Real-Time Token-Based Mutual Exclusion Algorithms

Mitchell L. Neilsen
Department of Computing and Information Sciences
Kansas State University
Manhattan, KS, USA
{neilsen@ksu.edu}

Abstract

Many token-based, distributed mutual exclusion algorithms can be generalized by a single algorithm. The algorithm’s performance is dependent upon the logical topology imposed on the nodes and the policy used to forward requests.

This paper extends the generalized algorithm to support prioritized, real-time requests in a generalized fashion and presents models that can be used to analyze the performance and verify the correctness of the generalized algorithm. Both safety and liveness properties are verified. Model checking is also used to analyze performance. Using the best topology, the generalized algorithm attains the same worst-case performance as a centralized algorithm; i.e., three messages per critical section. In the average case, the generalized algorithm performs better than a centralized one when the star topology is used. Finally, requests by nodes at each priority level are processed in order, resulting in bounded, predictable worst-case response times.

Keywords: distributed algorithm, model checking, mutual exclusion, real-time, token-based

1 Introduction

Many distributed mutual exclusion algorithms have been proposed [1, 4, 6, 8, 12, 13, 16, 17]. These algorithms can be classified into two groups [13]. The algorithms in the first group are called permission-based [1, 4, 6]. A node enters its critical section only after receiving permission from a quorum of nodes. The algorithms in the second group are called token-based [8, 12, 16, 17]. The possession of a system-wide unique token gives a node the right to enter its critical section.

Lamport proposed one of the first distributed mutual exclusion algorithms [4]. Lamport’s algorithm is permission-based and requires $3 \times (N - 1)$ messages to provide mutual exclusion. Another permission-based algorithm, proposed by Ricart and Agrawala, reduces the number of required messages to $2 \times (N - 1)$ messages per critical section entry [14]. Maekawa proposed a permission-based algorithm in which the number of messages required is $O(\sqrt{N})$ [6].

Ricart and Agrawala proposed a token-based algorithm which is essentially the same as Suzuki and Kasami’s algorithm [16]. The maximum number of messages required by these algorithms is $N$ because request messages are sent to all other nodes, and the token is passed in a single message.

By imposing a tree-based logical structure on the nodes, another class of token-based algorithms has been obtained. 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 logical structures: dynamic and static.

An algorithm, based on a dynamic logical structure, was proposed by Trehel and Naimi [17]. The basic notion underlying this algorithm is path reversal. Path reversal at each node is performed as a request from node $x$ travels along the path from node $x$ to the root node. As the request travels, node $x$ becomes the new parent of each node on the path, except for node $x$. Thus, node $x$ becomes the new root node. A complete analysis of path reversal has been given by Ginat [3]. 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 call edge reversal [11]. Edge reversal at each node is performed as the request from node $x$ travels along the path from node $x$ to the root node. At each node, the direction of each edge on the path is changed to point towards node $x$; that is, to the neighboring node who sent the request on behalf of node $x$. However, the shape of the logical structure never changes. Suprisingly, this small change results in algorithms which have a small fixed upper bound on the number of messages required per critical section, and the upper bound only depends on the logical structure. Algorithms based on edge-reversal were proposed by Neilsen and Mizuno [11] and Raymond [12]. Raymond’s algorithm assumes that the static logical structure is an unrooted tree. If the ra-
2 Generalized Algorithm

We assume that the system consists of \( N \) nodes, which are uniquely numbered from 0 to \( N - 1 \). At any 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, but logically, the nodes at each priority level are organized in a directed acyclic graph (dag). Nodes in different levels are assigned different priorities.

Two types of messages, called REQUEST and TOKEN, are exchanged among nodes. When a node wants to obtain the token to enter its critical section, it sends a REQUEST message. A TOKEN message represents the token; when a node receives a TOKEN message, it may enter its critical section.

Each node maintains three simple variables: integer variables \( \text{LAST} \) and \( \text{NEXT} \), and a boolean variable \( \text{HOLDING} \) or \( \text{SINK} \). The logical directed acyclic graph (dag) structure indicates the path along which a REQUEST message travels and is imposed by the \( \text{LAST} \) 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 \( \text{LAST} \) variable (unless the node is a sink, in which case its \( \text{LAST} \) variable is -1).

The \( \text{NEXT} \) 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 \( \text{NEXT} \) variable is -1. Thus, by following the \( \text{NEXT} \) variables from the token holder to the node whose \( \text{NEXT} \) variable is -1, the implicit waiting queue of pending requests can be deduced. When a node leaves its critical section, it forwards the token to the node at the front of the waiting queue and also performs a dequeue operation. That is, it sends a TOKEN message to the node indicated by its \( \text{NEXT} \) variable and sets \( \text{NEXT} \) to -1 (this corresponds to the dequeue operation), unless \( \text{NEXT} \) is already -1. If \( \text{NEXT} \) is -1, the node continues to hold the token if it is at the highest priority level by setting \( \text{HOLDING} \) to true, otherwise the token is returned to the highest priority level as described below.

Semantically, a sink node in the system is (1) the last node in the implicit waiting queue (i.e., its \( \text{NEXT} \) variable is -1), and (2) the last node on the path along which a request travels within a given priority level (i.e., its \( \text{LAST} \) variable is -1). When a sink node receives a REQUEST message, it \text{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 neighbor indicated by its \( \text{LAST} \) variable and sets its \( \text{LAST} \) variable to -1 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 \( \text{LAST} \) variable to \( \text{any} \) node on the path traveled by the REQUEST message. If a node receives a subsequent request, it forwards the request in the direction of the new sink. In Trehel and Naimi's algorithm, the \( \text{LAST} \) variable is set to the node that initiated the request; this is called path reversal. In Neilsen and Mizuno's algorithm, the \( \text{LAST} \) variable is set to point to the neighboring node from which it received the REQUEST message; this is called edge reversal.

When a sink node receives a REQUEST message, 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 \( \text{any} \) node on the path traveled by the REQUEST message. Note that if a sink node holds the token, but is not in its critical section (indicated by a boolean variable \( \text{HOLDING} \)) when it receives a request, it immediately enters its critical section (indicated by a boolean variable \( \text{HOLDING} \)). When a sink node receives a REQUEST message, it forwards the token to the node initiating the request. This corresponds to an enqueue operation.

The complete generalized algorithm for a single priority level is shown in Figure 1. There are two procedures at each high priority node: ProcessWork and ProcessRequest. Procedure ProcessWork is executed when a high priority node \( I \) requests for entry into its critical section, and procedure ProcessRequest is executed when a high priority node \( I \) receives a request from some other high priority node. In the algorithm, REQUEST messages are of form \( \text{REQUEST}(X_1, X_2, \cdots, X_k) \) where \( X_1, X_2, \cdots, X_k \) denotes the path on which the request traveled and \( X_1 \) denotes the node where the request originated. Each node executes procedures ProcessWork and ProcessRequest in local mutual exclusion. The only exception is that a node does not have to execute in local mutual exclusion while waiting for a TOKEN message to arrive or while in its critical section.

```
const
I : node identifier

var
HOLDING : boolean;
LAST, NEXT : integer;

proc ProcessWork;
begin
  if (not HOLDING) then begin
    send REQUEST(I) to LAST;
    LAST := -1;
    wait until a TOKEN message is received;
    critical section (CS)
    hold until a TOKEN message is received;
    NEXT := -1;
    if (NEXT \neq -1) then begin
      send REQUEST to NEXT;
      hold until a TOKEN message is received;
    end;
    else HOLDING := true;
  end;
end;

proc ProcessRequest; (receive REQUEST(X_1, \cdots, X_k))
begin
  if (LAST = -1) then begin
    hold until a TOKEN message is received;
    send REQUEST to X_1;
    if HOLDING then begin
      HOLDING := true;
      NEXT := X_1;
    end;
    else send REQUEST(X_1, X_2, \cdots, X_k, I)
    to LAST;
    if HOLDING then begin
      HOLDING := false;
      NEXT := X_1;
    end;
  end;
end;
```

Figure 1. Generalized algorithm

Because of message delay, there may be several sink nodes in the system while requests are in transit. The system is initialized so that only one node at the highest priority level possesses the token. Initially, there is only one sink node at each priority level, and its LAST variable is initialized to \(-1\). In all other nodes, LAST is set to point to the neighbor which is on a path to a sink node. When a request reaches a sink node at a lower priority level, the request is forwarded as a proxy request to a node acting as the lower level’s proxy at the highest priority level. Proxy requests are forwarded just like ordinary requests, but no edges or paths are reversed as the proxy requests travel to a sink node (or some node with a pending request).

The prioritized algorithm for low priority nodes is similar, except that the boolean variable \( \text{HOLDING} \) is replaced with \( \text{SINK} \) as shown in Figure 2. Initially, a node in the highest priority level holds the token, and \( \text{HOLDING} \) for that node is set to true. Likewise, the root node at each lower priority level has its \text{SINK} variable set to true. When a request from a low priority node reaches the sink node, a PROXY REQUEST is sent up to a node at the highest priority level, called the proxy, and eventually the PROXY REQUEST reaches a node that holds the token or will receive the token in the future. To further generalize the algorithm, requests or proxy requests from lower priority nodes only need to be passed to the point where they reach a node that is currently requesting the token. The key point is that all low priority requests are eventually enqueued in the token. Finally, if a request reaches the highest priority level, then it is forwarded just like a regular request, except that the edges are not reversed and in the generalized case the request only needs to reach a node that is cur-

Figure 2. Two priority levels
rently requesting the token. Note that higher priority nodes never request the token from lower priority nodes so there is no need to dynamically adjust the edges. When a node that is holding the token passes the token to a lower priority node, a PROXY TOKEN message is used to pass the token, and the token is returned back to the high priority node who sent the token using a PROXY RETURN message.

3 Verification Model

In this section we provide models that can be used to verify the correctness of the generalized algorithm with respect to guaranteed mutual exclusion, deadlock freedom, and starvation freedom using UPPAAL [2]. The model for a single priority level consists of two templates, ProcessWork and ProcessRequest, corresponding to the procedures shown above in Figure 1. Channels are used to model the exchange of request messages and the token. Global arrays are used to model the state at each node using the arrays Holding, Sink, Next, and Last as defined above. Initially, one node will hold the token, so Holding[1] = true at that node. Also, the topology is defined by the initial values assigned to Last.

The first UPPAAL template, ProcessWork, is shown below in Figure 3. It models the work performed at each node. Initially, all nodes are in the Idle state. The template is parameterized using id to identify each node where id ∈ {0, 1, · · · , N − 1}. At node 0, id = 0, etc. Also, Holding[id] is set to true at the node currently holding, but not using, the token. This node can enter its critical section immediately after setting Holding[id] to false to indicate that the token is in use.

![Figure 3. ProcessWork template](image)

All other nodes must send a request message to the node identified by LAST[id], and set LAST[id] = -1. Upon receipt of the token via a Token message, the requesting node may enter its critical section. A local clock, x, is used to prevent a node from remaining in its critical section forever. The model limits critical sections to be at most 10 time units through the location invariant x ≤ 10.

To process requests, the ProcessRequest template is used as shown in Figure 4. When a request message is received, at node id from node p, using Request[p][id][s]?, the source node requesting to enter its critical section is node s.

If the node receiving the request is a sink node, Last[id] == -1, the request can be satisfied immediately if the node receiving the request is holding, but not using, the token; that is, if Holding[id] == true. In this case, the Token message can be sent immediately. On the other hand, if the token is currently in use, then the request is simply enqueued by setting Next[id] = t which is a local meta variable assigned to s when the request is received.

If the node receiving the request is not a sink node, Last[id] ≥ 0, then the request is forwarded on to the node indicated by Last[id].

In all cases, Last[id] is set to the identifier of the neighboring node which sent the request; that is, edge reversal is used in Figures 3 and 4. The generalized algorithm is slightly more complex because each node can set its Last[id] value to be any node visited from the requesting node to the sink. Consequently, a list or queue of visited node numbers must be carried with the Request message. This is modeled with a set of global queues that get updated as the Request travels. One queue is assigned to each node and used to enqueue the nodes on the path from the given requesting node to a sink.

![Figure 4. ProcessRequest template](image)

The nodes can be initialized to impose any logical topology. For example, to impose the star topology, with node 0 as the root, simply set Holding[i] = false and Last[i] = 0 for all i ≠ 0, and Holding[0] = true and Last[0] = -1.

The generalized ProcessWork template is shown in Figure 5. By making Init a committed state, all nodes will enter the Ready state before any nodes start requesting. The only other change required is to add a function, initRequest(id), that is used to initialize the queue passed with the Request message to contain a single element id.
The generalized ProcessRequest template is shown in Figure 6. As the Request message travels from a requesting node to a sink node, the identifier of each node receiving the request must be enqueued in the request message. Since UPPAAL messages have zero capacity, this is modeled using a set of global queues and a function call enQueue(t, id) to enqueue id on the queue for the requesting node t. Also, each node on the path sets Last[id] to be some element in the queue.

The function, someQueue(), relies on a random number generator, modeled by the RandomValue template shown below, to randomly select a random element from the queue carried with the request. Finally, to support different priority levels, we can add the notion of a proxy node at the highest priority level, and use algorithms similar to the above to request the token.

Due to space constraints, we only include the prioritized models for edge reversal (extending the templates shown in Figures 3 and 4), but the generalized case is similar.

4 Performance Analysis

To verify the correctness of the algorithm we have developed analytical proofs of correctness. We have also verified both safety and liveness properties using UPPAAL [2] for models such as those shown in Figure 2. To initialize the system for that model, abbreviate names by setting P(i) = ProcessWork(i) and R(i) = ProcessRequest(i) for i=0,1,2; P(i) = ProcessWorkLow(i) and R(i) = ProcessRequestLow(i) for i=3,4. It is easy to verify that the algorithm satisfies mutual exclusion. The property A[] (forall (i:node) forall (j:node)
\[(i==j) \text{ or not } (P(i).CS \text{ and } P(j).CS)\]
is satisfied, where \textit{node} is defined as a new type \texttt{int}[0,4]; that is, only one process can be in its critical section (state \texttt{CS}) at any time. To verify liveness properties, we first limit the amount of time each process can remain in its critical section forever. The \texttt{CS} state has a location invariant of \(x \leq 10\) for a real-valued clock \(x\) which is initialized to 0 upon entry to the critical section state. The choice of ten time units is arbitrary. To verify that a node that wants to enter its critical section can enter, we verify the property \(P(i).\text{Requesting} \rightarrow P(i).\text{CS}\); that is, requesting the critical section “leads to” entry. Formally, a process, at the highest priority level, in the \texttt{Requesting} state eventually reaches the \texttt{CS} state.

To verify real-time behaviour, real-valued clocks can be used. For the example shown in Figure 2, each higher priority node only needs to wait at most 30 time units before entering its critical section. This can be verified using the models in Figures 8-11 using the query \(E<>\left(P(1).\text{Waiting} \text{ and } P(1).x \geq 30\right)\) which is satisfied, and the query \(E<>\left(P(1).\text{Waiting} \text{ and } P(1).x > 30\right)\) which is not. Note that clock \(x\) is reset to 0 when entering the \texttt{Waiting} state. This case occurs when a low priority node is in its critical section when all three higher priority nodes request to enter. In general, starvation freedom is not satisfied by lower priority processes, so no such bound exists for low priority nodes unless there is sufficient delay between subsequent requests. Using simple worst-case response time analyses, it is easy to show that the worst-case response time before a given node is allowed to enter its critical section is found to be the worst-case blocking time caused by a lower priority node being in its critical section, followed by the maximum possible interference that results from all equal or higher priority nodes requesting at the same time. The models can be modified to include a minimum delay between subsequent requests. The results match those found using analytical models and simple response time analysis.

The performance of the algorithm depends on the topology of the logical structure. The best topology with respect to message complexity is the star topology, with one node in the center and all other nodes as leaf nodes. For the analysis, we define the diameter \(D\) of a logical structure to be the length of the longest path in the structure. As the logical structure evolves, the value of \(D\) may also change.

With a single priority level, the upper bound is equal to \((D+1)\) messages per critical section entry: \(D\) 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, using the straight line topology, the upper bound is \(N\), the number of nodes in the system. For the best topology, a star, the upper bound is 3, which is the same as a centralized mutual exclusion algorithm. To verify this upper bound, counters can be used, as shown above. Once a request is satisfied, the counter, count\[id\], is set back to zero. The model can be verified to determine the maximum number of messages required; e.g., \(E<>\left(\text{count}[2]==3\right)\) – on some path, eventually, does the count reach 3. For the star topology with \textit{edge reversal}, each leaf node can require up to 3 messages per critical section; thus, the property \(E<>\left(\text{count}[2]==3\right)\) is satisfied if node 2 is a leaf node, and the property \(E<>\left(\text{count}[2]==4\right)\) is never satisfied. Likewise, the central node requires at most two messages per critical section, and lower priority node 3 requires at most 5 messages. Not surprisingly, for the worst topology – a straight line – the worst-case for path reversal and the generalized algorithm is \(N\) messages, \(N-1\) \texttt{REQUEST} messages and one \texttt{TOKEN} message.

If there is more than one priority level, then the number of messages required at the highest priority level is only dependent on the number of nodes in the highest level with the same analysis as above. For lower priority messages, after the request reaches a sink node, by traversing at most the diameter of the
topology used for the given priority level, a proxy request may need to be forwarded up to the highest priority level, and to a sink node – again, at most the diameter of the nodes at the highest priority level. The token is passed to the lower priority node with a single message, and returned with a single message. Thus, the worst-case message complexity is $D_H + D_L + 2$ for low priority requests, where $D_H$ is the diameter of high priority nodes, and $D_L$ is the diameter of low priority nodes. If the low priority level consists of a single node, then $D_L = 0$. For the star topology, and the proxy node set as the root, at most 3 messages are required per critical section for both low and high priority nodes.

5 Summary

This paper presented a generalized, prioritized, token-based algorithm for distributed mutual exclusion. In the generalized algorithm, requests from lower priority nodes can be processed in different ways – either by passing each low priority request up to a sink at the highest priority level or by only passing low priority requests up to some other requesting node.

The algorithm imposes very little storage overhead on each node and in each message. Furthermore, the algorithm generalizes several existing token-based algorithms and can be extended for prioritized, real-time systems. Using the best topology and edge reversal, the algorithm attains comparable performance to a centralized mutual exclusion algorithm; that is, three messages per critical section entry. In the average case, the algorithm attains the best performance of any known algorithm. Real-time performance can be determined using verification of the models and analytically using response time analysis.

References


