A Dag-Based Algorithm for Distributed Mutual Exclusion

Mitchell L. Neilsen  Masaaki Mizuno*

Department of Computing and Information Sciences
Kansas State University
Manhattan, Kansas 66506

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 upon the logical topology im-
posed on the nodes. Using the best topology, the algorithm attains comparable performance to a cen-
tralized mutual exclusion algorithm; i.e., three messages per critical section entry. Also, the algorithm achieves
minimal heavy-load synchronization delay and imposes very little storage overhead.

1 Introduction

Many distributed mutual exclusion algorithms have been proposed [1, 2, 3, 5, 6, 7, 8, 9, 11, 12, 13, 14, 16, 17,
18, 19, 20, 21, 22, 23]. These algorithms can be classi-

cified into two groups [15, 20]. The algorithms in the first
group are permission based [1, 3, 6, 7, 14, 16, 18, 19].
A node enters the critical section only after receiving
permission from a set of nodes. The algorithms in the
second group are token based [2, 5, 8, 9, 11, 12, 17,
20, 21, 22, 23]. The possession of a system-wide unique
token gives a node the right to enter the critical section.

Lamport proposed one of the first distributed mutual
exclusion algorithms [6]. The algorithm is permission
based and requires $3 \cdot (N - 1)$ messages to provide mu-
tual exclusion. Another permission based algorithm,
proposed by Ricart and Agrawala, reduced the number
of required messages to $2 \cdot (N - 1)$ messages per criti-
cal section entry [16]. By using a different definition of
symmetry, Carvalho and Roucairol were able to reduce
the number of messages to be between 0 and $2 \cdot (N - 1)$
[3]. Maekawa proposed a permission based algorithm
in which the number of messages required is $O(\sqrt{N})$
[7]. Sanders generalized permission based algorithms
by defining the information structure which each node maintains [18].

In response to Carvalho and Roucairol's algorithm, Ricart and Agrawala propose a token based algo-

rithm [17] which is essentially the same as Suzuki
and Kasami's algorithm [21]. 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 [20]. The
maximum number of messages required by these three
algorithms is $N$. Recently, Nishio, et. al., proposed a
failure recovery algorithm which can be used with these

By imposing a tree-based logical structure on the
nodes, another class of token based algorithms has been
obtained. In this class of algorithms, all of the nodes,
except for the root node, are on a path to the root
node (a sink node) in the logical structure. The logical
structure determines the path along which a request
message travels. There are two different types of logi-
cal structures: dynamic and static. For node $i$, define
NEIGHBOR$_i$ to be the set of nodes $j$ such that there is
an edge from node $i$ to node $j$ or from node $j$ to node $i$.

A path reversal at a node $z$ is performed as the request by node $x$ travels along the path from node $x$ to the root node. As the request travels,
node \( z \) becomes the new parent of each node on the path, except for node \( z \). Thus, node \( z \) becomes the new root node. A complete analysis of path reversal has been given by Ginat [4]. The average number of messages required per critical section is \( O(\log(N)) \).

If a static logical structure is used, the basic notion underlying the algorithm is what we will call edge reversal. An edge reversal at a node \( z \) is performed as the request by node \( z \) travels along the path from node \( z \) to the root node. As the request travels, the direction of each edge on the path is changed to point towards node \( z \). However, the shape of the logical structure never changes. Surprisingly, this small change results in algorithms which have a small fixed upper bound on the number of messages required per critical section entry, and the upper bound only depends on the logical structure. One such algorithm was proposed by Raymond [12]. The algorithm assumes that the static logical structure is an unrooted tree. If the radiating star topology is used, the average number of messages required is \( O(\log(N)) \). However, this is not the optimal topology, as we shall show. A similar algorithm was proposed by van de Snepscheut [23].

This paper presents a token-based algorithm which further improves Raymond's tree-based approach. The algorithm assumes a fully connected physical network and a static, directed acyclic graph (dag) structured logical network. It reduces the number of required messages to roughly half the number required in Raymond's algorithm. The best logical topology, in terms of the maximum number of messages required per critical section, is the star topology. Using the star topology, the maximum number of messages required is three, which is the same performance exhibited by centralized schemes.

Furthermore, the heavy-load synchronization delay (the maximum number of sequential messages required between one node leaving the critical section and another waiting node entering the critical section) is minimal, i.e., one message. The light-load synchronization delay (the maximum number of sequential messages required for a critical section entry assuming no pending requests exist when a request is issued) is the diameter of the logical topology (i.e., the length of the longest path in the topology) plus one.

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. The algorithm requires very simple data structures; each node maintains a few simple variables, and the token carries no data structure.

Section 2 informally describes the algorithm by using a simple example. The detailed algorithm is presented in Section 3. Section 4 presents proofs of correctness with respect to guaranteed mutual exclusion, deadlock freedom, and starvation freedom. Section 5 analyzes the performance of the algorithm.

## 2 Overview of the algorithm

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 the critical section. Physically, the nodes are fully connected by a reliable network. Logically, they are arranged in a dag structure with only one sink node. The out 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.

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

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

The NEXT variable indicates the node which will be granted mutual exclusion after this node. If the node is currently the last node to be granted mutual exclusion, its NEXT variable is 0. Thus, by following the NEXT variables from the token holder to the node whose NEXT variable is 0, the implicit waiting queue of the system can be deduced. When a node leaves the critical section, it forwards the token to the node at the top of the implicit waiting queue and performs a dequeue operation. That is, it sends a PRIVILEGE message to the node indicated by its NEXT variable (this is the node at the top of the queue) and clears the variable (this corresponds to the dequeue operation), unless NEXT is 0. If NEXT is 0, the node continues to hold the token. This case is explained below.
Semantically, a sink node in the system is (1) the last node in the implicit waiting queue (i.e., its NEXT variable is 0), and (2) the last node in the path along which a request travels (i.e., its LAST 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. Each edge in the path must change direction to point in the direction of the new sink. This is done by 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 LAST variable and sets its LAST 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 neighboring node X, it passes the message to the neighboring node indicated by its LAST variable. Then, the node sets its LAST variable to X. Thus, if it receives another request later, it forwards the request in the direction of the new sink. In Trehel and Naimi's algorithm, the LAST variable is set to the node that initiated the request and not to the neighboring node on the path to the node that initiated the request.

- When a sink node receives a REQUEST message from a node X, it sets its NEXT variable to the identifier of the node initiating the request. This corresponds to an enqueue operation. The node also sets its LAST 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 the 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 are several sink nodes in the system while requests are in transit. Assume that two nodes, X and 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 node 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. Eventually, node Y becomes the only sink node in the system.

The system is initialized so that one node possesses the token. This is the sink node, and its LAST variable is initialized to 0. In all other nodes, LAST is set to point to the neighbor which is on the path to the node holding the token.

Consider the example given in Figure 1. Node 5 holds the token initially. Let the directed edges indicate the direction in which the LAST variables are pointing. The initial configuration is shown in Figure 1a. Suppose node 5 wants to enter the critical section.

![Figure 1a](image1)

Figure 1a.

Now, suppose node 3 wants to enter the critical section. It sends a REQUEST message to node 4 and sets its LAST variable to 0 to become a new sink (refer to Figure 1b). Node 4 receives the request and sets its LAST variable to point to node 3 and forwards the REQUEST message to node 5, on behalf of node 3 (refer to Figure 1c). Node 5 receives the REQUEST message. Since node 5 is a sink node, it sets its NEXT variable to point to node 3 and sets its LAST variable to point to node 4 to become a non-sink. When node 5 leaves the critical section, it sends a PRIVILEGE message to the node indicated by its NEXT variable, i.e., node 3. Finally, node 3 receives the PRIVILEGE message and enters the critical section (refer to Figure 1d).

![Figure 1b](image2)

Figure 1b.

![Figure 1c](image3)

Figure 1c.

![Figure 1d](image4)

Figure 1d.

**Figure 1. Simple example**
3 The Algorithm

The complete algorithm is shown in Figure 2. There are two procedures at each node: P1 and P2. Procedure P1 is executed when node I requests for entry into the critical section, and procedure P2 is executed when node I receives a request from some other node.

const

\( I = \) node identifier

var

HOLDING : boolean;
LAST, NEXT : integer;

procedure P1; (* Enter critical section *)
begin
  if (not HOLDING) then
    begin
      send REQUEST(I, I) to LAST;
      LAST := 0;
      wait until a PRIVILEGE message is received;
    end;
  HOLDING := false;
  critical section
  if (NEXT \# 0) then
    begin
      send PRIVILEGE message to NEXT;
      NEXT := 0;
    end;
  else HOLDING := true;
end;

procedure P2; (* Handle REQUEST(X, Y) msg *)
begin
  if (LAST = 0) then
    begin
      if HOLDING then
        begin
          send PRIVILEGE message to Y;
          HOLDING := false;
        end;
      else NEXT := Y;
    end;
  else send REQUEST(I, Y) to LAST;
  LAST := X;
end;

Figure 2. Algorithm

In the algorithm, we assume that the REQUEST message is of 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 the critical section.

The initialization procedure is shown in Figure 3.

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

Figure 3. Initialization procedure

4 Properties of the Algorithm

In this section we sketch the proofs of correctness with respect to guaranteed mutual exclusion, deadlock freedom, and starvation freedom. Complete proofs may be found in a KSU technical report [10].

4.1 Mutual exclusion

Theorem 1: The algorithm guarantees mutual exclusion.

Proof: In any token based mutual exclusion algorithm, possession of the token gives a node the exclusive privilege to enter the 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 the critical section, mutual exclusion is guaranteed.

4.2 Deadlock and starvation freedom

We first recall a few assumptions:

1. A node can have at most one outstanding request to enter the critical section at any given time. We do not allow multiple requests from a single node.

2. The initial logical structure is acyclic without considering the directions of the edges. Sending a PRIVILEGE message does not change the graph. Forwarding a REQUEST message simply results in edge reversal. Thus, the logical structure is always acyclic.

3. Initially, exactly one node possesses the token and in all other nodes, LAST is initialized to point to the neighboring node which is on the path to the node holding the token.

We will give a sketch of the proof for deadlock and starvation freedom. Let LAST\(_x\) denote the value of LAST at node \(x\) and NEXT\(_x\) denote the value of NEXT at node \(x\).

**Lemma 1:** If LAST\(_I\) = 0, for some node \(I\), 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.

**Lemma 2:** At any point in time, every node \(I\) is on a path, of length less than \(N\), to a node \(J\), such that LAST\(_J\) = 0. Furthermore, a request from node \(I\) will reach a sink node after at most \(D\) messages are forwarded on behalf of node \(I\), where \(D\) is the diameter of the logical structure.

**Theorem 2:** The algorithm is deadlock and starvation free.

**Proof:** By Lemma 1, the only time a node \(J\) sets LAST\(_J\) = 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 NEXT variable to point to the node originating the request. If more requests are received, they will simply be forwarded.

By Lemma 2, every request from node \(I\) will reach a sink node (say node \(J\)) after at most \(D\) messages are forwarded on behalf of node \(I\).

Once the request from node \(I\) has been enqueued in the implicit waiting queue (node \(J\) sets NEXT\(_J\) = \(I\)), node \(I\) will only have to wait until every other node ahead of node \(I\), in the implicit waiting queue, enters and leaves the critical section. Since each node can have at most one outstanding request, this means that node \(I\) will have to wait on at most \(N - 1\) critical sections after it has been enqueued.

Therefore, deadlock and starvation freedom are guaranteed.

5 Performance Analysis

As in Raymond's algorithm, the performance of the algorithm depends on the topology of the logical structure. The worst topology, in terms of the number of messages required per critical section entry, is a straight line, as shown in Figure 1. Raymond claimed that the best topology is the radiating star topology [12]. However, the best topology is the star topology, with one node in the center and all other nodes as leaf nodes. In the following discussion, we define the diameter \(D\) of the topology to be the length of the longest path.

**5.1 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 node 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 star topology is 2. Note that this is the same as the performance of a centralized mutual exclusion algorithm.

Other algorithms have the following upper bounds:

- Suzuki and Kasami's algorithm : \(N\)
- Trehel and Naimi's algorithm : \(N\) \((O(log(N))\) on the average)
- Raymond's algorithm : \(2 \times D\)
5.2 Average bound

We analyze the average performance for the best topology. If the requesting node holds the token, no messages are required. If the token is being held by a 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 and the center node requires 2 messages: one REQUEST message and one PRIVILEGE message. Thus, 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. We assume that a control node may request to enter the critical section. In which case, it requires no message. Thus, (3 – 3/N) messages are required: ((N – 1) * 3 + 1 * 0)/N. 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.

5.3 Storage overhead

Each node only needs to maintain three simple variables. A REQUEST message carries two integer variables, and a PRIVILEGE message needs no data structure. This is the same as Trehel and Naimi’s algorithm and significantly less overhead compared with other distributed mutual exclusion algorithms which maintain an array structure or a waiting queue of requesting nodes, either in every node or in the token.

5.4 Synchronization delay

Synchronization delay is the maximum number of sequential messages required after a node, say node I, leaves the critical section and before another node, say node J, can enter the critical section.

5.4.1 Heavy-load

Under heavy-load, we assume that the request from node J is to be processed next and that node J is blocked waiting for node I to leave the critical section. In this case, NEXT_I = J and node J will be passed the token immediately after node I leaves the critical section. Since only one PRIVILEGE message needs to be passed, the heavy-load synchronization delay is one message. This is even better than a centralized scheme in which the heavy-load synchronization delay is two: one RELEASE and one GRANT message.

Other token based algorithms have the following heavy-load synchronization delays:
- Suzuki and Kasami’s algorithm: 1
- Trehel and Naimi’s algorithm: 1
- Raymond’s algorithm: D

5.4.2 Light-load

Under light-load, we assume that the request from node J is to be processed next and that node J has not requested when node I leaves the critical section. In this case, NEXT_I = 0 and node J will have to initiate a REQUEST message to obtain the token. Since at most D messages are required to forward the REQUEST and only one PRIVILEGE message is required to forward the token, the light-load synchronization delay is D + 1 messages. This is not as good as a centralized scheme in which the light-load synchronization delay is two: one REQUEST and one GRANT message.

Other token based algorithms have the following light-load synchronization delays:
- Suzuki and Kasami’s algorithm: 2
- Trehel and Naimi’s algorithm: O(log(N)) (on the average)
- Raymond’s algorithm: 2 * D.

6 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 in each message.

If the logical structure used is the star topology, the algorithm attains comparable performance to centralized schemes. On the average, about 3 messages are required per critical section entry in both schemes. However, our scheme reduces the heavy-load synchronization delay to one message compared with two messages in centralized schemes.
References


