A DAG-BASED ALGORITHM FOR DISTRIBUTED MUTUAL EXCLUSION

by

Mitchell L. Neilsen

A THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1989

Approved by:

Major Professor

LD
2668
.T4
CMSC
1989
N45
c. 2

A11208 617933

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 HISTORY 3

2.1 Lamport's Algorithm 3
2.2 Ricart and Agrawala's Algorithm 5
2.3 Carvalho and Roucairol's Algorithm 5
2.4 Suzuki and Kasami's Algorithm 6
2.5 Singhal's Algorithm 7
2.6 Maekawa's Algorithm 8
2.7 Raymond's Algorithm 10

CHAPTER 3 OVERVIEW 13

3.1 Message Types 14
3.2 Variable Types 14
3.3 Example 16

CHAPTER 4 ALGORITHM 19

4.1 Algorithm 19
4.2 Complete Example 22

CHAPTER 5 PROOFS 27

5.1 Mutual Exclusion 27
5.2 Deadlock and Starvation Freedom 27

CHAPTER 6 PERFORMANCE ANALYSIS 34

6.1 Upper Bound 35
6.2 Average Bound 35
6.3 Synchronization Delay 36
6.4 Storage Overhead 37

CHAPTER 7 CONCLUSION 38

BIBLIOGRAPHY 39

LIST OF FIGURES

Figure 1. Directed Acyclic Graph Structure 13

Figure 2. Example 17

Figure 3. Algorithm 20

Figure 4. State Transition Graph for Node I 21

Figure 5. Initialization Procedure 22

Figure 6. Complete Example 24

Figure 7. Case (1), (2), and (2c) in Lemma 2 30

Figure 8. Centralized Topology 34

ACKNOWLEDGEMENTS

I am deeply grateful for having the opportunity to engage in research with my Major Professor, Dr. Masaaki Mizuno. I was inspired by his willingness and patience to work through every detail in our research. Working together, we were able to produce some very nice research results.

I also want to thank my wife, Rebecca, and daughters, Anne and Beth, for their patience and support during the development of this thesis.

Chapter 1

INTRODUCTION

Many distributed mutual exclusion algorithms have been proposed [1, 2, 3, 4, 5, 6, 7, 8, 9]. These algorithms can be classified into two groups [8]. The algorithms in the first group are token based [4, 5, 8, 9]. The possession of a system-wide unique token gives a node the right to enter its critical section. The algorithms in the second group are assertion based [1, 2, 3, 6, 7]. At any given time, the assertion can only be true at one node; a node enters its critical section only after the assertion becomes true.

This paper presents a token based algorithm which further improves on other algorithms. The algorithm assumes a fully connected physical network and a directed acyclic graph (dag) structured logical network. Using the best logical topology, the maximum number of messages required is three. This is the same performance exhibited by centralized schemes. Furthermore, the synchronization delay is minimal, i.e., one message. A node or a token does not need to maintain a queue of outstanding requests for mutual exclusion. Instead, the queue is maintained implicitly in a distributed manner and may be deduced by observing the states of the nodes. Our algorithm requires very simple data structures; each node maintains a few simple variables, and the token carries no data structure. This

is significantly less overhead compared with other distributed mutual exclusion algorithms, where they maintain a queue or an array structure, either in every node or within the token.

The history of distributed mutual exclusion algorithms and the quest for optimization in those algorithms is presented in Chapter 2. An informal description of the algorithm is presented in Chapter 3. The detailed algorithm is presented in Chapter 4, followed by a more complete example. Chapter 5 presents the proofs of the correctness with respect to guaranteed mutual exclusion, deadlock freedom and starvation freedom. Chapter 6 analyzes the performance of the algorithm.

Chapter 2

HISTORY

We assume that the system consists of N nodes, which are uniquely numbered from 1 to N. At any given time, each node can have at most one outstanding request to enter its critical section. Physically, the nodes are fully connected by a reliable network. Messages sent by the same node are not allowed to overtake each other while in transit.

2.1 Lamport's Algorithm

Lamport proposed one of the first distributed mutual exclusion algorithms [2]. The algorithm has two major components: the total ordering of messages and the distribution of a queue over all nodes.

The total ordering is done by using *logical clocks* to generate sequence numbers at each node. Between any two requests, the *logical clock* increments a node's sequence number. All messages sent from node I are of the form (msg, c_I, I), where msg is the message and c_I is the sequence number generated at node I. On receipt of a message, a node increments its own sequence number to be larger than the sequence number in the message. Hence, the receipt of a message always (logically) comes after when it was sent. Two messages with the same sequence

number are ordered based on the unique integer values assigned to each node. A total ordering is defined on messages by saying (msg, c_I, I) comes before (msg, c_J, J) if $c_I < c_J$ or ($c_I = c_J$ and $I < J$).

To distribute the queue, each node maintains a copy of the queue. The queue generally holds the latest message received from each node. Messages are totally ordered in the queue by sequence number as described above.

In the algorithm, there are three different types of messages: REQUEST, ACKNOWLEDGE, and RELEASE. When a node wants to enter its critical section, it sends a REQUEST message to all other nodes and inserts the request in its own queue. Upon receipt of a REQUEST message, a node inserts the request in its queue and sends an ACKNOWLEDGE message back to the node originating the request. A node can enter its critical section when its request has the lowest sequence number (highest priority) of any request in its queue and has received messages, with higher sequence numbers, from all other nodes (this is the assertion of the algorithm). When a node leaves its critical section, it sends a RELEASE message to all other nodes. Upon receipt of a RELEASE message, a node replaces the corresponding request from its queue with the RELEASE message. When a node receives an ACKNOWLEDGE message, the message is put in its queue only if a REQUEST message from the node sending the message is not already in its queue. Hence, a node does not have to send an ACKNOWLEDGE message if it has sent a REQUEST message and has not received the corresponding RELEASE message because the ACKNOWLEDGE message will be discarded. Since $(N-1)$ REQUEST messages, at most $(N-1)$ ACKNOWLEDGE messages, and $(N-1)$ RELEASE messages are required, per critical section entry, a total of at most $3*(N-1)$ messages are required.

2.2 Ricart and Agrawala's Algorithm

By combining the ACKNOWLEDGE and RELEASE messages into a single REPLY message, Ricart and Agrawala reduced the number of messages required per critical section entry [6]. If a node wants to enter its critical section, it generates a sequence number and sends a REQUEST message (including the sequence number) to all other nodes. Upon receipt of a REQUEST message, a determination is made immediately to determine whether to send or defer a REPLY message. If the node receiving the request wants to enter its critical section and has already requested with a lower sequence number (higher priority), then the REPLY message is deferred until the node receiving the request leaves its critical section. Otherwise, a REPLY message is sent back to the node originating the request immediately. When a node has received a REPLY message back from all other nodes, it may enter its critical section. Upon leaving its critical section, a node sends a REPLY message to all nodes which it has deferred. Since $(N-1)$ REQUEST messages and $(N-1)$ REPLY messages are required, per critical section entry, a total of $2*(N-1)$ messages are required.

2.3 Carvalho and Roucairol's Algorithm

By using a different definition of symmetry, Carvalho and Roucairol were able to reduce the number of messages to be between 0 and $2*(N-1)$ [1]. Ricart and Agrawala's definition of symmetry requires that any node wanting to enter its critical section must inform every other node of its intention. However, a node, which has received a REPLY message back from all other nodes, may enter its critical section repeatedly until it receives a REQUEST message from another node. Also, a node, wanting to enter its critical section again, needs to send

REQUEST messages only to the nodes from which it has received a request. In this way, the number of messages required per critical section entry may be reduced. The upper and lower bounds on the number of messages required may be obtained.

2.4 Suzuki and Kasami's Algorithm

In response to Carvalho and Roucairol's algorithm, Ricart and Agrawala proposed a token based algorithm [5] which is essentially the same as Suzuki and Kasami's approach [9]. In particular, we present Ricart and Agrawala's approach. Initially, one node holds an explicit token. When a node wants to enter its critical section, it checks to see if it is holding the token. If it is holding the token, it may enter its critical section immediately. Otherwise, a REQUEST message (with a sequence number) is sent to all other nodes. Each node maintains a queue of outstanding requests. Upon receipt of a REQUEST message, a node puts the request in its queue sorted by sequence number. When the node holding the token no longer wants to use the token, it looks for the request with the smallest sequence number in its queue and sends a PRIVILEGE message to pass the token. Every time a request is satisfied, the sequence number is recorded on the token. This makes it easy to determine if a request has been satisfied yet. A request which has not yet been satisfied must have a sequence number larger than the one recorded on the token. If a node is not holding the token, the algorithm requires $(N-1)$ REQUEST messages and one PRIVILEGE message. Hence, either 0 or N messages are required per critical section entry.

2.5 Singhal's Algorithm

Based on Suzuki and Kasami's algorithm, Singhal proposed a heuristically-aided algorithm that uses state information to more accurately guess the location of the token [8]. Each node maintains state information on all other nodes. When a node wants to enter its critical section, it uses a heuristic to guess which nodes are probably holding the token, based on its state information. Then, a REQUEST message is sent only to those nodes. A node can be in one of four different states:

(R) A node is requesting to enter its critical section.

(E) A node is executing in its critical section.

(H) A node is holding the token and not requesting.

(N) A node is not requesting and not holding the token.

Each node maintains *state vectors* to store information about the state of each node. In particular, information on the latest known state and highest known sequence number is maintained. The REQUEST and PRIVILEGE messages used for mutual exclusion are also used to pass state information.

Several possible heuristics exist. The one used in Singhal's algorithm is to send REQUEST messages to all nodes in state (R); i.e. nodes which have recently sent a request for the token.

As demand for critical section entry increases, the number of messages required, per critical section entry, approaches N. Hence the upper bound is the same as the number of messages required in Suzuki and Kasami's algorithm.

2.6 Maekawa's Algorithm

Maekawa proposed another assertion based algorithm in which the number of messages required is approximately $c * \sqrt{N}$, where c is between 3 and 7 [3][1]. The basic idea behind Maekawa's algorithm is that it is not necessary to obtain permission from every other node. For each node I, it is necessary to predefine a committee which includes node I, say S_I. Any two committees must have a nonempty intersection; i.e. a node in common. If a node receives permission to enter its critical section from all of its members, no other node may receive permission from all of its committee members. The problem of finding a set of committees is equivalent to finding a finite projective plane. If each committee has K members, then K is minimized when the number of nodes N is given by $N = K * (K - 1) + 1$. Hence, $K \approx \sqrt{N}$.

Every node maintains a queue of outstanding requests. When a node wants to enter its critical section, it sends a REQUEST message (with a sequence number) to every node in its committee and pretends to have received the REQUEST message itself.

Upon receipt of the REQUEST message, the receiving node puts the request on its queue, ordered by sequence number. A node returns a LOCKED message to the requesting node and marks itself as locked, if it is not locked for another request. The LOCKED message corresponds the REPLY message in Ricart and Agrawala's algorithm. If the node is locked for another request (has sent a LOCKED message) and the sequence number of the request currently locked is smaller, then a FAIL

[1] In [3], Maekawa claimed that c is between 3 and 5. However, in [7], Sanders pointed out that the algorithm may deadlock and that not all the required messages were counted. As suggested in [7], the algorithm can easily be modified to be deadlock free, and with this modification, c is between 3 and 7.

message is returned to the requesting node. Otherwise, an INQUIRE message is sent to the node originating the current locked request. In a modification suggested by Sanders in [7] to prevent deadlock, a FAIL message is sent to any node with a request in its queue with a larger sequence number, if one has not already been sent.

Once a node receives a LOCKED message from each node in its committee, it may enter its critical section. When a node leaves its critical section, it sends a RELEASE message to each committee member. When a node receives a RELEASE message, it removes the current locking request and locks itself for the request with the lowest sequence number (highest priority) in its queue. If its queue is empty, the node becomes unlocked.

If a node receives an INQUIRE message and will not be able to enter its critical section, it sends a RELINQUISH message back to the inquiring node. A node will not be able to enter its critical section if it has received a FAIL message or has already sent a RELINQUISH message and has not received a new LOCKED message. If a node can't immediately determine if it will be able to enter its critical section (i.e, has only received some LOCKED messages), it simply defers its response to the inquiring node until it can decide. When a RELINQUISH message is received, the node marks itself as unlocked and the relinquished request is returned to the request queue. The node proceeds as if a RELEASE message had been received.

In the best case, approximately $3 * \sqrt{N}$ messages are required per critical section entry. Three sets of messages are sent: REQUEST, LOCKED, and RELEASE.

In the worst case, seven sets of messages are exchanged: REQUEST, LOCKED

(this will be relinquished later), INQUIRE, RELINQUISH, FAIL, LOCKED, and RELEASE. Thus, approximately $7 * \sqrt{N}$ messages are required per critical section entry.

2.7 Raymond's Algorithm

Recently, Raymond proposed a token based approach which assumes that the network topology is an unrooted tree structure [4]. The number of messages required by the algorithm depends on the topology of the tree. For the radiating star topology, the number of messages is between four and $O(logN)$[2]. When a node wants to enter its critical section and is not holding the token, it sends a REQUEST message to the neighboring node on the path, in the logical structure, to the node holding the token. The neighboring node forwards the request on the path to the node holding the token. When the node holding the token leaves its critical section, it sends the token back on the same path from which the request came. As the token travels, each node forwarding the token sets its $NEXT$ variable to point to the neighboring node on the path to the token. Hence, the directed tree structure is maintained and the $NEXT$ variables always indicate the direction in which the token is located.

Two types of messages are used: REQUEST and PRIVILEGE. Each node maintains several variables:

- $NEXT$ indicates the relative location of the token.
- $USING$ indicates if a node is in its critical section.
- $ASKED$ indicates if a REQUEST message has already been sent.

[2]This is, however, not the optimal topology, as we will show.

When a node wants to enter its critical section, it puts itself on its own queue and checks to see if it is holding the token. If it is holding the token, it can enter its critical section. Otherwise, it must send a REQUEST message to the neighboring node indicated by its $NEXT$ variable.

Upon receipt of a REQUEST message from a neighboring node X, the neighboring node's request is put on the local request queue. If the node is holding the token and node X's request is at the top of the queue, a PRIVILEGE message is sent to node X to pass the token. Otherwise, a REQUEST message is forwarded onto the node indicated by the $NEXT$ variable. Since a single node may receive several requests, the $ASKED$ variable is used to ensure that only one outstanding REQUEST message is forwarded on behalf of the request at the top of the request queue. This reduces the number of messages required. Note, the only time $NEXT = \textit{self}$ is when a node is holding the token.

If a node receives a PRIVILEGE message, it sets its $NEXT$ variable to point to itself and does the following. If the node's request is at the top of the request queue, the request is dequeued and the node sets $USING$ to true and enters its critical section. When a node leaves its critical section, it sets $USING$ to false. Then, if the request queue is nonempty, the node holding the token sends a PRIVILEGE message to the node in the request at the top of its queue, dequeues the request, sets its $NEXT$ variable to point to the neighboring node making the request, and sets its $ASKED$ variable to false.

Note, this is the first algorithm discussed which does not use sequence numbers to order requests. Requests are ordered based on the order in which they are received at an adjacent node.

The number of messages required, per critical section entry, is between 0 and

$2 * D$, where D is the *diameter* of the logical structure. The upper bound is attained when the node originating the request and the node holding the token are at opposite ends of the longest path in the logical structure. In this case, D REQUEST messages and D PRIVILEGE messages are required. The synchronization delay (the number of sequential messages required between one node leaving its critical section and another (waiting) node entering its critical section) is at most D.

Chapter 3

OVERVIEW

Physically, these nodes are fully connected by a reliable network. Logically, they are arranged in a dag structure with only one sink node[1]. The degree of each node is at most one. We further impose that the structure of the graph is acyclic even without considering the directions of the edges. An example is shown in Figure 1.

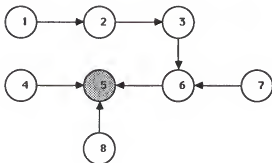


Figure 1. Directed Acyclic Graph Structure

[1]This assertion is temporarily violated when REQUEST messages (introduced below) are in transition, as explained later in this section.

3.1 Message Types

Two types of messages, REQUEST and PRIVILEGE, are passed between nodes. When a node requests to enter its critical section, it initiates a REQUEST message. A PRIVILEGE message represents the token; when a node receives a PRIVILEGE message, it can enter its critical section.

3.2 Variable Types

Each node maintains three simple variables: a boolean variable $HOLDING$ and integer variables $NEXT$ and $FOLLOW$. A logical dag structure indicates the path along which a REQUEST message travels and is imposed by the $NEXT$ variables in the nodes. When a node initiates or receives a REQUEST message, the node forwards it to the neighboring node pointed at by its $NEXT$ variable (unless the node is a sink, in which case its $NEXT$ variable is 0; this case will be explained below).

The $FOLLOW$ variable indicates the node which will be granted mutual exclusion after this node. When a node exits its critical section, it sends a PRIVILEGE message to the node indicated by its $FOLLOW$ variable and clears the variable, unless its $FOLLOW$ variable is 0. If its $FOLLOW$ variable is 0, the node continues to hold the token. This case is explained below. By following the $FOLLOW$ variables in the system, the implicit waiting queue of the system can be deduced.

Semantically, a sink node in the system is (1) the last node in the implicit waiting queue (i.e., its $FOLLOW$ variable is 0), and (2) the last node in the path along which a request travels (i.e., its $NEXT$ variable is 0). When a sink node receives a REQUEST message, it *enqueues* the request into the implicit waiting queue and becomes a non-sink. The node initiating the request becomes the new

sink since it is now the last node in the queue. The path must be changed in the direction of the new sink. This procedure is done by the cooperation of the nodes along the path in a distributed manner as follows:

- When a node initiates a new REQUEST message, it forwards the message to its neighboring node indicated by its $NEXT$ variable and sets its $NEXT$ variable to 0 to become a new sink. It remains a sink until it receives a subsequent request.

- When an intermediate (non-sink) node receives a REQUEST message from a node X, it passes the message to the neighboring node indicated by its $NEXT$ variable. The node then sets its $NEXT$ variable to X. Thus, if it receives another request later, it forwards the request in the direction of the new sink.

- When a sink node receives a REQUEST message from a node X, it sets its $FOLLOW$ variable to the identifier of the node initiating the request. This corresponds to an *enqueue* operation. The node also sets its $NEXT$ variable to X to enter the path in the direction of the new sink. Note that if a sink node holds the token but is not in its critical section (this state is indicated by a boolean variable $HOLDING$) when it receives a request, it immediately forwards the token to the node initiating the request.

Because of message delay, there may be more than one sink node in the system while some requests are in transit. Assume that node X and node Y initiate requests at about the same time. There may be at most three sink nodes while the requests are in transit: node X, node Y and the current sink node. The current sink becomes a non-sink when it receives one of the requests (assume it receives

a request from node X). Node X becomes a non-sink when it receives the request from node Y [2]. Eventually, node Y becomes the only sink node in the system.

The system is initialized so that one node possesses the token[3], and in all others, the $NEXT$ variable is set to point to the neighbor which is on the path to the node holding the token. A simple procedure for initializing the system is shown later in Figure 5.

3.3 Example

Consider the example given in Figure 2. Node 5 holds the token initially. Let the directed edges indicate the direction in which the $NEXT$ variables are pointing. The initial configuration is shown in Figure 2a. Suppose node 5 wants to enter its critical section. Since the node holds the token, it can enter immediately. Now, suppose node 3 wants to enter its critical section. It sends a REQUEST message to node 4 and sets its $NEXT$ variable to 0 to become a new sink (refer to Figure 2b). Node 4 receives the request and sets its $NEXT$ variable to point to node 3 and forwards the REQUEST message to node 5, on behalf of node 3 (refer to Figure 2c). Node 5 receives the REQUEST message. Since node 5 is a sink node, it sets its $FOLLOW$ variable to point to node 3 and sets its $NEXT$ variable to point to node 4 to become a non-sink. When node 5 leaves its critical section, it sends a PRIVILEGE message to the node indicated by its $FOLLOW$ variable, i.e., node 3 (refer to Figure 2d). Finally, node 3 receives the PRIVILEGE message and enters its critical section (refer to Figure 2e).

[2]It is proved in section 4 that node X is guaranteed to receives a request from node Y if the current sink receives a request from node X and there is no other requesting node in the system.

[3]This is the sink node, and its $NEXT$ variable points to 0.

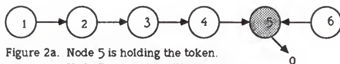


Figure 2a. Node 5 is holding the token.
Node 5 enters its critical section.

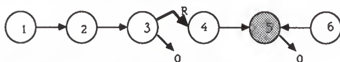


Figure 2b. Node 3 wants to enter its critical section.
Node 3 sends a REQUEST message to
node 4 and sets NEXT_3 = 0.

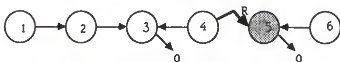


Figure 2c. Node 4 receives the request from node 3,
forwards a REQUEST message to node 5,
and sets NEXT_4 =3.

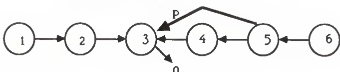


Figure 2d. Node 5 receives the request from node 4,
sets FOLLOW_5 = 3, and NEXT_5 = 4.
When node 5 leaves its crititcal section,
it sends a PRIVILEGE message to node 3.

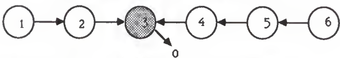


Figure 2e. Node 3 receives the PRIVILEGE message
and enters its critical section.

Note: The shaded regions indicate the token holder.

Figure 2. Simple Example

Our algorithm is not *fully distributed* as defined by Ricart and Agrawala [6]. Instead, the algorithm, as in [3] and [4], is based on a *surrogate mechanism*, in which a node asks other nodes to act on its behalf. Also, as in other token based algorithms, it is not *symmetric* since a node is allowed to hold the token while not actually using the resource. These issues are, however, a matter of definitions as stated in [9]. Because of these characteristics, the number of messages required per critical section entry is reduced significantly in our algorithm.

Chapter 4

ALGORITHM

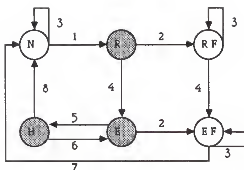
4.1 Algorithm

The complete algorithm is shown in Figure 3. The diagram in Figure 4 shows the state transition graph of each node. The initialization procedure is shown in Figure 5. There are two procedures at each node: P1 and P2. P1 is responsible for making requests for entry into the critical section, and P2 is responsible for processing request messages received from other nodes.

We assume that the REQUEST message is of the form REQUEST(X, Y), where X denotes the adjacent node from which the request came and Y denotes the node where the request originated. Each node executes procedures P1 and P2 in local mutual exclusion. The only exception is that a node does not have to execute in mutual exclusion while waiting for a PRIVILEGE message to arrive or while in its critical section.

```
const
    I = node identifier
var
    HOLDING         : boolean;
    NEXT, FOLLOW    : integer;

procedure P1; (* node I wants to enter its critical section *)
    begin
        if (not HOLDING) then
            begin
                send REQUEST(I,I) to NEXT;
                NEXT := 0;
                wait until PRIVILEGE message is received;
            end;
        HOLDING := false;

                critical section

        if (FOLLOW ≠ 0) then
            begin
                send PRIVILEGE message to FOLLOW;
                FOLLOW := 0;
            end;
        else HOLDING := true;
    end;

procedure P2; (* node I received REQUEST(X,Y) from X *)
    begin
        if (NEXT = 0) then (* node I is a sink*)
            begin
                if HOLDING then
                    begin
                        send PRIVILEGE message to Y;
                        HOLDING := false;
                    end;
                else FOLLOW := Y;
            end;
        else send REQUEST(I,Y) to NEXT;
        NEXT := X;
    end;
```

Figure 3. Algorithm



Note: The shaded regions indicate a sink state (NEXT_I = 0).

STATES:	
N	Node I is not requesting and not holding the token.
R	Node I is requesting the token, but has not received a subsequent request for the token.
R F	Node I is requesting the token, and has received a subsequent request for the token.
E	Node I is executing in its critical section and has not received a subsequent request for the token.
E F	Node I is executing in its critical section and has received a subsequent request for the token.
H	Node I is holding the token and has received no requests for the token.

TRANSITIONS:	
1	Node I sends a REQUEST(I,I) message to NEXT_I. Node I sets NEXT_I = 0.
2	Node I receives a REQUEST(X,Y) message from node X. Node I sets NEXT_I = X and FOLLOW_I = Y.
3	Node I receives a REQUEST(X,Y) message from node X. Node I sends a REQUEST(I,Y) message to NEXT_I, and sets NEXT_I = X.
4	Node I receives a PRIVILEGE message. Node I can enter its critical section.
5	Node I leaves its critical section. Node I sets HOLDING = true.
6	Node I enters its critical section. Node I sets HOLDING = false.
7	Node I leaves its critical section. Node I sends a PRIVILEGE message to FOLLOW_I, and sets FOLLOW_I = 0.
8	Node I receives a REQUEST(X,Y) message. Node I sets NEXT_I = X, HOLDING = false, and sends a PRIVILEGE message to node Y.

Figure 4. State Transition Graph for Node I

```
procedure INIT; (* node I wants to initialize *)
  begin
    if (holding the token) then
      begin
        HOLDING := true;
        NEXT := 0; (* the node is a sink *)
        FOLLOW := 0;
        send INITIALIZE(I) message to all neighboring nodes;
      end;
    else
      begin
        wait for INITIALIZE(J) message to arrive from node J;
        HOLDING := false;
        NEXT := J;
        FOLLOW := 0;
        send INITIALIZE(I) message to all neighboring nodes,
          except J;
      end;
  end
```

Figure 5. Initialization Procedure

4.2 Complete Example

We now give a complete example in Figure 6. A subscript is used to denote the value of a variable at node I; i.e., $HOLDING_I$, $NEXT_I$, and $FOLLOW_I$ denote the values of $HOLDING$, $NEXT$ and $FOLLOW$ at node I.

1. Initially node 3 is holding the token and is not in its critical section. All nodes have been initialized as shown in Figure 6a.

2. Node 3 wants to enter its critical section. Node 3 sets $HOLDING_3$ = false and enters its critical section.

3. Node 2 wants to enter its critical section, so node 2 sends a REQUEST(2,2) message to node 3 and sets $NEXT_2 = 0$ to become a sink (refer to Figure 6b).

4. Node 3 receives a REQUEST(2,2) message from node 2. Since node 3 is a sink and in its critical section, it saves the request by setting $FOLLOW_3 = 2$. Node 3 then sets $NEXT_3 = 2$ and becomes a non-sink (refer to Figure 6c).

5. Node 1 wants to enter its critical section, so node 1 sends a REQUEST(1,1) message to node 2 and sets $NEXT_1 = 0$ to become a sink.

6. Node 5 wants to enter its critical section, so node 5 sends a REQUEST(5,5) message to node 2 and sets $NEXT_5 = 0$ to become a sink (refer to Figure 6d).

7. Node 2 receives a REQUEST(1,1) message from node 1. Since node 2 is a sink, it saves the request by setting $FOLLOW_2 = 1$. Node 2 also sets $NEXT_2 = 1$ and becomes a non-sink (refer to Figure 6e).

8. Node 2 receives a REQUEST(5,5) message from node 5. Since node 2 has already reset its $NEXT$ variable to 1, this request is processed by sending a REQUEST(2,5) message to node 1 and setting $NEXT_2 = 5$ (refer to Figure 6f).

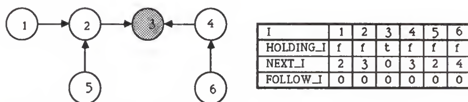
9. Node 1 receives a REQUEST(2,5) message from node 2. Since node 1 a sink, it saves the request by setting $FOLLOW_1 = 5$. Node 1 also sets $NEXT_1 = 2$ and becomes a non-sink (refer to Figure 6g). Note that the global waiting queue of the system at this point consists of 2, 1, 5. This is easily known by following the $FOLLOW$ values starting from the current token holder, node 3.

10. Node 3 leaves its critical section, sends a PRIVILEGE message to node 2, and sets $FOLLOW_3 = 0$ (refer to Figure 6h).

11. Node 2 receives the PRIVILEGE message, enters and leaves its critical section. It then sends a PRIVILEGE message to node 1, and sets $FOLLOW_2 = 0$ (refer to 6(i)).

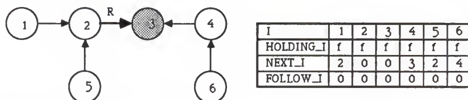
12. Node 1 receives the PRIVILEGE message, enters and leaves its critical section. It then sends a PRIVILEGE message to node 5, and sets $FOLLOW_1 = 0$ (refer to Figure 6j).

13. Node 5 receives the PRIVILEGE message, enters and leaves its critical section. It then sets $HOLDING_5$ = true and waits for a request (refer to Figure 6k).



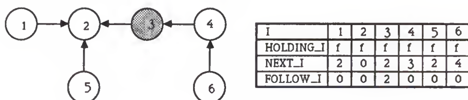
I	1	2	3	4	5	6
HOLDING_I	f	f	t	f	f	f
NEXT_I	2	3	0	3	2	4
FOLLOW_I	0	0	0	0	0	0

Figure 6a. Node 3 is holding the token.



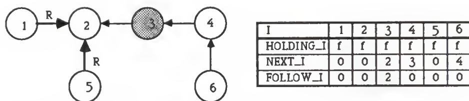
I	1	2	3	4	5	6
HOLDING_I	f	f	f	f	f	f
NEXT_I	2	0	0	3	2	4
FOLLOW_I	0	0	0	0	0	0

Figure 6b. Node 3 enters its critical section.
Node 2 sends a request to node 3.



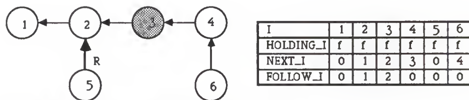
I	1	2	3	4	5	6
HOLDING_I	f	f	f	f	f	f
NEXT_I	2	0	2	3	2	4
FOLLOW_I	0	0	2	0	0	0

Figure 6c. Node 3 processes a request from node 2, sets FOLLOW_3 = 2, and NEXT_3 = 2.



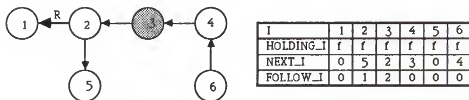
I	1	2	3	4	5	6
HOLDING_I	f	f	f	f	f	f
NEXT_I	0	0	2	3	0	4
FOLLOW_I	0	0	2	0	0	0

Figure 6d. Nodes 1 and 5 send requests to node 2.



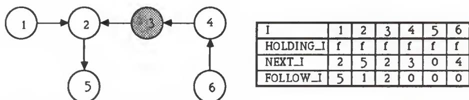
I	1	2	3	4	5	6
HOLDING_I	f	f	f	f	f	f
NEXT_I	0	1	2	3	0	4
FOLLOW_I	0	1	2	0	0	0

Figure 6e. Node 2 processes a request from node 1, sets FOLLOW_2 = 1, and NEXT_2 = 1.



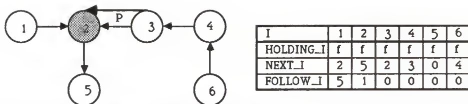
I	1	2	3	4	5	6
HOLDING_I	f	f	f	f	f	f
NEXT_I	0	5	2	3	0	4
FOLLOW_I	0	1	2	0	0	0

Figure 6f. Node 2 processes a request from node 5, sends a request to node 1, and sets NEXT_2 = 5.



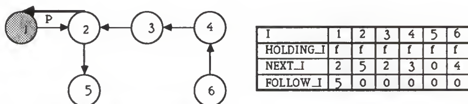
I	1	2	3	4	5	6
HOLDING_I	f	f	f	f	f	f
NEXT_I	2	5	2	3	0	4
FOLLOW_I	5	1	2	0	0	0

Figure 6g. Node 1 processes a request from node 2, sets FOLLOW_1 = 5, and NEXT_1 = 2.



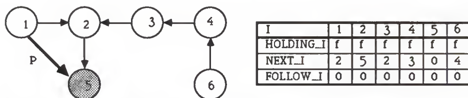
I	1	2	3	4	5	6
HOLDING_I	f	f	f	f	f	f
NEXT_I	2	5	2	3	0	4
FOLLOW_I	5	1	0	0	0	0

Figure 6h. Node 3 leaves its critical section and sends a PRIVILEGE message to node 2.



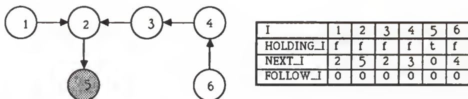
I	1	2	3	4	5	6
HOLDING_I	f	f	f	f	f	f
NEXT_I	2	5	2	3	0	4
FOLLOW_I	5	0	0	0	0	0

Figure 6i. Node 2 enters and leaves its critical section and sends a PRIVILEGE message to node 1.



I	1	2	3	4	5	6
HOLDING_I	f	f	f	f	f	f
NEXT_I	2	5	2	3	0	4
FOLLOW_I	0	0	0	0	0	0

Figure 6j. Node 1 enters and leaves its critical section and sends a PRIVILEGE message to node 5.



I	1	2	3	4	5	6
HOLDING_I	f	f	f	f	t	f
NEXT_I	2	5	2	3	0	4
FOLLOW_I	0	0	0	0	0	0

Figure 6k. Node 5 enters and leaves its critical section and sets HOLDING_5 = true.

Figure 6. Complete Example

Chapter 5

PROOFS

5.1 Mutual Exclusion

In any token-based scheme, possession of the token gives a node the exclusive privilege to enter its critical section. Initially, there is exactly one node holding the token. A node holding the token can pass the token to another node by sending a PRIVILEGE message and setting $HOLDING$ to false. Thus, there can be at most one node holding the token. Since possession of the token is necessary for a node to enter its critical section, mutual exclusion is guaranteed.

5.2 Deadlock and Starvation Freedom

We first recall a few assumptions:

1. A node can have at most one outstanding request to enter its critical section at any given time. We do not allow multiple requests from a single node. Hence, N nodes can have at most N-1 outstanding requests.

2. The initial logical structure is acyclic without considering the directions of the edges. Sending a PRIVILEGE message does not change the graph. Since forwarding a REQUEST message simply changes the direction of an edge, it

does not change the acyclic shape of the graph. Thus, the acyclic structure is always preserved.

3. Initially, exactly one node is possessing the token and in all others, $NEXT$ is initialized to point to the neighboring node which is on the path to the node holding the token.

Let G be the directed acyclic graph (dag) defined by $G = (V, E)$, where

$$V = \{1, 2, \cdots, N\} \text{ and } E = \{(x, y) | x, y \in V \text{ and } y = NEXT_x\}.$$

Lemma 1: If $NEXT_I = 0$, for some I in V, then either node I is holding the token and has not received a request from another node since receiving the token, or node I has requested the token on its own behalf and has not received a subsequent request for the token.

Proof: This is a direct consequence of the initial configuration and the algorithm. In the state transition graph, shown in Figure 4, the shaded regions indicate a sink state, i.e., $NEXT_I = 0$. States E and H indicate that the node is holding the token, and has not received a subsequent request. State R indicates that the node has requested the token on its own behalf and has not received a subsequent request for the token.

Lemma 2: At any point in time, every node I in V is on a path, of length less than N, to a node J in V, such that $NEXT_J = 0$ (i.e. there exists a sequence $I_1(= I), I_2, \cdots, I_m(= J)$ of nodes in V, such that $1 \leq m \leq N, I_1 = I, I_m = J$, and $NEXT_{I_k} = I_{k+1}$ for $k = 1, 2, \cdots, m-1$, and $NEXT_J = 0$).

Proof: Initially, this is true. The only time the path from node I to a sink node changes is when:

1. a node I_j, for some j in $\{1, 2, \cdots, m-1\}$, wants to enter its critical section, sends a REQUEST(I_j, I_j) message to I_{j+1}, and sets $NEXT_{I_j} = 0$ (refer to Figure 7a), or

2. a node I_j, for some j in $\{1, 2, \cdots, m\}$, receives a REQUEST(X_p, X_1) message from a neighboring node X_p, where $X_p \neq I_k$ for all $k \in \{1, 2, \cdots, m\}$, which has been forwarded on the path $X_1, X_2, \cdots, X_p$ (refer to Figure 7b).

We will consider the two cases separately.

1. In this case, $NEXT_{I_j} = 0$. So the lemma is trivially satisfied. The sequence $I_1, I_2, \cdots, I_j$ is a path from node I to node I_j, where $NEXT_{I_j} = 0$ and $j < m \leq N$.

2. In this case, E becomes $E - \{(I_j, I_{j+1})\} \cup \{(I_j, X_p)\}$. Now, we have only three cases to consider:

   2a. no node on the path $X_p, \cdots, X_1$ has received a subsequent request, or

   2b. a node X_q on the path $X_p, \cdots, X_1$ wants to enter its critical section, sends a REQUEST(X_q, X_q) message to X_{q-1}, and sets $NEXT_{X_q} = 0$, or

   2c. a node X_q on the path $X_p, \cdots, X_1$ receives a REQUEST(Y_r, Y_1) message from a neighboring node Y_r not on the path (refer to Figure 7c).

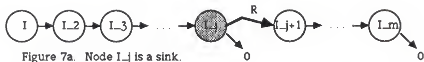


Figure 7a. Node I_j is a sink.

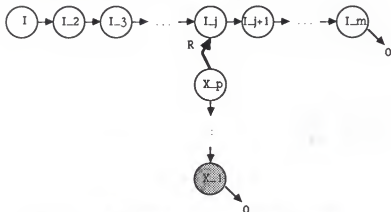


Figure 7b. Node I_j receives a REQUEST(X_p,X_1) message.

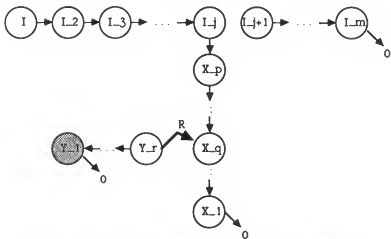


Figure 7c. Node X_q receives a REQUEST(Y_r,Y_1) message.

Figure 7. Case (1), (2), and (2c) in Lemma 2

We consider the three cases separately.

2a. The sequence $I_1, \cdots, I_j, X_p, \cdots, X_1$ satisfies the lemma. Note, due to the acyclic logical stucture imposed on the nodes, $p + j \leq N$.

2b. Case 2b is considered to be the same as case 1, if the sequence

$$I_1, \cdots, I_j, X_p, \cdots, X_q, \cdots, X_1$$

is viewed as the sequence $I_1, \cdots, I_m$, where I_j is viewed as X_q. Thus, the lemma holds.

2c. Case 2c is considered to be the same as case 2, if the sequence

$$I_1, \cdots, I_j, X_p, \cdots, X_q, \cdots, X_1$$

is viewed as the sequence $I_1, \cdots, I_m$, where I_j is viewed as X_q. This reduces to either case 2a, 2b or 2c. In the former two cases, the lemma holds. In case 2c, since there are N nodes, it may recur at most (N-1) times and eventually reaches case 2a or 2b. Thus, the lemma holds.

Theorem 1: The algorithm is deadlock free.

Proof: The only time a node I sets $NEXT_I = 0$ is when it is initially holding the token or has requested the token, but has not received a subsequent request for the token. In either case, a node will save at most one subsequent request for the token by setting its $FOLLOW$ variable to point to the node originating the request. If more requests are received, they will simply be forwarded.

By lemma 2, every node I is always on a path of length less than N, to a node J, such that $NEXT_J = 0$. Suppose there are $1 \leq k \leq N$ requests for the token from nodes $I_1, \cdots, I_k$. Also, suppose I_j is holding the token.

If $j \notin \{1, \cdots, k\}$ then the request which reaches node I_j first will be granted first. All nodes are arranged in an acyclic graph. Since all requests are forwarded to a node X where $X \in \{I_1, \cdots, I_k\}$, or to node I_j, at least one request must be forwarded to node I_j. Otherwise, all k requests are forwarded to the k requesting nodes; this means we have a cycle. Without loss of generality, we may assume that the request from node I_i is received by node $I_{i+1}, i = 1, 2, \cdots, k-1$, after node I_{i+1} has already requested, and the request from node I_k is received by node I_j (i.e. renumber the nodes if necessary). Then each of the nodes will set $FOLLOW_{I_{i+1}} = I_i$. Naturally, node I_k will receive the token first. Then the token will be forwarded in the order $I_k \rightarrow I_{k-1} \rightarrow \cdots \rightarrow I_1$ and each of the requests will be satisfied.

Similarly, if $j \in \{1, \cdots, k\}$, the theorem holds. Since I_j holds the token, it enters its critical section and then the above argument applies. Therefore, deadlock cannot occur.

Theorem 2: The algorithm is starvation free.

Proof: By our preceding argument, the only time starvation could occur is if a small group of nodes are allowed to retain possession of the token while other nodes have requested the token.

Suppose node I wants to enter its critical section and sends a REQUEST(I,I) message to $NEXT_I$. By Lemma 2, any request is guaranteed to reach a sink node in less than N messages. Once a request reaches a sink node, it will be immediately served (if the sink node holds the token), or stored in the $FOLLOW$ variable and be eventually served. This is because of the total ordering of requests given by the $FOLLOW$ variables in the preceding argument. When a node J leaves its critical

section it must send the token to $FOLLOW_J$ if $FOLLOW_J \neq 0$. Therefore, the algorithm is starvation free.

Chapter 6

PERFORMANCE ANALYSIS

As in Raymond's algorithm, the performance of the algorithm depends on the logical topology of the dag structure. The worst topology, in terms of the number of messages required per critical section entry, is a straight line, as shown in Figure 2. In [4], they indicate that the best topology is a radiating star formation. However, the best topology is what we call a *centralized topology*, with one node in the center and all other nodes as leaf nodes (refer to Figure 8).

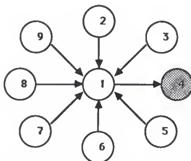


Figure 8. Centralized Topology

In the following discussion, we define the *diameter* D of the topology to be the length of the longest path.

6.1 Upper Bound

The upper bound is equal to $(D + 1)$ messages per critical section entry. This occurs when a requesting node and a sink node are at opposite ends of the longest path: D messages for the request to travel to the sink and one message for the token to be sent back to the requesting node. Thus, in the straight line topology, the upper bound is N, where N is the number of nodes in the system. In the best topology, the upper bound is 3, since the diameter of the *centralized topology* is 2. Note that this is the same as the performance of a centralized mutual exclusion algorithm, where one REQUEST message, one GRANT message and one RELEASE message are required.

For comparison, other algorithms have the following upper bounds:

- Lamport's algorithm : $3 * (N - 1)$
- Ricart and Agrawala's algorithm : $2 * (N - 1)$
- Carvalho and Roucairol's algorithm : $2 * (N - 1)$
- Suzuki and Kasami's algorithm[1] : N
- Singhal's algorithm : N
- Maekawa's algorithm : $7 * \sqrt{N}$
- Raymond's algorithm : $2 * D$ (i.e., 4 in a centralized topology)

6.2 Average Bound

We analyze the average performance for the best topology. If the requesting node holds the token, it requires no messages. If the token is being held by a

[1] This algorithm is essentially the same as Ricart and Agrawala's algorithm [5].

leaf node, then on the average $3 - 4/N$ messages per critical section entry are required. This is calculated as follows: The other $(N - 2)$ leaf nodes require 3 messages (refer to section 5.1). The center node requires 2 messages: one REQUEST message and one PRIVILEGE message. Therefore, the average is $((N - 2) * 3 + 1 * 2)/N = 3 - 4/N$.

If the token is being held by the center node, then only $2 - 2/N$ messages are required: $((N-1)*2+1*0)/N = 2-2/N)$. We assume that at any given time each node has an equal likelihood of holding the token. There are $(N-1)$ leaf nodes and one center node; therefore, on the average, $((N-1)*(3-4/N)+1*(2-2/N))/N = 3-5/N+2/N^2$ messages are required per critical section entry. In the centralized scheme, on the average, $(3-3/N)$ messages per critical section entry are required[2]. Both methods approach 3 messages per critical section entry as N approaches infinity. Under heavy demand, the performance is about the same, i.e., at most three messages per critical section entry.

6.3 Synchronization Delay

Synchronization delay is the maximum number of sequential messages required after a node I leaves its critical section before a node J can enter its critical section. We assume that the request from node J is to be processed next and node J is blocked waiting for node I to complete its critical section. In this case, $FOLLOW_I = J$ and node J will be passed the token immediately after node I leaves its critical section. Since only one PRIVILEGE message needs to be passed, the synchronization delay is minimal, i.e., one message. This is even better than a centralized scheme in which the synchronization delay is two: one RELEASE

[2]We assume that a control node may request to enter its critical section. In which case, it requires no message. Thus, $(3 - 3/N)$ messages are required: $((N - 1) * 3 + 1 * 0)/N$.

and one GRANT message.

Other token based algorithms have the following synchronization delays:

- Suzuki and Kasami's algorithm : 1
- Singhal's algorithm : 1
- Raymond's algorithm : D.

6.4 Storage Overhead

Each node maintains three simple variables. A REQUEST message carries two integer variables, and a PRIVILEGE message needs no data structure. This is significantly less overhead compared with other distributed mutual exclusion algorithms, where they maintain an array structure or a waiting queue of requesting nodes, either in every node or within the token.

Chapter 7

CONCLUSION

This paper presented a token based algorithm for distributed mutual exclusion which assumes a fully connected physical network and a dag structured logical network. The algorithm imposes very little storage overhead on each node and message.

In the *centralized topology*, the algorithm attains comparable performance to centralized schemes. On the average, about three messages are required per critical section entry in both schemes. However, our scheme reduces the amount of synchronization delay to one message compared with two messages in centralized schemes.

Bibliography

[1] S. F. Carvalho and G. Roucairol. On mutual exclusion in computer networks. *Communications of the ACM*, 26(2):146–147, 1983.

[2] L. Lamport. Time, clocks and ordering of events in distributed systems. *Communications of the ACM*, 21(7):558–564, 1978.

[3] M. Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, 1985.

[4] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, 1989.

[5] G. Ricart and A. K. Agrawala. Author's response to 'on mutual exclusion in computer networks' by Carvalho and Roucairol. *Communications of the ACM*, 26(2):147–148, 1983.

[6] G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, 1981.

[7] B. Sanders. The information structure of distributed mutual exclusion algorithms. *ACM Transactions on Computer Systems*, 5(3):284–299, 1987.

[8] M. Singhal. A heuristically-aided algorithm for mutual exclusion in distributed systems. *IEEE Transactions on Computers*, 38(5):651–662, 1989.

[9] I. Suzuki and T. Kasami. A distributed mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 3(4):344–349, 1985.

A DAG-BASED ALGORITHM FOR DISTRIBUTED MUTUAL EXCLUSION

by

Mitchell L. Neilsen

AN ABSTRACT OF A THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1989

Abstract

The paper presents a token based distributed mutual exclusion algorithm. The algorithm assumes a fully connected reliable physical network and a directed acyclic graph (dag) structured logical network.

The number of messages required to provide mutual exclusion is dependent on the logical topology of the nodes. Using the best topology, the algorithm attains comparable performance to a centralized mutual exclusion algorithm; i.e., three messages per critical section entry. It also achieves minimal synchronization delay.

In our algorithm, no node or message explicitly holds a waiting queue of pending requests. The queue is maintained implicitly in a distributed fashion among nodes; at any given time, the queue may be constructed by observing the states of the nodes. As a result, the algorithm imposes very little storage overhead; each site maintains only a few simple variables, and the token carries no data structure.