# A Tree-Based Algorithm for Distributed Mutual Exclusion

KERRY RAYMOND
University of Queensland

We present an algorithm for distributed mutual exclusion in a computer network of $N$ nodes that communicate by messages rather than shared memory. The algorithm uses a spanning tree of the computer network, and the number of messages exchanged per critical section depends on the topology of this tree. However, typically the number of messages exchanged is $O(\log N)$ under light demand, and reduces to approximately four messages under saturated demand.

Each node holds information only about its immediate neighbors in the spanning tree rather than information about all nodes, and failed nodes can recover necessary information from their neighbors. The algorithm does not require sequence numbers as it operates correctly despite message overtaking.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.4.1 [**Operating Systems**]: Process Management—*mutual exclusion, synchronization*

General Terms: Algorithms

Additional Key Words and Phrases: Critical section, decentralized systems

## 1. INTRODUCTION

We propose a new algorithm for distributed mutual exclusion for a computer network of $N$ nodes, communicating by messages rather than shared memory. In keeping with earlier work on this problem, we assume that message delivery is guaranteed by the communications network, but neither the time nor order of message arrival can be predicted. Initially we shall assume that nodes are completely reliable, and node failure will be considered in a later section.

Ricart and Agrawala [3] proposed an algorithm that required $2^*(N - 1)$ messages exchanged for each critical section entry, while the algorithm of Suzuki and Kasami [4] requires at most $N$ messages. Maekawa [2] further reduces the number of messages per critical section entry to $O(\sqrt{N})$. The performance of our algorithm depends on the precise topology of the network spanning tree used, but the average number of messages required is $O(\log N)$.

For a node to obtain the mutual exclusion, the algorithms of Ricart and Agrawala and Suzuki and Kasami require a REQUEST message to be sent to all of the other $N - 1$ nodes. Consequently each node must hold information

Author's address: Department of Computer Science, University of Queensland, St. Lucia, Queensland, 4067, Australia.

relating to all other nodes (REPLY_DEFERRED in [3], RN and LN in [4]). In Maekawa's algorithm, a REQUEST is sent to approximately $\sqrt{N}$ nodes, each of which must maintain some information about that request. In our algorithm, each node communicates only with neighboring nodes of the spanning tree, and holds information pertaining only to those neighbors.

In keeping with earlier work, our algorithm is concerned with implementing mutual exclusion at the node level. If multiple requests for the mutual exclusion can be generated within a node, then these requests must be serialized [3]. If not serialized, it would be possible for a node to circulate the mutual exclusion among its internal components, starving the other nodes.

## 2. AN INFORMAL DESCRIPTION OF OUR ALGORITHM

In our algorithm, we regard the nodes as being arranged in an unrooted tree structure, as shown in Figure 1. All messages used in our algorithm are sent along the (undirected) edges of this tree. The tree may either be a minimal spanning tree of the actual network topology, or merely a logical structure imposed on the complete network assumed by [2–4].

There is no need for each node to be aware of the tree as a whole. It is sufficient that each node knows of the existence of its neighbors in the tree. In Figure 1, node A knows that it is connected to three distinct neighbor nodes B, C, and D. A does not need to know the location (or even the existence) of nodes E and F.

As in the Suzuki and Kasami algorithm, the privilege to enter the critical section equates to the possession of a token, the PRIVILEGE message. The privilege is always held by one node (except for the transient state when the PRIVILEGE message is in transit from one node to another). When no node wishes to enter the critical section, the last node to use the privilege continues to hold it. Each node has a variable HOLDER, which indicates the location of the privilege relative to the node itself. Either a node X holds the privilege itself, or else it is located in a particular subtree of X. Suppose that the node E of Figure 1 holds the privilege, then

| | |
|---|---|
| $HOLDER_A = D$ | E is within the D-subtree w.r.t. A |
| $HOLDER_B = A$ | E is within the A-subtree w.r.t. B |
| $HOLDER_C = A$ | |
| $HOLDER_D = E$ | |
| $HOLDER_E = self$ | |
| $HOLDER_F = D$ | |

If we were to represent $HOLDER_X = Y$ as a directed edge from X to Y, then the combined HOLDER information of all nodes represents a single directed path from each node to the privileged node, as shown in Figure 2.

When a nonprivileged node (e.g., A in Figures 1 and 2) wishes to enter the critical section, it sends a REQUEST message to $HOLDER_A$, i.e., D. On receiving the REQUEST message, the nonprivileged node D sends a REQUEST message to $HOLDER_D$, i.e., E. Thus a series of REQUEST messages travels along the path between the requesting node A and the privileged node E.

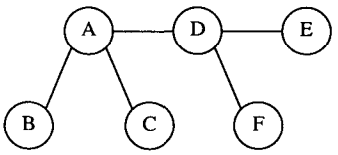
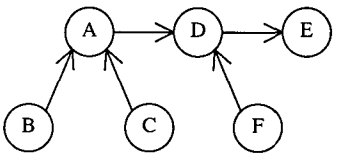



Fig. 1. Nodes arranged as a tree.

Fig. 2. Tree with HOLDER-directed edges (E holds the privilege).



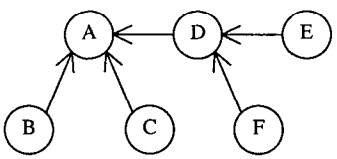




Fig. 3. Tree with HOLDER-directed edges (A holds the privilege).

When the privileged node E no longer requires the privilege, it sends the PRIVILEGE message to one of its neighbors who has requested the privilege (i.e., E has received a REQUEST from this neighbor). Thus E will send the PRIVILEGE message to D, and $\text{HOLDER}_E$ becomes D. Node D did not request the privilege for itself, but on A's behalf. So D sends the PRIVILEGE message to A, setting $\text{HOLDER}_D$ to A. On receiving the PRIVILEGE message, node A becomes the privileged node ($\text{HOLDER}_A$ := self) and A may enter the critical section.

Note the assignment of $\text{HOLDER}_E$ and $\text{HOLDER}_D$ when the PRIVILEGE message is sent. This ensures that when the PRIVILEGE message is received at A, the directed path tree of Figure 2 has become the tree shown in Figure 3, i.e., the HOLDER variables collectively maintain directed paths from each node to the privileged node.

It is important to note that this algorithm is not a "fully distributed" algorithm as defined by Ricart and Agrawala. By their definition, all nodes must directly participate in the decision to grant a mutual exclusion request. Such algorithms are condemned to be at best $O(N)$. Our algorithm, like Maekawa's, uses a "surrogate mechanism" in which a node X requests another node Y to act on X's behalf. Therefore node Y may act "collectively," reducing the number of messages required to effect mutual exclusion.

## 3. THE ALGORITHM

To implement the algorithm described in the previous section, each node X must hold certain information (described in the next section). The algorithm involves two types of communication, the sending of a REQUEST message and the sending of a PRIVILEGE message.

### 3.1 Information Held by Each Node

HOLDER
: Values: "self" or the name of one of the immediate neighbors. Indicates the relative position of the privileged node with respect to this node X.

USING
: A Boolean value. USING indicates if X is currently executing the critical section. Naturally, USING $\Rightarrow$ HOLDER = self.

REQUEST_Q
: A First-In-First-Out queue. The possible elements of the queue are the names of immediate neighbors and "self." REQUEST_Q holds the names of those neighbors that have sent a REQUEST message to X, but have not yet been sent the privilege in reply. "self" is placed in the queue when X wishes to obtain the privilege for its own use. As there is at most one occurrence of each name in a REQUEST_Q, the maximum size of a REQUEST_Q is number of neighbors + 1 (for self).

ASKED
: ASKED, a Boolean, is true when a nonprivileged node X has sent a REQUEST message to the current $\text{HOLDER}_X$; false otherwise. ASKED prevents unnecessary REQUEST messages from being sent, and also ensures that REQUEST_Qs do not contain duplicate entries.

### 3.2 Sending a PRIVILEGE Message

The necessary requirement for sending a PRIVILEGE message is

$$\text{HOLDER} = \text{self} \wedge \neg \text{USING} \wedge \text{REQUEST\_Q} \neq \text{empty} \wedge \text{head}(\text{REQUEST\_Q}) \neq \text{self}$$

(i.e., the node must hold the privilege but not be using it, and the oldest request for the privilege has come from another node). Superficially it may seem impossible that the head of REQUEST_Q of the privileged node could be "self." However this situation may occur immediately after this node has received the PRIVILEGE message. In this situation, the node will then begin to use the privilege (removing "self" from the head of the REQUEST_Q).

Therefore our algorithm uses a routine ASSIGN_PRIVILEGE given below. The recipient of the privilege (the HOLDER-to-be) is the node at the head of REQUEST_Q (the oldest request), and this node must be removed from REQUEST_Q as its request is being honored. Since the current privileged node will not have sent a REQUEST message to the HOLDER-to-be, ASKED must be set to false.

```
ASSIGN_PRIVILEGE:
    if HOLDER = self ∧ ¬ USING ∧ REQUEST_Q ≠ empty
    then
      HOLDER := dequeue (REQUEST_Q)
      ASKED := false
      if HOLDER = self
      then
        USING := true
        ⟨initiate entry into critical section⟩
      else
        send PRIVILEGE to HOLDER
```

### 3.3 Sending a REQUEST Message

The necessary requirement for sending a REQUEST message to the holder (w.r.t. this node) is that this node does not have the privilege but wants it (either for itself or others). Furthermore a REQUEST message should not be sent if one has already been sent to the holder. The sending of a REQUEST message has little impact on the state of a node. It causes no change to HOLDER, REQUEST_Q, or USING. Only ASKED becomes true. The algorithm for MAKE_REQUEST is:

```
MAKE_REQUEST:
    if HOLDER ≠ self ∧ REQUEST_Q ≠ empty ∧ ¬ ASKED
    then
      send REQUEST to HOLDER
      ASKED := true
```

The variable ASKED is true when "self" is in the REQUEST_Q of a neighboring HOLDER (or will be after the arrival of the REQUEST message), and false otherwise. Thus the ASKED variable ensures that duplicate entries of "self" cannot occur in a neighbor's REQUEST_Q. The serialization of any internal requests (see Section 1, INTRODUCTION) ensures that duplicate entries of "self" cannot occur in the local REQUEST_Q. Therefore REQUEST_Qs are indeed bounded, and so there is no potential for "flooding," even under heavy load conditions.

### 3.4 Four Events

There are four events which can alter the assignment of privilege and/or necessitate the sending of a REQUEST message. Consequently our algorithm consists of four parts corresponding to each of the four events, as shown below.

The node wishes to enter the critical section:

```
enqueue (REQUEST_Q, self); ASSIGN_PRIVILEGE; MAKE_REQUEST
```

If this is the privileged node, then ASSIGN_PRIVILEGE will allow this node to enter the critical section. If this is not the privilege node, MAKE_REQUEST may send a REQUEST to obtain the privilege.

The node receives a REQUEST message from neighbor X:

```
enqueue (REQUEST_Q, X); ASSIGN_PRIVILEGE; MAKE_REQUEST
```

If this node is the holder, ASSIGN_PRIVILEGE may send the privilege to the requesting node. If this node is not the holder, MAKE_REQUEST may propagate the REQUEST to obtain the privilege.

The node receives a PRIVILEGE message:

```
HOLDER := self; ASSIGN_PRIVILEGE; MAKE_REQUEST
```

ASSIGN_PRIVILEGE may pass the privilege to another node, or initiate a local entry to the critical section. If the privilege is passed to another node, MAKE_REQUEST may request that the privilege be returned.

The node exits the critical section:

```
USING := false; ASSIGN_PRIVILEGE; MAKE_REQUEST
```

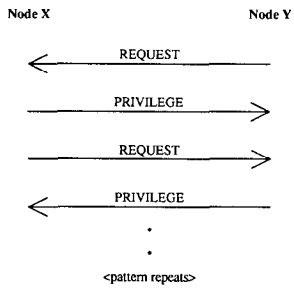



Fig. 4. Logical pattern of messages between nodes X and Y.

On releasing the mutual exclusion, ASSIGN_PRIVILEGE may pass the privilege to another node, and MAKE_REQUEST may then request the return of the privilege.

Note that these pieces of code must execute in local mutual exclusion.

### 3.5 Message Overtaking

Unlike earlier algorithms, the acyclic tree structure employed by our algorithm restricts the amount of conflicting information resulting from varying message transmission times and message overtaking to conflicts between pairs of neighboring nodes. Message traffic between a pair of neighboring nodes must conform to a logical pattern, and hence there is no need for message sequence numbers to enforce the physical order of arrival.

Consider the sequence of messages between a pair of neighboring nodes X and Y, shown in Figure 4. Assume that initially node X (or beyond) holds the privilege.

The only message overtaking that may occur is when a PRIVILEGE message is sent from node X to node Y, say, closely followed by a REQUEST message from node X to node Y. That is, X is giving the privilege to Y but immediately (or very soon after) requires the privilege to be returned.

Upon receiving the overtaking REQUEST message, node Y could be programmed to recognize that overtaking has occurred (since the next logical message must be a PRIVILEGE message) and defer the processing of the REQUEST message until after the receipt and processing of the overtaken PRIVILEGE message. However it is not necessary to do so, as the algorithm (as presented) is insensitive to such overtaking.

If the REQUEST message arrives at Y before the PRIVILEGE message, then X's request will be queued in REQUEST_$Q_Y$. Since Y is not yet the privileged node, ASSIGN_PRIVILEGE will not send a message or cause any other state change. Since X has sent the PRIVILEGE message to Y, it implies that $ASKED_Y$ must be true, and hence MAKE_REQUEST will not send a message or cause any state change.

When the PRIVILEGE message finally arrives at Y, either Y will enter the critical section or Y will send the PRIVILEGE message to the neighbor at the head of REQUEST_$Q_Y$ (which will not be X). The early arrival of the REQUEST message does not invalidate the operation of the algorithm in any way.

## 4. PROOF OUTLINES

### 4.1 Mutual Exclusion Is Assured

To ensure mutual exclusion, it is necessary that, at most, one node regards itself as privileged. A node becomes privileged when it receives a PRIVILEGE message. A privileged node becomes nonprivileged when it sends exactly one PRIVILEGE message to another (neighboring) node. PRIVILEGE messages cannot be sent by unprivileged nodes. Provided only one node is privileged initially, there will continue to be either only one privileged node, or else there is no privileged node (during the finite time it takes to transmit the PRIVILEGE message).

### 4.2 Deadlock Is Impossible

If no node is in the critical section and there are one or more nodes wishing to enter the critical section, yet unable to do so, then the system is deadlocked. This could occur as a consequence of any of the following:

(a) No node is privileged and hence the privilege cannot be given to other nodes.
(b) The privileged node is not aware that other nodes require the privilege.
(c) The PRIVILEGE message does not eventually reach a node which has requested it.

From Section 4.1 we know that one node must be privileged (or will shortly become privileged).

The collective ASKED variables ensure that (after a finite amount of message transmission time) there is a sequence of REQUEST messages (for which no PRIVILEGE message has been received in reply) between each node requiring the privilege and the privileged node, using the collective HOLDER variables to route these REQUEST messages towards the privileged node.

It is impossible for the PRIVILEGE message to be passed through the tree so that REQUEST messages never arrive at the currently privileged node. As the tree structure is acyclic, the PRIVILEGE messages cannot *outrun* the REQUEST messages indefinitely. The only possible way for the PRIVILEGE message to evade the pursuing REQUEST message would involve the PRIVILEGE message traveling from node A to node B, while the REQUEST message is traveling from node B to node A. However the logical (and consequent physical) order of messages on an edge of the tree prevents this (see Section 3.5). The PRIVILEGE message would not be sent from node A unless node A had received a REQUEST from node B to which node A had not replied. If node B had sent such a REQUEST message, then node B would not have sent the current REQUEST message (due to $\text{ASKED}_B$).

Thus the privileged node must eventually become aware that other nodes require the privilege. Furthermore the chain of successive REQUEST messages causes the collective REQUEST_Qs to provide a path from the privileged node to a node that wishes to enter the critical section. Hence the PRIVILEGE message is forwarded to a node that has requested the privilege.

In summary, our algorithm is deadlock-free, mostly due to the acyclic nature of a tree eliminating the potential for any "circular wait" situations.

### 4.3 Starvation Is Impossible

Suppose node Y holds the privilege (or will do so after the transmission of the PRIVILEGE message is complete). When a node X requires the privilege, the collective ASKED variables ensure that (after a finite amount of message transmission) there is a chain of requests between the requesting node X and the privileged node Y. Some of the REQUEST messages may be a direct consequence of X's requirement, while others may have been sent in response to the requirements of other nodes; such REQUEST messages now represent X's requirements as well.

More rigorously, let $P_1, P_2, \ldots, P_k$ be the sequence of nodes along the path from $X(P_1)$ to $Y(P_k)$. The tree structure ensures that such a path is unique, and that $k \leq N$. Then it follows that

$$P_1 \in \text{REQUEST\_Q}_{P_1}$$
$$\text{and} \quad P_{i-1} \in \text{REQUEST\_Q}_{P_i} \quad \text{for} \quad 2 \leq i \leq k$$

Consider the vector $[\![M_1, M_2, \ldots M_k]\!]$ where

$$M_1 \text{ is the position of } P_1 \text{ in REQUEST\_Q}_{P_1}$$
$$\text{and} \quad M_i \text{ is the position of } P_{i-1} \text{ in REQUEST\_Q}_{P_i} \quad \text{for} \quad 2 \leq i \leq k$$

The element at the head of the queue is numbered as position 1, the next element as position 2, etc. As the queues are held in FIFO order, the position of a particular element cannot increase.

Since the longest possible path in a tree is of length at most N, and the size of a REQUEST_Qs is at most N (self plus all other N − 1 nodes as neighbors), the vector can have only a finite number of possible values. Furthermore, vectors can be ranked in a strong total order by the lexicographic "<" operator.

Each of the possible actions of the privileged node Y (described below) reduces the value of the vector, and therefore successive operations of the successive privileged nodes must reduce the vector to $[\![1]\!]$ (i.e., X is the privileged node, and "self" is at the head of X's REQUEST_Q), which allows X to enter the critical section.

Consider the possible values of $M_k$.

If $k > 1$ and $M_k = 1$, then $P_{k-1}$ is at the end of $\text{REQUEST\_Q}_{P_k}$, and hence the PRIVILEGE will be sent to $P_{k-1}$ (i.e., towards X). If $k = 1$ and $M_1 = 1$, then X holds the privilege and X is at the head of X's REQUEST_Q; hence X will enter the critical section. In both cases, the effect on the vector will be:

$$[\![M_1, M_2, \ldots, M_{k-1}, 1]\!] \rightarrow [\![M_1, M_2, \ldots, M_{k-1}]\!]$$

If $k > 1$ and $M_k > 1$, then $P_{k-1}$ is not the element at the head of $\text{REQUEST\_Q}_{P_k}$. If $P_k$ is at the head of its own REQUEST_Q, then $P_k$ will enter the critical section. The effect on the vector will be:

$$[\![M_1, M_2, \ldots, M_{k-1}, M_k]\!] \rightarrow [\![M_1, M_2, \ldots, M_{k-1}, M_k - 1]\!]$$

(Note that if $k = 1$ and $M_k > 1$, then X cannot be at the head of its own REQUEST_Q, so this situation does not occur.)

If $k > 1$ and $M_k > 1$ and Z (a neighbor of $P_k$) is at the head of $P_k$'s REQUEST_Q, then the PRIVILEGE will next be sent to Z (i.e., away from X). Since $P_k$'s

REQUEST_Q will still be nonempty, a REQUEST message will also be sent to Z to ensure that the privilege is returned. If $k = 1$ and $M_1 > 1$, then node X holds the privilege but must pass the privilege to satisfy an earlier request from Z, one of X's neighbors. So X will send the PRIVILEGE followed by a REQUEST for the return of the privilege to Z. In either case, the effect on the vector is:

$$[\![M_1, M_2, \ldots, M_k]\!] \rightarrow [\![M_1, M_2, \ldots, M_k - 1, M_{k+1}]\!]$$

The claim that this operation reduces the value of the vector depends on the finite upper bounds of k and $M_i$. It is impossible for the vector to grow infinitely long, or for positions in a REQUEST_Q to become infinitely large.

Hence even the most remote node X cannot be overlooked. Once X's REQUEST message has propagated to either the privileged node or another requesting node, X is guaranteed to enter the critical section eventually.

## 5. COST OF THE ALGORITHM

Like [2–4], we will consider the number of messages required to effect an entry to a critical section. For our algorithm, the upper bound for the number of messages per critical section is

$2D$—where $D$ is the diameter (longest path length) of the tree

This worst case occurs when the privilege is passed from node A to node B, where nodes A and B have the greatest possible distance between them. It takes $D$ REQUEST messages originating at A, and $D$ PRIVILEGE messages originating from B, to pass the mutual exclusion from B to A.

The worst possible topology for this algorithm is a straight line arrangement as shown in Figure 5.

The diameter, $D$, of such a topology is $N - 1$, and thus the number of messages sent (i.e., $2^*(N - 1)$) is comparable with Ricart and Agrawala's algorithm and worse than the algorithm of Suzuki and Kasami. However this worst-case behavior occurs only in the pathological situation when the privilege is shuttled between the nodes at either end of the line. If all nodes are equally likely to require the privilege, then the average distance between the requesting node and the privileged node is $(N + 1)/3$, and thus the number of messages sent is

$$\approx \frac{2N}{3},$$

an improvement over the $N$ messages required by Suzuki and Kasami's algorithm.

The best topology for our algorithm is a radiating star formation, as illustrated in Figure 6.

If the valency of each nonleaf node of the star is $K$, then the diameter of the tree is

$$2\left\lceil \log_{K-1}\left(\frac{(N-1)(K-2)}{K} + 1\right)\right\rceil.$$

Thus the worst case for this topology is $O(\log_{K-1} N)$, which is better than Maekawa's $O(\sqrt{N})$ algorithm. It should be noted that the diameter of the tree decreases as the valency increases. Thus trees with a high fanout are preferable.

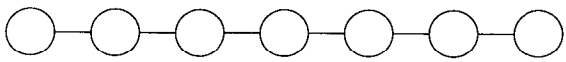


Fig. 5. Straight line topology.

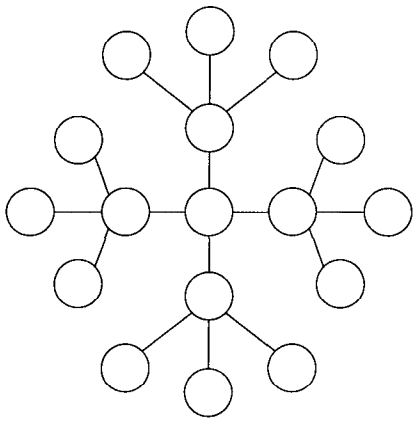


Fig. 6. Radiating star topology.

For the radiating star topology, we cannot find any simple formula for the average distance between nodes. The majority of the nodes exist on the outer rim of the tree, and for many pairs of these nodes the distance between them will be the diameter of the tree. Thus as N increases, the average distance between a pair of nodes approaches (but does not reach) the diameter of the tree. So $2*D$ messages represents an unreachable upper bound for this topology.

The diameter of a given arbitrary tree can always be calculated, and extensive empirical measurements on randomly constructed trees show that the diameter of such trees is typically $O(\log N)$. In any case, if the tree structure is logically imposed upon the underlying network, then pathological cases (e.g., the straight line) where the diameter is not $O(\log N)$ can be avoided in favor of trees which approximate a radiating star formation.

Thus we make the claim that the number of messages exchanged by our algorithm is typically $O(\log N)$, which is supported by the results of extensive simulations. Therefore we believe that, in most situations, our algorithm sends fewer messages than other reported algorithms [2–4].

### 5.1 Performance Under Heavy Demand

The preceding complexity analysis was based upon a chain of REQUEST messages and a subsequent chain of PRIVILEGE messages traveling between the requesting and the privileged nodes. This assumes that there are no other nodes requesting the privilege, that is, there is little demand for the privilege. When many nodes wish to obtain the privilege, REQUEST messages sent by a requesting node are not usually forwarded all the way to the privileged node. Instead the REQUEST message will arrive at a node X where $\text{ASKED}_X$ is true. Thus the REQUEST message sent earlier from X to $\text{HOLDER}_X$ represents the interests of all requesting nodes reachable from the privileged node via X.

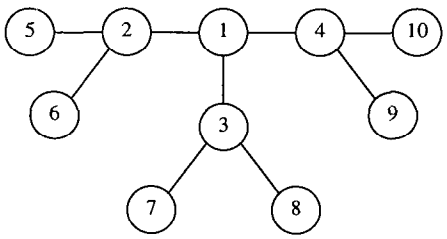



Fig. 7. Example topology.

When node X receives the privilege from its neighbor Y, all of X's other neighbors and X itself will use the privilege (if requested prior to Y's request) before it is returned to Y. The PRIVILEGE message visits a number of nodes within a subtree, and so the average distance that the PRIVILEGE message travels between nodes desirous of entering the critical section is much less than the diameter of the tree.

Thus our algorithm has the curious (but delightful) property of improved performance under heavy demand. As the number of nodes wishing to enter the critical section increases, the number of messages sent per entry to the critical section decreases. When the system is saturated (all nodes but the privileged node are waiting to enter the critical section), approximately four messages are sent per critical section entry. This constant behavior was first observed during simulations, and its explanation eluded us for some time.

To appreciate why only four messages are sent per critical section entry when the system is saturated, we must consider the path of the successive PRIVILEGE messages through the tree. (Note that only one PRIVILEGE message exists at any time.) The path of the PRIVILEGE messages in a saturated system is essentially a tree traversal, albeit a somewhat unordered one. For example, consider the topology shown in Figure 7. A possible path for the PRIVILEGE messages through this tree would be:

$$\underline{1}\ 2\ \underline{5}\ 2\ \underline{6}\ 2\ 1\ 4\ \underline{9}\ 4\ \underline{10}\ \underline{4}\ 1\ \underline{3}\ \underline{8}\ 3\ \underline{7}\ 3$$

where the underlined node numbers indicate that the node entered the critical section at that time. Despite the haphazard order in which nodes enter the critical section (determined by their position in REQUEST_Qs), the path of the PRIVILEGE messages travels along each of the $N - 1$ edges exactly twice in order to bring the privilege to all $N$ nodes. This behavior is a consequence of the tree structure, and will occur irrespective of the actual topology.

A PRIVILEGE message travels along an edge from node A to node B in reply to a REQUEST message, which traveled along the edge from node B to node A. Hence a total of $4*(N - 1)$ messages are sent among the $N$ nodes when the system is saturated, and so the number of messages per critical section entry is:

$$\frac{4(N-1)}{N} \approx 4.$$

## 6. TWO VARIATIONS ON THE ALGORITHM

We present two variations to our algorithm, which can be applied either separately or together to improve its efficiency.

### 6.1 The Piggyback Strategy

On exiting a critical section, the privileged node may send both a PRIVILEGE message (in ASSIGN_PRIVILEGE) and a REQUEST message (in MAKE_REQUEST) to a neighbor. If the actual transmission of the PRIVILEGE message is deferred until the need for sending the REQUEST message has been determined, then the REQUEST message can be "piggybacked" onto the PRIVILEGE message, eliminating one message. When received, a "piggybacked" message is processed as if the two separate messages had arrived. Only nonleaf nodes can send a piggyback REQUEST as leaf nodes will not require the return of the privilege at the time it is given away.

Piggybacking is most effective when there is considerable demand for the mutual exclusion. The number of REQUEST messages eliminated by piggybacking is almost zero in a low demand system (since it is extremely unlikely that a node will require that the privilege be returned to it), but rises to

$$\frac{\sum \text{valency of nonleaf nodes}}{\sum \text{valency of all nodes}}$$

when the system is saturated. Thus for the tree shown in Figure 7, approximately $\frac{2}{3}$ of the REQUEST messages could be piggybacked when the system is saturated. For large complete trees of valency $K$, piggybacking eliminates approximately $K/2(K-1)$ of the REQUEST messages when the system is saturated. For a straight line topology (a complete tree of valency 2), almost all REQUEST messages will be piggybacked in a saturated system.

Apart from reducing the number of messages required, piggybacking has no other impact on the algorithm or its performance. We know of no disadvantage to its use, other than its ineffectiveness in systems with light demand.

### 6.2 The Greedy Strategy

Our second variation, the Greedy Strategy, weakens the requirement that REQUEST_Qs are held as a FIFO list. When a node X wishes to enter the critical section, it places itself at the head of $\text{REQUEST\_Q}_X$ rather than at the tail. When X receives the PRIVILEGE message, X is able to enter its critical section straightaway, rather than wait until the earlier requesting neighbors have had their turn.

Like the Piggyback Strategy, the Greedy Strategy has little effect in a system with light demand for the privilege, as a requesting node would probably be the only node in its REQUEST_Q anyway.

Under heavier demand for the privilege, the Greedy Strategy reduces the *average* delay between a node requiring the privilege and receiving it. This, in turn, increases the number of entries to the critical section that can be achieved in a given time. However these improvements are at the expense of the leaf nodes.

In the standard algorithm, the delay times for all nodes, leaf or nonleaf, are almost equal, and the number of critical section entries achieved by each node are almost equal. Using the Greedy Strategy, leaf nodes may wait many times longer than the nonleaf nodes to receive the privilege, and consequently obtain it on fewer occasions than nonleaf nodes. For example, when the system shown

in Figure 7 is saturated, each nonleaf will obtain the privilege three times more often than a leaf node; and, correspondingly, the delay time for a leaf node will be three times longer than for a nonleaf node.

The Greedy Strategy preserves the starvation-free nature of the algorithm (no node is overlooked indefinitely) but is not as fair as the standard algorithm. In fact, it directly trades fairness for efficiency.

That the Greedy Strategy is starvation-free is not immediately obvious. However all local requests to enter the critical section are serialized, and hence whenever a node releases the mutual exclusion, it will pass the privilege to the node X at the head of its REQUEST_Q (if any). This node X cannot be "self" but must be a neighboring node. A neighboring node at the head of REQUEST_Q can be overtaken by "self" at most once. Therefore each node in the REQUEST_Q must eventually reach the head of the REQUEST_Q, and ultimately receive the privilege.

## 7. INITIALIZATION

To initialize the algorithm, one node is chosen as the initial privileged node. This node sends an INITIALIZE message to each of its neighbors. When node A receives an INITIALIZE message from a neighboring node B, A assigns $HOLDER_A$ to B, and then sends an INITIALIZE message to all of its neighbors (other than B). Once a node has received the INITIALIZE message, it may request the privilege (even though other parts of the tree may not yet be initialized).

The initialization of the other variables is the same for all nodes, and is shown below:

```
REQUEST_Q    := empty
USING        := false
ASKED        := false
```

## 8. NODE FAILURE

In the event of a node failing and losing the information required for this algorithm, it is possible to reconstruct that information from the node's neighbors when the node restarts.

When a node X restarts, it commences a recovery phase. The first action of the recovery phase is to delay for a period sufficiently long to ensure that all messages sent by node X *before* it failed have been received. Node X then sends RESTART messages to each of its neighbors, and awaits the ADVISE messages that each neighbor will send in reply.

During the recovery phase, node X may receive REQUEST and PRIVILEGE messages from neighboring nodes. If X receives a REQUEST message from node Y, then Y is placed in $REQUEST\_Q_X$. If X receives a PRIVILEGE message, then $HOLDER_X$ becomes "self." If node X wishes to enter the critical section during the recovery phase, then "self" is placed in $REQUEST\_Q_X$. All of these actions are the normal responses to these events.

However the procedures ASSIGN_PRIVILEGE and MAKE_REQUEST are not called during the recovery phase. The recovery phase involves information

gathering and reconstruction of local data. Until that task is complete, node X must not attempt to make decisions based on incomplete information. After the recovery phase is completed, ASSIGN_PRIVILEGE and MAKE_REQUEST are then called to allow node X to recommence its participation in the algorithm.

### 8.1 The ADVISE Message

When a neighboring node Y receives X's RESTART message, Y must reply send an ADVISE message informing X of the state of the X − Y relationship as Y sees it. Below are the four possible states (corresponding to each of the four messages in the logical pattern of X − Y communication), together with the information that X can deduce from this relationship.

(1) $HOLDER_Y = X$ and $ASKED_Y$ = false
Hence X *may* be the privileged node, and Y is not an element of $REQUEST\_Q_X$.

(2) $HOLDER_Y = X$ and $ASKED_Y$ = true
Again X *may* be the privileged node, and Y is an element of $REQUEST\_Q_X$.

(3) $HOLDER_Y \neq X$ and not $X \in REQUEST\_Q_Y$
Hence X is not the privileged node (it is node Y or beyond), and $ASKED_X$ must be false.

(4) $HOLDER_Y \neq X$ and $X \in REQUEST\_Q_Y$
Again X is not the privileged node, and it has requested the privilege so $ASKED_X$ must be true.

### 8.2 Determining $HOLDER_X$

If $HOLDER_Y = X$ for all of X's neighbors Y, then X is the privileged node, and $HOLDER_X$ = self. If X is not the privileged node, then $HOLDER_Y = X$ for all but one of X's neighbors Y. The dissenting neighbor Z is therefore closer to the privileged node than X, and so $HOLDER_X = Z$. It is impossible for $HOLDER_Y \neq X$ for more than one neighbor (see Section 4.1).

### 8.3 Determining $ASKED_X$

If X is the privileged node, then $ASKED_X$ is false. If X is not the privileged node and $HOLDER_X$ is Z, say, then $ASKED_X$ is true if $X \in REQUEST\_Q_Z$.

### 8.4 Reconstructing $REQUEST\_Q_X$

The entries in X's REQUEST_Q can be determined from the ADVISE messages. If $HOLDER_Y = X$ and $ASKED_Y$ = true, then Y should be an entry in $REQUEST\_Q_X$. The *order* of the elements of $REQUEST\_Q_X$ will be lost, but this is not essential for the continued operation of the algorithm.

If desired, each node could remember when it made its last REQUEST, and convey this information in the ADVISE message. Using these time-stamps, the order of X's reconstructed REQUEST_Q would more closely resemble the original, subject to the extent to which such time-stamps can be compared [1].

In theory, it is possible that a neighboring node Y could be starved by the loss of order in $REQUEST\_Q_X$. If the reconstruction of $REQUEST\_Q_X$ causes Y's request to be overtaken by another request, then (in the terminology of Section 4.3) the effect of the recovery is to increase the value of the vector. Consequently if each failure of X happens before Y receives the privilege, and the subsequent reconstruction of $REQUEST\_Q_X$ allows Y's REQUEST to be overtaken, then

node Y will be starved. However, such a situation is extremely improbable in real life.

Alternatively, the ADVISE message could contain the number of times that the neighbor Y had received the PRIVILEGE message from X. The reconstruction of REQUEST_$Q_X$ would be ordered by ascending number of PRIVILEGE messages previously received. Although this ordering does not attempt to reconstruct the original order of REQUEST_$Q_X$, it will prevent starvation, in the improbable scenario previously described.

### 8.5 Reassigning $USING_X$

$USING_X$ can be set to false.

### 8.6 Assistance from Neighboring Nodes

Apart from replying to the RESTART message with an ADVISE message, neighboring nodes do not further involve themselves in node X's failure and subsequent recovery. They are not required to cease message transmission to X, or delay their actions in any way. While this eliminates the need for complex resynchronization among X and its neighbors, it allows REQUEST and PRIVILEGE messages to overtake ADVISE messages.

If node X receives a PRIVILEGE message from neighboring node Y during recovery (whether sent before or after Y's ADVISE message), then X is the privileged node. If the PRIVILEGE message was sent before the ADVISE message, then the ADVISE message will correctly state that $HOLDER_Y = X$. If the PRIVILEGE message was sent *after* the ADVISE message, then the ADVISE message will contain the outdated information that $HOLDER_Y \neq X$. However the possession of the token is sufficient proof that X holds the privilege. Contradictory ADVISE messages must be outdated, and can be ignored.

If node X receives a REQUEST message from Y during recovery (whether sent before or after Y's ADVISE message), then Y wants the privilege from X. Therefore there is the possibility that Y will be placed in X's REQUEST_Q twice, once in response to the ADVISE message and once in response to the REQUEST message. To avoid this, one could check for such duplication during the recovery phase (or at the end of the recovery phase).

However, the presence of such duplicates does not endanger the correctness of the algorithm, provided that the REQUEST_Q is physically large enough (or extensible) to accommodate duplicates (at most two occurrences of each neighbor or self). If REQUEST_$Q_X$ contains two entries for node Y, then at some future time Y will receive the privilege when it is not expecting it. (This will not occur the next time Y receives the privilege, but on some later occasion.) Although this does not conform to the logical pattern of message traffic between X and Y, the algorithm is nonetheless insensitive to it.

Upon receiving the unexpected PRIVILEGE message, node Y will behave as if it *had* requested the privilege and then immediately finished using it. Since Y was not expecting the PRIVILEGE, REQUEST_$Q_Y$ must be empty. Node Y will then simply hold the privilege without using it. However, the collective HOLDER variables will be "pointing" towards node Y, and any REQUESTs for the privilege will propagate to node Y, which will respond by sending the PRIVILEGE. Despite

a few unnecessary message transmissions, the algorithm automatically corrects itself after the delivery of the PRIVILEGE to a disinterested node.

### 8.7 Failure During Recovery

It is possible for the node X to fail during the recovery phase of an earlier failure. If this occurs, then the second recovery phase may receive ASSIST messages related to the first recovery phase. Such messages may contain outdated information and should be ignored. There are a number of ways of identifying such messages.

If the time of the delay at the start of the recovery phase is carefully chosen, it should be possible to ensure that all outdated ASSIST messages will have arrived before the second set of RESTART messages is sent. Thus, only relevant ASSIST messages will be received during the second recovery phase.

A second method is to have some unique identifier on each RESTART message, which must be quoted in the replying ASSIST message. ASSIST messages with an incorrect identifier can be discarded. Possible sources for these unique identifiers are the real-time clock, nonvolatile storage, or an external agent (e.g., a server or a user).

### 8.8 Extent of Survivable Failures

In the preceding description of failure recovery, it is assumed that other nodes are operating normally, that is, are neither failed nor recovering. Provided that failure occurs infrequently, this is a reasonable and realistic assumption.

The recovery mechanism will work even in the event of concurrent node failures, provided that the failures do not occur in adjacent nodes.

When two or more adjacent nodes fail, it may be possible to determine the collective status of these nodes from the ASSIST messages received from their surviving neighbors. Once this macroscopic status has been established, the recovering nodes can negotiate their individual (or microscopic) status accordingly. However, further research is needed to determine the ability of this group recovery mechanism to survive the subsequent failure of either recovering or neighboring nodes.

## 9. CONCLUSIONS

We have presented a deadlock- and starvation-free algorithm for distributed mutual exclusion in which the average number of messages required per critical section is $O(\log N)$, reducing to a constant of four messages as the demand for the mutual exclusion increases.

Each node need know only of its neighbors in the tree, thus restricting the amount of data needed to be held by each node.

Message overtaking does not present a problem, and so sequence numbers are not required. During normal operation (i.e., not failure recovery), each message requires only sufficient bits to indicate

—the type of the message (e.g., REQUEST, PRIVILEGE, INITIALIZE)
—the identity of the neighbor that sent the message (if this information is not supplied by the underlying message-passing system).

In many systems, it will be possible to multiplex this small number of bits onto other outgoing messages to further reduce the bandwidth required.

Our algorithm is also resilient to a variety of unexpected events, and can recover from localized failures.

ACKNOWLEDGMENTS

Particular thanks are due to David Horton for his assistance in the analysis of simulation results, and in the preparation of this manuscript. We also wish to thank Professor Andrew Lister, from the University of Queensland, and the anonymous referees for their careful reading and helpful comments.

REFERENCES

1. LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558–565.
2. MAEKAWA, M. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst. 3*, 2 (May 1985), 145–159.
3. RICART, G., AND AGRAWALA, A. K. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM 24*, 1 (Jan. 1981), 9–17.
4. SUZUKI, I., AND KASAMI, T. A distributed mutual exclusion algorithm. *ACM Trans. Comput. Syst. 3*, 4 (Nov. 1985), 344–49.

Received October 1987; revised August 1988; accepted September 1988