Design of Mutual Exclusion Algorithms
for Real-Time Distributed Systems

Ye-In Chang
Dept. of Applied Mathematics
National Sun Yat-Sen University
Kaohsiung, Taiwan
R.O.C.
{E-mail: changyi@math.nsysu.edu.tw}
{Tel: 886-7-5316171 (ext. 3710)}
{fax: 886-7-5319479}

Abstract

In the problem of mutual exclusion, concurrent access to a shared resource or the Critical Section (CS) must be synchronized such that at any time, only one process can access the CS. In a real-time distributed system and a system which uses priorities for scheduling, events for requesting the CS should be ordered on the basis of priorities of the processes (as first proposed in Goscinksi’s algorithm), rather than on the basis of the time when these events happened. In this paper, we show how to modify several well-known distributed mutual exclusion algorithms (which are on the basis of the time when these events happened) to be priority-based mutual exclusion algorithms for real-time distributed systems.

Key Words: Computer networks, critical section, distributed algorithms, distributed operating systems, mutual exclusion.

\footnote{This research was supported by the National Science Council of the Republic of China, Grant No. NSC-81-0408-E-110-508}
1 Introduction

A distributed system consists of a collection of geographically dispersed autonomous nodes connected by a communication network. The nodes have no shared memory and communicate with one another by passing messages. Moreover, message propagation time delay is finite but unpredictable. In this paper, we assume that the communication subsystem is reliable and nodes do not crash.

In the problem of mutual exclusion, concurrent access to a shared resource or the Critical Section (CS) must be synchronized such that at any time only one process can access the CS. The problem was first described and solved by Dijkstra by using a shared variable in a centralized system [6]. Over the last decade, many algorithms have been proposed to achieve mutual exclusion in distributed systems. Those algorithms can be classified into two classes: token-based and non-token-based. In token-based algorithms [2, 3, 12, 14, 16, 18, 20], only the process holding the token can execute the CS and makes the final decision on the next process to enter the CS. In non-token-based algorithms [1, 4, 10, 11, 15, 17, 19], a requesting process can execute the CS only after it has received permission from each member of a subset of processes in the system, and every process receiving a CS request message participates in making the final decision.

In token-based algorithms, a unique token is shared among the processes, and the possession of the token gives a process the authority to execute the CS. Singular existence of the token implies the existence of mutual exclusion in a distributed system. Among these token-based algorithms, Suzuki and Kasami’s algorithm [20] requires 0 or N messages, where N is the number of nodes in the systems. Nishio et al.’s algorithm [12], which extends Suzuki and Kasami’s algorithm to be fault-tolerant to a node failure and token loss, also requires 0 or N messages. Singhal’s heuristically-aided algorithm [18] and Chang et al.’s dynamic token-based algorithm [3], which make use of the dynamic system state to reduce the number of messages exchanged, require between 0 and N messages. Raymond’s tree-based algorithm [14] requires only $O(\log N)$ messages, since the average length of a tree is $O(\log N)$, where the root of the logical tree always holds the token. Chang et al.’s algorithm [2], which extends Raymond’s algorithm to be fault-tolerant to node failures and communication failures, also requires $O(\log N)$ messages.
In non-token-based algorithms, a request set at a process X is used to record the identifiers of processes to which process X sends CS request messages when requesting the CS. A process invoking mutual exclusion can enter the CS only after it has received permission from all processes whose identifiers are in its request set. Among these non-token-based algorithms, Lamport’s algorithm [10], which is the first algorithm proposed for distributed mutual exclusion, requires $3(N - 1)$ messages. Ricart and Agrawala’s algorithm [15] has improved Lamport’s algorithm by deferring Reply messages and discarding the use of Release messages, which requires $2(N - 1)$ messages. Singhal’s dynamic information structure algorithm [19] and Carvalho and Roucairol’s algorithm [1] have improved Ricart and Agrawala’s algorithm by reducing the number of required Request and Reply messages, which require between 0 and $2(N - 1)$ messages. Maekawa’s algorithm [4, 11] requires only $O(\sqrt{N})$ messages, but needs longer time delay in successive executions of CS as compared to [1, 10, 15, 19]. A generalized Maekawa-type algorithm with several variants of initialization of request sets has been discussed by Sanders [17].

In a real-time distributed system and a system which uses priorities for scheduling, events should be ordered on the basis of priorities of the processes, rather than on the basis of the time when these events happened. (Note that in a real-time system, every process has strict time constraint. Processes must meet their deadlines; otherwise, the process will fail. Therefore, the remaining time to run the process defines the priority of the process. The larger the remaining time is, the lower the priority of the process has.) However, those algorithms mentioned above grant the permission to enter the CS in a first-come-first-served (FIFO) manner, that is, they order the events on the basis of the time when these events happened. Goscinski has proposed a priority-based approach (combining with a token-based approach) to mutual exclusion in real-time distributed systems [7, 8, 9]. In Goscinski’s algorithm, two priority queues are used: one is used to store the incoming requests, the other one is received with the token. Requests are stored in those priority queues according to priorities. This priority-based approach requires 0 or N messages in a fully reliable network, and requires 0 or $(N + 1)$ messages in an unreliable network in which nodes may fail. However, in [5], Chang has pointed out several errors in Goscinski’s both algorithms, including the problem of starvation, the problem of out-of-date messages
The most difficult part in the design of mutual exclusion algorithms for a real-time distributed system and a system which uses priorities for scheduling, is how to avoid starvation, in addition to ensuring that events for requesting the CS should be ordered on the basis of priorities of the processes. In this paper, we show how to modify several well-known distributed mutual exclusion algorithms (which are on the basis of the time when these events happened) to be priority-based mutual exclusion algorithms for real-time distributed systems. We give general guidance to modify token-based, and non-token-based distributed mutual exclusion algorithms, respectively, to ensure that events for requesting the CS should be ordered on the basis of priorities of the processes. To avoid starvation, for different algorithms, we present different ways to achieve the goal, since every algorithm has different ways to determine who is the next one to enter the CS.

The rest of the paper is organized as follows. In the next section, we describe the system model. Sections 3 and 4 present how to modify several well-known token-based and non-token-based mutual exclusion algorithms, respectively, to be priority-based mutual exclusion algorithms. Finally, Section 5 contains concluding remarks.

2 The System Model

A distributed system consists of N nodes, uniquely numbered from 1 to N. Each node contains a process that makes a request to mutually exclusive access the CS. This request is communicated to other processes. Message propagation delay is finite but unpredictable. The communication network is assumed to be reliable (i.e., messages are neither lost nor duplicated and are transmitted error-free), and nodes do not crash. There is only one CS in the system, and any process currently in the CS will exit in finite time. Moreover, a process cannot issue another request until the current request is granted and the process itself exits the CS.

In a system which uses priorities for scheduling and a real-time distributed system, each process $i$ is characterized as follows: in the case of a system with priorities (called the P-system) by its priority $p(i)$, or in the case of a real-time system (called the RT-system) by remaining time $T(i)$ to run the process [7, 8, 9]. In the P-system, a priority
queue $Q$ is defined as follows: $Q = (i_1, i_2, ..., i_m)$, where $i_j$ indicates a process which wants to enter the CS, $i_1, i_2, ..., i_m$ is a permutation of the sequence of numbers $1, 2, ..., m$, such that, $m \leq N$, and defined by relation $\succ$ in the following way: $i_k \succ i_t$ if and only if for each $k < t$, $p(i_k) > p(i_t)$. In the RT-system, a priority queue $Q$ is defined as follows: $Q = ((i_1, T(i_1)), (i_2, T(i_2)), ..., (i_m, T(i_m)))$, where $i_j$ indicates a process which wants to enter the CS, $i_1, i_2, ..., i_m$ is a permutation of the sequence of numbers $1, 2, ..., m$, such that, $m \leq N$, and defined by relation $\succ\succ$ in the following way: $i_k \succ\succ i_t$ if and only if for each $k < t$, $T(i_k) < T(i_t)$. For both cases, if two processes have the same priority or remaining time, the process with the smaller value of index $i$ is placed first. Moreover, for a real-time distributed system, we assume that the time to finish the execution of a CS (denoted as CS-Time) is short enough such that there is no $T(i)$ which is smaller than CS-Time, since it can result in starvation if $T(i) <$ CS-Time. Furthermore, the system can apply a clock synchronization algorithm [10] to synchronize those individual physical clocks in the system periodically.

3 Token-Based Mutual Exclusion Algorithms for Real-Time Distributed Systems

In token-based algorithms, a unique token is shared among the processes, and the possession of the token gives a process the authority to execute the CS. Depending on the given network is logically fully connected or not fully connected, and how a request arrives at the process holding the token, different token-based algorithms have been proposed. To determine who is the next one to hold the token (i.e., to enter the CS), some of the algorithms use a queue included in the Token messages [3, 12, 20], some of the algorithms use an arbitration rule [16, 18], and some of the algorithms use a distributed queue structure [2, 14].

In this section, we describe how to modify several well-known token-based mutual exclusion algorithms for real-time distributed systems. In general, we include a priority to every Request message. We then try to maintain a "virtual" priority queue of requests in the system. Moreover, to avoid starvation, after a process exits the CS, we apply the aging strategy, which is commonly used for the problem of CPU scheduling with the priority scheduling algorithm in operating systems [13]. Aging is a technique of gradually increasing
the priorities of processes that wait in the system for a long time. However, since for each type of algorithms as mentioned above, they have different way to determine who is the next one to hold the token (i.e., the virtual queue of requests has been implemented in a different ways), we need different ways to implement the aging strategy.

3.1 A Modification to Suzuki and Kasami’s Algorithm

In Suzuki and Kasami’s algorithm [20], it uses a queue included in a Token message to determine the next one to enter the CS. (That is, the virtual queue of requests has been implemented by a physical queue.) When a process invokes mutual exclusion, it sends Request messages to all other processes. To detect an out-of-date Request message, a $\text{Seq}_s$ array (with N entries) at each process $s$ and a $\text{TSeq}$ array (with N entries) in a Token message are required (which are called the sequence number strategy). $\text{Seq}_s[i]$ at process $s$, $1 \leq i \leq N$, records the number of CS requests made by process $i$ as far as process $s$ knows; $\text{TSeq}[i]$ records the number of CS executions finished by process $i$. When a process $s$ invokes mutual exclusion, it increases its sequence number $\text{Seq}_s[s]$ by one and includes this sequence number in each Request message. Upon receiving process $s$’s Request($\text{SN} = \text{Seq}_s[s], s$) message, a process $y$ can tell whether this request is out-of-date by testing the following conditions: (1) if $\text{SN} > \text{Seq}_y[s]$, then this is a new Request message and process $y$ should update $\text{Seq}_y[s] = \text{SN} (= \text{Seq}_s[s])$; (2) otherwise, this is an old Request message and process $y$ should ignore it.

A FIFO (First-In-First-Out) queue $\text{TQ}$ included in a Token message contains those waiting requests. After finishing execution of the CS, process $z$ updates $\text{TSeq}[z] = \text{Seq}_z[z]$ and checks any new waiting requests by testing the following conditions ($1 \leq i \leq N$): (1) if $\text{Seq}_z[i] \leq \text{TSeq}[i]$, the value of $\text{Seq}_z[i]$ is out-of-date; (2) if $\text{Seq}_z[i] > \text{TSeq}[i]$, process $i$ has finished the ($\text{TSeq}[i]$)'th CS execution, and is requesting the ($\text{Seq}_z[i]$)'th ($= \text{Seq}_z[i]$) CS execution. These new waiting requests are added to $\text{TQ}$ provided these requests are not already in $\text{TQ}$. Process $z$ then sends a Token($\text{TQ}, \text{TSeq}$) message to the process whose identifier is the first entry of $\text{TQ}$.

To make the modified version of Suzuki and Kasami’s algorithm work for a real-time distributed system, every Request message includes a priority. That is, when a process $i$
wants to enter the CS, it sends a Request(Seqᵢ, i, p(i)) message (in the case of the P-system) or a Request(Seqᵢ, i, T(i)) message (in the case of the RT-system) to whom it should send. At each process i, a LTPᵢ array (with N entries) is used to record the request priority of each process. When a process i receives a non-out-of-date Request message from process s, it updates both Seqᵢ[s] and LTPᵢ[s] to the new values included in the Request message. Moreover, a priority queue TQ is used, instead of a FIFO queue. All requests arriving at this node are stored in the queue in an order according to relation > or relation >>.

To avoid starvation, we apply the aging strategy. In the case of the P-system, after process z holding the token exits the CS, it increases the priority p(i) of each request i in TQ by a constant t, say 1. (Here, we assume that there is an upper bound (= K) on the priority p(i) of every new request.) After that, new requests are added as described above. (Note that in the case of the RT-system, the clock routine must perform decrementing operations on appropriate fields of the queue TQ elements, that is, T(i) to maintain real-time consistency, when the process holding the token exits the CS. This decrementing operation is similar to the function of aging. Moreover, if T(i) ≤ 0, then process i’s request should be removed from TQ since the request has passed the due time, and a process can start its next CS request when the current one has passed the due time.) Process z then sends the Token(TQ, TSeq) message to the process whose request is the first entry of TQ. At this time, the information about those new updated priorities of requests is already in TQ and will be passed to the next token owner. The correctness of the algorithm is proved in Appendix A.

### 3.2 A Modification to Singhal’s Heuristically-Aided Algorithm

Singhal’s heuristically-aided algorithm [18] applies arbitration rules to determine the next one to enter the CS. It has improved Suzuki and Kasami’s algorithm by making use of state information, which is defined as the set of states of mutual exclusion processes in the system. A process can be in the state of “requesting the CS” (denoted as R), “executing the CS” (denoted as E), “not requesting the CS” (denoted as N), or “not requesting the CS but holding the token” (denoted as H). A process maintains a SV array (called state vector) to store the state of processes of the system. A TSV array (i.e., Token State Vector)
is included in a Token message to refresh the information in a process’s state vector. (Note that the virtual queue of requests has been implemented in terms of TSV, i.e., requests in the virtual queue are those entries \( i \) with \( \text{TSV}[i] = R \).)

In this algorithm, when a process invokes mutual exclusion, it uses a heuristic to guess which processes are likely holding the token and sends token request messages only to those processes rather than to all of the processes. The heuristic is as follows: All the processes for which the state vector \( \text{SV} \) entries are \( R \) (i.e., Requesting the CS) are in the probable set. As the algorithm is executed, the state information is kept updated such that the heuristic can work correctly and efficiently (i.e., a request will always arrive at a process holding the token). A sequence number strategy is used to detect an out-of-date message and to determine whether the entry in \( \text{SV} \) or in \( \text{TSV} \) is up-to-date.

When process \( s \) finishes execution of the CS, it updates \( \text{SV}_s[s] = N \) and \( \text{TSV}[s] = N \). Then, process \( s \) updates other entries in \( \text{TSV} \) by using update rules. The update rules essentially compare the state vectors at the the process performing updates with the token vectors to determine which one has more current information about the state of the processes and restore the out-dated entries of these vectors with more current ones. This involves comparing respective entries of \( \text{Seq}_s \) and \( \text{TSeq} \) at process \( s \). For example, if \( \text{Seq}_s[i] > \text{TSeq}[i] \), then process \( s \) (which performs the updates) has more current information about process \( i \); therefore, \( \text{TSV}[i] \) and \( \text{TSeq}[i] \) should be properly set. After that, if none of the entries in \( \text{SV}_s \) is in state \( R \), process \( s \) holds the token and sets \( \text{SV}_s[s] = H \). Otherwise, process \( s \) uses arbitration rules to determine which requesting process should get the token.

To make Singhal’s algorithm [18] work well in a real-time distributed system, we do the following modifications:

1. A priority is included in each Request message and a LTP array as introduced before is maintained at each process.

2. A one-dimension TTP array (with \( N \) entries) which records the newest priority of each process is included in the Token message. When a process \( x \) exits the CS, it will update either its local information (including the sequence number, the state information, and the priority information) or the information stored in the Token message to the newest one by comparing \( \text{Seq}_x[i] \) with \( \text{TSeq}[i] \), \( 1 \leq i \leq N \). If \( \text{Seq}_x[i] > \text{TSeq}[i] \), the process updates its local information and sends a Token message to refresh the information in the \( \text{TSV} \) entry of the requesting process.
TSeq[\textit{i}], (1 \leq i \leq N), the information stored in the Token message is updated to the one stored locally in process \textit{x}. If Seq[\textit{i}] < TSeq[\textit{i}], the information stored locally in process \textit{x} is updated to the one stored in the Token message. If Seq[\textit{i}] = TSeq[\textit{i}], the priority information is copied from the Token message to the one locally. After that, the local information and the information stored in the Token message are the same.

3. Next, process \textit{x} increases the priorities of those current new requests recorded locally by a certain constant, where a new request \textit{i} is indicated by SV[\textit{i}] = R. Moreover, it copies the new information about the priorities stored in LTPx to the TTP array included in the Token message, and then sends out the Token message to the next one which has the highest priority. In this way, we ensure that the priorities of those requests which are in the requesting position after process \textit{x} and are known by process \textit{x} at this time, have been increased. Therefore, those requests will enter the CS in finite time.

The proof of the starvation-free property of the modified version of Singhal’s algorithm can be described briefly as follows. In the system, the virtual queue of requests has been implemented in terms of TSV. In the modified version, we do not change the way to detect new requests; therefore, all requests are correctly recorded in TSV. We only change the priorities of those new requests, and then inform the next token owner this change by including TSV and TTP arrays in the Token message. The next token owner can correctly update its local information by comparing its Seq with TSeq as described in item 2.

### 3.3 A Modification to Raymond’s Algorithm

Raymond’s algorithm [14] uses a distributed queue data structure to determine the next one to enter the CS, where a process’s request is stored only at one of its neighbor’s local queue, instead of at all other processes’ local queues. (That is, the virtual queue of requests has been implemented in terms of distributed physical local queues.) Therefore, the out-of-date Request message problem will not occur and sequence numbers are not needed.

In Raymond’s algorithm [14], the network topology is a tree and the root holds the token. Every process communicates only with its neighboring processes and holds informa-
tion only about its neighbors. At every process \(s\), a variable \(\text{Near}_s\) records the identifier of its neighbor on the path leading to the process holding the token, and a local FIFO queue \(Q_s\) records the identifiers of its requesting neighbors. (Note that when \(\text{Near}_s = s\), process \(s\) holds the token.)

When a process \(s\) which does not hold the token invokes mutual exclusion, it first adds its request to the end of \(Q_s\) and then sends a Request message to \(\text{Near}_s\) (provided it has not sent out a Request message for a waiting request in \(Q_s\)). When a process \(y\) which does not hold the token receives a Request message from one of its neighbors, it first adds the identifier of this neighbor to the end of \(Q_y\) and then sends a Request message to \(\text{Near}_y\) (provided it has not sent out a Request message for a waiting request in \(Q_y\)).

A sequence of Request messages are sent between the requesting process and the process holding the token (along the path constructed by \(\text{Near}\)'s) until a Request message arrives at the process holding the token. Then, the token is passed along the same path in the reverse direction. As the token passes through, the direction of the edges traveled by the token is reversed such that every path always leads to the process holding the token.

When a process \(x\) receives the token, it sends the token to the process whose identifier is the first entry of \(Q_x\), which is either itself or one of its requesting neighbors \(y\), and removes this request from \(Q_x\). In the case that the first waiting request is not process \(x\) itself and \(Q_x \neq \emptyset\), process \(x\) will send another Request message to its requesting neighbor \(y\) to ask for the return of the token.

To make Raymond's algorithm [14] work well in a real-time distributed system, in addition to replacing each local FIFO queue with a priority queue and including a priority to each Request message, more Request messages with different priorities from the same process may have to be sent out. In the modified version of Raymond's algorithm, When a process \(s\) which does not hold the token invokes mutual exclusion, it first adds its request to the priority queue \(Q_s\). Then, process \(s\) will send a Request\((s, p(s))\) message to \(\text{Near}_s\) in the following two cases:

1. \(|Q_s| = 1\), i.e., it is the only entry in \(Q_s\).
2. \(|Q_s| > 1\), and its priority is higher than any other request in \(Q_s\).
When a process \( y \) which does not hold the token receives a Request\((s, p(s))\) message from one of its neighbors, say \( s \), it first adds process \( s \)'s request to the priority queue \( Q_y \) if process \( s \)'s request is not in \( Q_y \). If process \( s \)'s request has already been in \( Q_y \), then process \( y \) updates process \( s \)'s priority in \( Q_y \) to this new priority. Then, process \( y \) will send a Request\((y, p(s))\) message to Near\(_y\) in the following two cases:

1. \(|Q_y| = 1\), i.e., process \( s \)'s request is the only entry in \( Q_y \).
2. \(|Q_y| > 1\), and process \( s \)'s priority is higher than any other request in \( Q_y \).

When process \( x \) receives the token, it sends the token to the process whose request is the first entry of the priority queue \( Q_x \), which is either itself or one of its requesting neighbors \( y \), and removes this request from \( Q_x \). In the case that the first waiting request is not process \( x \) itself and \( Q_x \neq \emptyset \), process \( x \) will send another Request\((x, p(t))\) message including the highest priority \( p(t) \) in \( Q_x \) to its requesting neighbor \( y \) to ask for the return of the token. (Note that to simplify the design of this algorithm, we assume that the pipelining property holds, i.e., between any pair of nodes, messages are received in the order in which they are sent; therefore, we can avoid the problem of message-overtaking as discussed in [14].) To improve the performance, we can apply a piggyback strategy. When a process sends out a Token message and then sends a Request message to ask for the return of the token, this Request message can be embedded in the Token message. If no Request message should be embedded in the Token message, a null message is included.

To apply the aging strategy (to avoid starvation) in the distributed queue, the following modifications are needed:

1. A counter \( CX \) which records the number of CS executions is included in the Token message. Before each request \((i, p(i))\) is added to a local queue, an additional integer variable \( \text{Flag}_i \) with an initial value \( = 0 \) is included to the request, which is used to decide the value of the priority that should be added when the token arrives. That is, the entry in the local queue has the form of \((i, p(i), \text{Flag}_i)\).

2. When a process \( x \) exits the CS, it increases the priority of each request in \( Q_x \) by one, and increases counter \( CX \) by one. After deleting one request (with identifier \( = y \)
from $Q_x$, if there is no more request in $Q_x$, process $x$ sends a Token($CX$, Null-Request) message to process $y$. If there is any request in $Q_x$, process $x$ sends a piggybacked Token($CX$, ($x$, $p(t)$)) message to process $y$, where $t$ is the highest priority in $Q_x$. Moreover, for each request $w$ in $Q_x$, process $x$ sets Flag$_x$ = $CX$.

3. When a process $x$ which holds the token and is not requesting the CS, receives a request, it will increases the priority of each request in $Q_x$ by one. It then sends a Token($CX$, Null-Request) message or a piggyback Token message to Near$_x$ as described above. Moreover, for each request $w$ in $Q_x$, process $x$ sets Flag$_x$ = $CX$.

4. When a process $y$ receives a Token message, first, for each request $w$ in $Q_y$, process $y$ sets $p(w) = p(w) + (CX - $Flag$_w$) and $Flag_w = CX$, if $Flag_w < > 0$, where CX is the new counter value included in the Received Token message. That is, process $y$ increases the priorities of those old requests $w$ in $Q_y$ by the additional number of CS executions after the token is sent out from process $y$ in the previous time, where the old requests means that those requests which were in $Q_y$ when process $y$ sent out the token. Those old requests can be distinguished from the new requests by the value of $Flag_w$, where the new requests means that those requests which are added to $Q_y$ after process $y$ sent out the token in the previous time. Note that when process $y$ sends out the token, it updates the value of $Flag_w$ of each request $w$ in $Q_y$ to CX; while a new request $z$ arrives at process $y$ after process $y$ sends out the token, the value of Flag$_z$ is set to 0. Moreover, for those old requests $w$, process $y$ now updates $Flag_w = CX$, i.e., it updates $Flag_w$ to the current value of CX and still uses the condition ($Flag_w > 0$) to indicate that those requests are old.

After deleting one request as the next token holder from $Q_y$, process $y$ adds the piggybacked request to $Q_y$ if any and uses the current value of CX as the value of Flag of this piggybacked request. The reason why we use Flag = CX instead of Flag = 0 for this piggybacked request is that we want to distinguish this request from those new requests.

If Next$_y = y$, process $y$ can enter the CS, it sets Flag$_w$ = 0 for each request $w$ in $Q_y$. At this moment, there is no need to distinguish old or new requests.
If Next\(_y\) \(!=\) \(y\), process \(y\) increases the priorities of those requests \(w\) in \(Q_y\) which have \(\text{Flag}_w = 0\) by one. That is, it increases the priorities of those new requests; for those old requests, including the piggybacked request, their priorities have been properly updated as described above. Process \(y\) then sends a Token(CX, Null-Request) message or a piggybacked Token message to Near\(_x\) as described above. Moreover, for each request \(w\) in \(Q_x\), process \(x\) sets \(\text{Flag}_x = \text{CX}\).

The proof of the starvation-free property of the modified version of Raymond’s algorithm can be described briefly as follows. When \(|Q_y| \neq 0\), the requests stored in \(Q_y\) can be divided into two classes: old requests which are in \(Q_y\) before process \(y\) sends out the token, and new requests which are added to \(Q_y\) after process \(y\) sends out the token. Old requests will have \(\text{Flag} > 0\) (i.e., Flag contains the value of \(\text{CX}\) when process \(y\) sends out the token) and new requests will have \(\text{Flag} = 0\). When process \(y\) receives the token again, for those old requests \(w\), it increases their priorities by \((\text{CX} - \text{Flag}_w)\), i.e., the additional number of CS execution finished during the time when process \(y\) sends out the token and the time when the token is returned to process \(y\) again. For the new requests, process \(y\) does not have to change their priorities at this moment. For the piggybacked request, process \(y\) also does not have to update its priority. When process \(y\) sends out the token again due to \(\text{Next}_y \neq y\), it increases the priority of each new request by one. In this way, we increase different values of priority to old/new requests. In fact, to avoid starvation, the key point is to keep increasing priorities of waiting requests; the way to increase the priorities of those waiting requests may not be fair enough due to the queue is distributed. However, this approach really avoids the case that new requests continue to monopolize all the chance to enter the CS to a certain degree.

### 3.4 An Example of the Modified Version of Raymond’s Algorithm

Figure 1 and Figure 2 show an example of the modified version of Raymond’s algorithm. In Figure 1-(a), processes A, B and D are requesting the CS with priorities 3, 1 and 2, respectively, and process C holds the token (with \(\text{CX} = 0\)) and is executing the CS. Processes A, B, and D have sent their requests to Near\(_A\) (\(= B\)), Near\(_B\) (\(= C\)) and Near\(_D\) (\(= C\)), respectively. When process C receives the token again due to Next\(_C \neq C\), process C increases the priority of each new request by one. In this way, we increase different values of priority to old/new requests. In fact, to avoid starvation, the key point is to keep increasing priorities of waiting requests; the way to increase the priorities of those waiting requests may not be fair enough due to the queue is distributed. However, this approach really avoids the case that new requests continue to monopolize all the chance to enter the CS to a certain degree.
respectively, and have added their requests to their local priority queues. In Figure 1-(b), these three Request messages have arrived at their destinations and then are added to the local priority queues of those message receivers. When process B receives the Request(A, 3) message from process A, it sends one more Request(B, 3) message to NearB since this new incoming request has a priority higher than any other one in QB. In Figure 1-(c), after process C has received the Request(B, 3) message, it changes the priority of process B in QC from 1 to 3 and rearranges the position of process B’s request in QC.

In Figure 1-(d), after process C has finished execution of the CS, it increases CX by one, increases the priority of each request in QC by one, and then sends the token to process B (which is the first entry in QC). Since \(|QC| \neq 0\), the token message contains the value of CX and a piggybacked request \((C, 3)\) to process B to ask for the return of the token, where 3 is the priority of the only request in QC. In Figure 1-(e), before process B sends the token to process A which is the first request in QB, it adds the piggybacked request \((C, 3)\) to QB. The priority of process B’s request is also increased by one since Flag\(_B\) = 0 before process B sends out the token. Moreover, process B sets the value of Flag in each request in QB to the same value of CX (= 1) before process B sends the token out. Process B then sends a piggybacked Token(1, (B, 3)) message to process A. In Figure 1-(f), process A receives the token, adds the piggybacked request to QA and then enters the CS.

In Figure 2-(g), process C invokes a CS request with a priority 3. After process A exits the CS, it increases CX by one and sends the token to process B. When process B receives the Token message with CX = 2, it updates the priority of each request \(w\) in QB if Flag\(_w\) \(\neq 0\). In Figure 2-(h), process B sends a piggybacked Token(2, (B, 3)) message to process C. When process C receives the Token message, it updates the priority of process D’s request since Flag\(_D\) \(\neq 0\), and sets Flag\(_D\) = CX.

In Figure 2-(i), process C adds the piggybacked request (B, 3) to QC, and then sends a piggybacked Token(2, (C, 4)) message to process D. In Figure 2-(j), process D receives the token, adds the piggybacked request to QD and then enters the CS. Figure 2-(k) shows the system state when process D exits the CS and process C then enters the CS. Finally, Figure 2-(l) shows the system state when process C exits the CS and then sends the token to process B.
Figure 1: An example of the modified version of Raymond’s algorithm: (a)-(f)
Figure 2: An example of the modified version of Raymond's algorithm: (g)-(l)
Note that in this example, the sequence to enter the CS is (C, A, D, C, B), and the algorithm does ensure the following properties which exist if there is a global queue included in the Token message:

1. Initially, we have $p(A) = 3$, $p(B) = 1$ and $p(D) = 2$ and process C holds the token. When process A holds the token, we have $p(B) = 2$ and $p(D) = 3$ as shown in Figure 1-(e).

2. When process D holds the token, we have $p(B) = 3$ as shown in Figure 2-(i). Since process C's request with priority = 3 arrives at the time when process B sends the token to process C as shown in Figure 2-(g), $p(C) = 4$ when process D holds the token.

3. When process C holds the token, we have $p(B) = 4$ as implied in Figure 2-(k).

4. Finally, process B holds the token.

The change of those priorities is not recorded explicitly at the requesting processes since the queue is distributed, but it has been updated appropriately by making use of variables Flag and CX.

4 Non-Token-Based Mutual Exclusion Algorithms for Real-Time Distributed Systems

In non-token-based algorithms, a process invoking mutual exclusion can enter the CS only after it has received permission from all the processes whose identifiers are in its request set. To simplify the design of a non-token-based algorithm, the following two assumptions are usually made: (1) the network topology is logically fully connected; (2) between any pair of nodes, messages are delivered in the order in which they are sent (i.e., the pipelining property holds); therefore, the out-of-date message problem will not occur. Depending on how a request set is formed, different non-token-based algorithms have been proposed. Moreover, among these algorithms, the contents of the queues stored locally are different. In some of the algorithms [10], each local queue will contain the same set of identifiers of
waiting requesting processes (in finite time), and those requests are stored in the order of logical timestamps. In some of the algorithms [1, 15, 19], every process \( x \) stores only the identifiers of those waiting requesting processes which have priorities lower than that of itself in the system if process \( x \) is requesting. (Note that if a process is not requesting the CS, its queue is empty.) In the algorithms of Maekawa’s type [4, 11, 17], every process \( x \) stores only the identifiers of part of those requesting processes which have priorities lower than that of a process \( y \) to which process \( x \) has given its permission (i.e., \( x \in R_y \), where \( R_y \) is the request set of process \( y \)).

In this section, we describe how to modify several well-known non-token-based mutual exclusion algorithms for real-time distributed systems. Most of the non-token-based algorithms use timestamps [10] (described in Section 4.1) to resolve conflicting requests (i.e., comparing timestamps to decide whether a Reply message should be sent out immediately), for examples, Ricart and Agrawala’s algorithm [15], Singhal’s dynamic information structure algorithm [19], Carvalho and Roucairol’s algorithm [1] and Maekawa’s algorithm [11]. For these algorithms, to ensure that every process enters the CS according to the priorities, we simply include a priority into each Request message and use these priorities to resolve conflicting requests, and replace each local FIFO queue with a priority queue. Timestamps are no longer needed in these algorithms. (However, timestamps have the other usage in Lamport’s algorithm and cannot be discarded; the details will be discussed in this section.)

Next, since the contents of the queues stored locally are different among these algorithms, for each type of algorithms based on the above classification, we present how to modify them such that the aging strategy can be implemented correctly to avoid starvation. (Note that since in the algorithms of Maekawa’s type, a process can only give its permission to a requesting process one at a time and stores other requests in its local queue, it is much difficult to implement the aging strategy completely. In fact, we can only implement the aging strategy in this type of algorithms to a limited degree. Therefore, we do not discuss this type of algorithms.)
4.1 A Modification to Lamport’s Algorithm

In Lamport’s algorithm [10], the requests for the resource are granted in the order in which they are made, instead of the order in which they arrive at a node. That is, in Lamport’s algorithm, each local queue will contain the same set of identifiers of waiting requesting processes (in finite time), and those requests are stored in the order of timestamps. (Note that different sets of waiting requests may exist in the local queues of different processes; but in finite time, they will become the same.) To order events in a distributed system without using a physical global clock, a timestamp strategy is proposed by Lamport.

This timestamp strategy assumes that events within a single process are totally ordered in time, and the event of “sending” a message occurs before the event of “receiving” the same message; therefore, it provides a partial ordering of events in the system, which is termed the happened-before relation. Lamport uses a logical clock $C$ to reflect this relation such that $(\forall a, b)$ if event $a$ happened before event $b$ then $C(a) < C(b)$. In other words, the following two conditions must hold: (1) For two events $a$ and $b$ in a node such that $a$ comes before $b$, $C(a) < C(b)$, and (2) if $a$ is the event of sending a message $m$ by node $x$, and $b$ is the receipt of $m$ by node $y$, then $C(x) < C(y)$. This logical clock can be implemented by using a counter $C_x$ at node $x$ as follows:

1. Node $x$ increases $C_x$ between any two successive events.

2. If $a$ denotes the event of sending a message $m$ by node $x$, then the message $m$ bears the timestamp $T_m = C_x(a)$.

3. When the message $m$ is received by node $y$, it sets $C_y$ to a value that is greater than or equal to its present value and greater than $T_m$.

To impose a total ordering on the collection of events in the distributed system, the following rules are used to define event $a$ in node $x$ precedes event $b$ in node $y$:

1. $C_x(a) < C_y(b)$, or

2. $C_x(a) = C_y(b)$, and $x < y$. 

18
In Lamport’s mutual exclusion algorithm, every message includes a timestamp T. Moreover, every process maintains a priority queue Q in which requests are ordered by the timestamps. When a process x invokes mutual exclusion, it adds its request to Q_x and sends a Request(T_x, x) message to every process. When a process y receives the Request message, it returns a timestamped Reply message and adds process x’s request to Q_y. Process x can enter the CS when the following two conditions are satisfied:

1. Process x’s own request is in the front of Q_x.

2. Process x has received a message from every process with a timestamp larger than (T_x, x).

Since the pipelining property holds, the second condition guarantees that process x has learned about all requests that preceded its current request. Based on the total order defined by the timestamps, the first condition will permit one and only one process (i.e., the process with the smallest timestamp among current requesting processes) to enter the CS.

To release the resource, process x removes its request from Q_x and sends timestamped Release messages to all other processes. When a process y receives process x’s Release message, it removes process x’s request from Q_y.

In Lamport’s algorithm, the timestamp is not used to resolve conflicting requests as used in [1, 11, 15, 18] since a Reply message will be immediately sent out when a Request message is received. A timestamp is used to detect that all other processes have already received process i’s request when a requesting process i has received a message from every process with a timestamp larger than its requesting timestamp. Therefore, in the modified version of Lamport’s algorithm for a real-time distributed system, every message still includes a timestamp T. A priority is included in each Request message. Moreover, every process maintains a priority queue Q in which requests are ordered by the priority, instead of by the timestamp.

To apply the aging strategy to avoid starvation, after process x exits the CS and removes its own request from Q_x, it increases the priority of each request in Q_x by a certain constant. Then, process x includes the copy of Q_x information into the timestamped Release
messages to all other processes. When process $y$ receives process $x$’s Release message, it removes process $x$’s request from $Q_x$. Moreover, for each entry in the copy of $Q_x$, process $y$ either updates the priority of this entry to the new one if this entry exists in $Q_y$, or appends the entry to its local buffer $B_y$ if this entry does not exist in $Q_y$ so far. (Note that the latter case occurs due to unpredictable message propagation delay. Moreover, in this case, if this entry already exists in $B_y$, process $y$ updates the priority to the larger one.) When a new request $z$ arrives at process $y$ and is stored in $Q_y$, process $y$ checks its local buffer $B_y$ to see whether process $z$’s request exists in $B_y$ or not. If process $z$’s request exists in $B_y$, process $y$ updates the priority of process $z$’s request to the one stored in $B_y$ and removes it from $B_y$.

The proof of the starvation-free property of the modified version of Lamport’s algorithm can be described briefly as follows. In finite time, each local queue will contain the same set of requests. Therefore, we can simply inform all the processes the change of the priorities by Release messages. However, due to unpredictable message propagation delay, some local queues may miss some requests which are already recorded in the current token owner. To make up this miss, a local buffer $B$ is used.

### 4.2 An Example of the Modified Version of Lamport’s Algorithm

Figure 3 shows an example of the modified version of Lamport’s algorithm. In Figure 3-(a), processes A and B are requesting the CS with $T_A = 3$, $p(A) = 8$, $T_B = 1$, and $p(B) = 2$. Process A has received the request from process B, but the request from process A has not arrived at process B. Both requests have arrived at process C. The largest timestamp information from each process has been recorded at each process. At this moment, $DQ(Q_A) = A$, where $DQ$ is an operation which deletes one entry from a queue and returns the entry, process A has satisfied the first condition to enter the CS; however, process A has not received a message with a timestamp larger than its own requesting timestamp from each process (i.e., the second condition to enter the CS is not satisfied). Therefore, process A can not enter the CS now, so does the case of process B.

In Figure 3-(b), process B has received the request from process A; the contents of
Figure 3: An example of the modified version of Lamport’s algorithm
the local queues at all the processes are the same at this moment. Process C has sent out its Reply messages to processes A and B. Process A also has sent out its Reply message to process B. In this case, \( DQ(Q_A) = DQ(Q_B) = A \), but process A has not satisfied the second condition to enter the CS; therefore, no one can enter the CS. (Note that for this example in the original Lamport’s algorithm, at this time, process B can enter the CS, since the requests in the local priority queue are ordered by timestamps, \( DQ(Q_A) = DQ(Q_B) = B \) and process B has satisfied the second condition to enter the CS. Moreover, for this example in the modified version of Lamport’s algorithm, if the pipelining property does not hold, (i.e., process A’s Reply message with timestamp 4 arrives early than process A’s Request message at process B), process B will enter the CS since \( DQ(Q_B) = B, DQ(Q_A) = A \) but process B has satisfied the second condition to enter the CS.)

In Figure 3-(c), processes A and B have received the Reply message sent from each other. At this moment, process A has satisfied the second condition to enter the CS and \( DQ(Q_A) = A \); process A enters the CS. Process C starts to send out its request with \( T_C = 5 \) and \( p(C) = 3 \). In Figure 3-(d), process A exits the CS, removes its own request from \( Q_A \), and sends Release messages to processes B and C. Note that the priority of process B in \( Q_A \) has been increased by one and is included in each Release message sent out by process A. When process B received this Release message, it updates the priority of itself in \( Q_B \) to the same value as the one included in the Release message, so does process C. At this time, the content of \( Q_C \) has been changed as compared to the one shown in Figure 3-(c). (Note that if two processes have the same priority or remaining time, the process with the smaller value of index \( i \) is placed in the priority queue first.) Process B can enter the CS now. In Figure 3-(e), process C’s request has arrived at processes A and B, and then receives Reply messages from processes A and B; however, \( DQ(Q_C) \neq C \), process C cannot enter the CS. (Note that the Reply message sent from process A to process C is not important to process C at this time since the largest timestamp sent from process A (\( = 6 \)) in Figure 3-(d) has been already larger than the timestamp of process C’s request (\( = 5 \)).) Finally, after process B exits the CS and process C has received a Release message from process B, process C enters the CS.
4.3 A Modification to Ricart and Agrawala’s Algorithm

In Ricart and Agrawala’s algorithm [15], at each requesting process \( x \), the content of the local queue contains only the identifiers of those waiting requests which have priorities lower than that of process \( x \). Ricart and Agrawala’s algorithm has reduced the message traffic in Lamport’s algorithm [10] by using an implicit Release message strategy. In this algorithm, a process \( x \) can defer its reply to any other process \( y \) which has a priority lower than that of process \( x \) (by comparing timestamps included in the Request messages) until process \( x \) finishes execution of the CS. Therefore, when a process \( y \) receives a Reply message from a process \( x \), the Reply message implies that process \( x \) has finished execution of the CS and no Release message is needed.

To make this algorithm work well for a real-time distributed system, in addition to including a priority to each Request message and using the priorities to resolve conflicting requests, we have to do the following modifications to avoid starvation:

1. When process \( x \) exits the CS, it increases the priority of each request in \( Q_x \) by a certain constant. Then, process \( x \) includes the copy of \( Q_x \) information into the Reply message to those processes whose identifiers are in \( Q_x \).

2. When process \( y \) receives process \( x \)’s Reply message, for each request \( w \) in the copy of \( Q_x \), if the priority of process \( w \) is higher than that of process \( y \), process \( y \) removes this request \( w \) from the copy of \( Q_x \). (Note that in this case, whether process \( w \)’s request has arrived at process \( y \) or not is not important since process \( y \) will not defer its Reply message to process \( w \), i.e., it will not add process \( w \)’s request to \( Q_y \).) Next, as described in Section 4.1, for each request \( w \) in the copy of \( Q_x \), process \( y \) either updates the priority of request \( w \) in \( Q_y \) to the new one if this request exists in \( Q_y \), or appends the request to its local buffer \( B_y \) if this request does not exist in \( Q_y \) so far.

The proof of the starvation-free property of the modified version of Ricart and Agrawala’s algorithm can be described briefly as follows. Since at each requesting process \( x \), the content of the local queue contains only those requests which have priorities lower than that of process \( x \), process \( x \) only has to keep update new priorities of those requests which are
recorded in the process that just finished the execution of CS and have priorities lower than that of process \( x \).

Carvalho and Roucairol’s algorithm [1] and Singhal’s dynamic information structure algorithm [19] have improved Ricart and Agrawala’s algorithm by reducing the size of request sets. The content of the local queue at each process \( x \) also contains only the identifiers of those waiting requests which have priorities lower than that of process \( x \) as in Ricart and Agrawala’s algorithm. Therefore, we can apply the same modifications to these two algorithms [1, 19] to make them work well for a real-time distributed system.

5 Conclusions

In a real-time distributed system and a system which uses priorities for scheduling, events should be ordered on the basis of priorities of the processes, rather than on the basis of the time when these events happened. In this paper, we have presented how to modify several well-known distributed mutual exclusion algorithms which are on the basis of the time when these events happened to work well for real-time distributed systems. The most difficult part in these modifications is how to avoid starvation, in addition to ensuring that events for requesting the CS should be ordered on the basis of priorities of the processes. Basically, we apply the aging strategy to avoid starvation. However, depending on different ways to determine the next process to enter the CS, different implementations of the aging strategy are needed. While in some algorithms, the aging strategy can be implemented perfectly, like the modified version of Suzuki and Kasami’s algorithm, since this algorithm includes a queue included in the Token message passing around the processes; in some algorithms, the aging strategy can only be implemented to a certain degree. However, in all the cases, we still ensure that there is no starvation in the system.

In fact, several problems arise when mutual exclusion is required in a real-time distributed system. For example, time-out mechanism to detect token loss (or request loss) may not be useful in a real-time distributed system, since requests are granted in an order of priorities, instead of a FIFO order. Moreover, the assumption that one process cannot issue the next CS request until the current one has been finished (which is usually made in distributed mutual exclusion algorithms), may have to be removed, since in a real-time
distributed system, a process should have the right to issue a new CS request with a higher priority than the current one. These are the future research directions in designing mutual exclusion algorithms for real-time distributed systems.

References


Appendix A: Correctness of the Modification to Suzuki and Kasami’s Algorithm

In the modified version of Suzuki and Kasami’s algorithm, what we have changed is the order to enter the CS; therefore, the modification will not affect the achievement of the properties of mutual exclusion and freedom of deadlock. Hence, to prove the correctness of the modified algorithm, we only have to prove that the algorithm is starvation free. Starvation occurs when few processes repeatedly execute the CS while other processes wait indefinitely for their turns to do so. Since the algorithm is free from deadlock, a request will arrive at a process holding the token; therefore, no request is lost, i.e., every new request is recorded in the priority queue TQ in finite time.

In the case of the P-system, the requests are stored in TQ according to the relation > as defined before, i.e., TQ = (i_1, i_2, ..., i_m) and i_k > i_t if and only if for each k < t, p(i_k) > p(i_t). Assume that the last entry in TQ is a request issued by process i_m with priority p(i_m) = W, and there are (m - 1) requests with priorities higher than p(i_m). Before the process z holding the token adds new requests to TQ, process z increases the priority of each entry in TQ by a constant t, say 1. Therefore, the priority of process i_m is updated to P(i_m) = W + 1. Then, new requests are added to TQ by process z. Three cases occur among those new requests:

1. There is a new request with a priority lower than W.
2. There is a new request with a priority equal to W.
3. There is a new request with a priority higher than W.

In case 1, the new request will be added to TQ in the position after the entry i_m. In case 2, the new request will also be added to TQ in the position after the entry i_m since the priority of process i_m has been updated to W + 1. In both cases, process i_m’s request will be served before the new request. In case 3, the new request will be added to TQ in the position before the entry i_m, where we ignore the case that the request has the same priority as (W + 1) and is issued by a process with its identifier (or index) larger than i_m since this case is similar to cases 1 and 2. (Note that if two processes have the same
priority or remaining time, the process with the smaller value of index $i$ is placed first.) Assume that there are $X_{m-1}$ processes with the request types equal to case 3. Therefore, at this moment, process $i_m$ must wait for $(m - 1 + X_{m-1})$ processes to enter the CS before it can enter the CS. After the next process exits the CS, the priority of process $i_m$ has been updated to $(W + 2)$. When new requests are added again, there are $(m - 2 + X_{m-1} + X_{m-2})$ more processes before process $i_m$ to enter the CS, where $X_{m-2}$ denotes the number of new requests in case 3 at this time.

After $(m - 1)$ times of CS execution have been finished, $p(i_m)$ is updated to $(W + m - 1)$, and there are $\sum_{i=1}^{m-1} X_i (< N)$ processes before process $i_m$ can enter the CS. (Note that a process cannot issue the next request until the current one is finished; therefore, there are at most N requests in the system.) In the worst case, after $(K + 1) - (W + m - 1)$ more processes have entered the CS, the priority of process $i_m$ has been updated to $(K + 1)$, which is higher than the upper bound of the priority of any new request. In this case, there is no more process which can have a priority higher than that of process $i_m$ except those processes which are already in TQ; however, in the exception case, the number of processes is finite and cannot be increased, since any new request can have a priority no higher than $K$. Therefore, process $i_m$ will enter the CS in finite time.

In the case of the RT-system, the requests are stored in TQ according to the relation $>>$, where $TQ = ((i_1, T(i_1)), (i_2, T(i_2)), ..., (i_m, T(i_m)))$ and $i_k >> i_t$ if and only if for each $k < t$, $T(i_k) < T(i_t)$. Like the P-system, there is a low bound on the priority of $T(i)$. After a process exits the CS, $T(i)$ will be decreased by a certain value. In a finite time, $T(i_m)$ of a process $i_m$ will be decreased to a value which is still larger than 0 (and the request from process $i_m$ may be moved to the front of TQ), or will be decreased to 0 finally and discarded from the system. In the latter case, the request cannot be finished in the required time; therefore, the request is discarded. The process which issues this request can start to issue a new request after its local clock has passed the period of $T(i_m)$. That is, the process will not wait forever for its turn to enter the CS. Note that since there is no global clock in the system, there may be the case in which the request $i_m$ is still in TQ while process $i_m$ has issued another request. Since a new request will be added to TQ only if it is not already in TQ; therefore, in this case, the new request will be discarded. Although in this case of

28
the RT-system, a request may be discarded due to passing the due time, this case agrees with the requirement of the real-time system and is not considered as starvation.

In both systems, every process which issues mutual exclusion will enter the CS in finite time. Therefore, the modified version of Suzuki and Kasami’s algorithm is free from starvation.