the same as those of the non-centralized token-based algorithms: temporary failures are tolerated but availability of the system is not guaranteed *during* failures. If availability of the service during failures is of greater importance than the performance concern, then neither our protocol nor the existing non-centralized token-based protocols are appropriate, and quorum-based protocols should be used.

5 Further Improvements of the Algorithm

This section discusses two enhancements to the basic algorithm. First, we discuss how to improve access time for nodes that use the lock heavily, by migrating the central node or the token. Second, we address the concern that the central node may become the bottleneck in the system.

5.1 Improving Response Time for Frequent Requesters

Having the master node universally known makes it possible to tune the performance of the algorithm to the specific needs of the application. For example, when a node is releasing the token and there are no outstanding requests, the node has the discretion to retain the token (if it anticipates requesting the token again in the near future), send the token to another node (if it believes this node is most likely be the next to request the token), or return the token to the central node (if it has no opinion). Since the light load synchronization delay is 0 when the requesting node has the token, 2 when a non-master requests the token held by the master, and 3 when a non-master requests the token held by another non-master,[1] it makes sense to place the unclaimed token at the most likely requester or at least at the master. Therefore, if the application has information on the anticipated pattern of lock requests, it can use this information to improve performance. Alternatively, the master can collect statistics on the prior behavior of the system and, based on this information, send instructions (piggybacked with the *TokenGrant* and *ForwardToken* messages) as to what to do with the token if there are no outstanding requests.

Further, it is obviously beneficial to make the node most frequently requesting the token the master node. Indeed, in this case, if the token is occasionally requested by another node, the most frequent user can always acquire it with light load synchronization delay 2. Our algorithm can make it possible for the application to request the migration of the master. (During the transition period, the old master will forward requests to the new one using the technique described in [5].) Or, again, the decision to migrate can be made by the master itself based on the prior behavior of the system.

Future work is required to refine the ideas outlined here. What this subsection shows is that a centralized approach opens up many possibilities for performance improvement.

5.2 Improving Load Sharing

It may appear that the master node in our algorithm can become the bottleneck of the system. However, the amount of work the master does per lock request is small. Indeed, all it has to do is enqueue the request into the locally kept token queue. The remaining work (receiving and processing the *DoneReport* message and sending out the *TokenGrant* or *ForwardToken* message with the newly accumulated requests) is amortized over the requests arrived since the last time the master sent out a *TokenGrant* or *ForwardToken* message. Under heavy load, when the master can possibly become the bottleneck, the number of accumulated requests will be high, and, hence the amount of work per request will be low. Also, one can assign different masters

[1] Note that the last case never happens in the algorithm as presented in the previous section.

to synchronize access to different resources, thus sharing the load on a per-resource basis. Although real measurements are needed to conclusively substantiate our claim, we assert that the master node is not likely to be the bottleneck in the system.

However, in the unlikely event of the central node becoming temprorarily overloaded, a technique recently proposed by Helary et al. [8] can be used to switch dynamically without any service interruption from our centralized to a distributed token based algorithm. This combined protocol enables the system to use our efficient centralized algorithm most of the time while resorting to a non-centralized algorithm during the rare periods when the central node is overloaded.

In the approach by Helary et al., each node maintains a variable *father* that points to another node. Together, these variables specify a rooted tree structure over the nodes, with the token holder at the root. The token can be either lent (in which case it carries the identity of the lender) or given. After having used the token, a node returns it to the lender if the token was lent or otherwise keeps it. A node that wants to get the token sends the request to its father. When a node i receives the request from a node j, it behaves differently depending on its *behavior* variable, which may be either *transit* or *proxy*. In the transit mode, i simply forwards the request to its father and, having done that, sets $father_i$ to j. The token is then sent directly to j, bypassing i. In the proxy mode, i requests the token *for itself.* On receiving the token, if the token is lent, i sends it to j and sets $father_i$ to the lender[2]. Otherwise, i *lends* the token to j and sets $father_i$ to *nil.*

This mutual exclusion algorithm is very general and can emulate different existing algorithms by setting the behavior variables of the nodes in the system to different combinations. For example, by statically assigning all nodes to the *transit* mode, the distributed algorithm of Naimi and Trehel [10] is obtained, while if the tree structure specified by the father variables is actually a star with the central node being *proxy* and all peripheral nodes *transit*, we get the existing centralized algorithm. Moreover, the correctness of the algorithm is not affected by changing behavior variables dynamically at arbitrary times. We can use this fact to switch back and forth from a centralized to a distributed algorithm by assigning two additional variables, *master* and *rate* to the token. In the centralized mode, one (central) node in the system is *proxy*, all other nodes are *transit*, and the token variable *master* stores the identity of the proxy node. If at some point, the

[2] Actually, in [8], i sets its father variable to the node from which the token is received in this case. The behavior we describe here makes the algorithm slightly simpler and more efficient.

central node becomes overloaded, it sets itself to the transit mode and changes the *master* variable of the token to *nil* at the first opportunity (i.e., next time the token is returned to it). After that, the system will evolve according to the Naimi and Trehel algorithm, with the load shared. The token variable *rate* records the rate of requests for the token as an approximation of load. When the rate becomes low, and the token's *master* variable is *nil*, some node i holding the token, at its discretion, changes its mode to *proxy* and sets the token's *master* variable to i, so that no other node can change to proxy. From now on, the algorithm will become centralized again.

As simple as this idea is, the details as to how to make the system in the centralized mode behave according to our algorithm rather than the existing centralized algorithm, and how to make an arbitrary tree of nodes with a single proxy node always evolve into a star structure[3], are rather complicated and will be reported separately.

6 Conclusion

We have presented a new centralized token-based algorithm for distributed mutual exclusion that combines the advantages of the existing centralized and token-based algorithms: it takes only two messages to acquire the lock when it is not held by another process, and only one message to pass the lock from one process to another. At the same time, message traffic generated by our algorithm is significantly lower than message traffic generated by the non-centralized algorithms. In fact, under heavy load, i.e., when message traffic is of most concern, our algorithm generates about one third less traffic than the existing centralized algorithms. We also show that, within our approach, the performance of the algorithm can be further improved by making use of information on the pattern of lock requests, if available. Finally, we discuss the issues of fault tolerance and load sharing properties of our protocol. We show that our protocol exhibits the same fault tolerance as non-centralized token based protocols and outline the way that would allow the system to switch dynamically from our centralized to a distributed algorithm to deal with rare periods when the central node gets overloaded.

[3] In a general tree, requests from the nodes that are far away from the root take multiple hops to reach the central node, and, hence, the performance of the algorithm degrades.

References

[1] D. Agrawal and A. El Abbadi. An efficient solution to the distributed mutual exclusion problem. In *Proc. of Symp. on Principles of Distributed Computing*, pp. 193-200, 1989.

[2] E. A. Arnold, F. J. Bitz, E. C. Cooper, H. T. Kung, R. D. Sansom, and P. A. Steenkiste. The design of Nectar: A network backplane for heterogeneous multicomputers. In *Proc. 3rd Conf. on Arch. Support for Programming Lang. and Operating Systems*, April 1989.

[3] O. S. F. Carvalho and G. Roucairol. On mutual exclusion in computer networks. *Comm. of the ACM*, 26(2), pp. 146-148, February 1983.

[4] Y-I. Chang, M. Singhal, and M. T. Liu. A dynamic token-based distributed mutual exclusion algorithm. In *Proc. 10th IEEE Int'l Phoenix Conf. on Computers and Communications*, pp. 240-246, 1991.

[5] R. J. Fowler. *Decentralized Object Finding Using Forwarding Addresses.* Ph.D. dissertation. University of Washington, December 1985. UW CS Tech. Report 85-12-1.

[6] D.K. Gifford. Weighted voting for replicated data. In *Proc. of the 7th ACM Symposium on Operating Systems Principles*, pp. 150-159, December 1979.

[7] H. Garcia-Molina and D. Barbara. How to assign votes in a distributed system. *J. of the ACM,* 32(4), pp. 841-860, October 1985.

[8] J. M. Helary, A. Mostefaoui, M. Rainal. A very general information structure for tree based distributed mutual exclusion algorithms. Research report 649 , IRISA, University of Rennes, France, March 1992.

[9] M. Mizuno and M. L. Neilsen. A token based distributed mutual exclusion algorithm based on quorum agreements. In *Proc. of the 11th Int. Conf. on Distributed Computing Sys.,* pp. 361-368, 1991.

[10] M. Naimi and M. Trehel. An improvement of the $\log(n)$ distributed algorithm for mutual exclusion. In *Proc. of the 7th IEEE Int. Conf. on Distributed Computing Sys.*, pp. 371-375, 1987.

[11] M. L. Neilsen and M. Mizuno. A DAG-based algorithm for distributed mutual exclusion. In *Proc. of the 11th Int. Conf. on Distributed Computing Sys.,* pp. 354-360, 1991.

[12] S. Nishio, K. F. Li, and E. G. Manning. A time-out based resilient token transfer algorithm for mutual exclusion in computer networks. In *Proc. of the IEEE 9th Int. Conf. on Distributed Computing Sys.*, pp. 386-393, 1989.

[13] M. Maekava. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Trans. on Comp. Systems.* 3(2), May 1985.

[14] M. Rabinovich and E. D. Lazowska. The dynamic tree protocol: avoiding "graceful degradation" in the tree protocol for distributed mutual exclusion. In *Proc. 11th IEEE Int'l Phoenix Conf. on Computers and Communications*, April 1992.

[15] G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Comm. of the ACM*, 7(1), pp. 61-77, February 1983.

[16] G. Ricart and A. K. Agrawala. Authors' response to 'On mutual exclusion in computer networks' by Cavalho and Roucairol. *Comm. of the ACM*, 26(2), pp. 147-148, 1983.

[17] M. D. Schroeder, A. D. Birrell, M. Burrows, H. Murray, R. M. Needham, T. L. Rodeheffer, E. H. Satterthwaite, and C. P. Thacker. *Autonet: A high-speed, self-configuring local area network using point-to-point links.* Research Report 59, DEC Systems Research Center, April 1990.

[18] M. Singhal. A dynamic information-structure mutual exclusion algorithm for distributed systems. In *Proc. of the 9th International Conf. on Distributed Computing Sys.,* pp. 70-78, 1989.

[19] M. Singhal. A heuristically-aided algorithm for mutual exclusion in distributed systems. *ACM Trans. on Computer Systems,* 38(5), pp. 651-662, May 1989.

[20] I. Suzuki and T. Kasami. A distributed mutual exclusion algorithm. *ACM Trans. on Comp. Systems*, 3(4), pp. 344-349, 1985.