An Efficient Distributed Mutual Exclusion Algorithm Based on Relative Consensus Voting

Jiannong Cao
Hong Kong Polytechnic University
Kowloon, Hong Kong
csjcao@comp.polyu.edu.hk

Jingyang Zhou
Nanjing University
Nanjing, China
jingyang@nju.edu.cn

Daoxu Chen
Nanjing University
Nanjing, China
cdx@nju.edu.cn

Jie Wu
Florida Atlantic University
Boca Raton, USA
jie@cse.fau.edu

Abstract

Many algorithms for achieving mutual exclusion in distributed computing systems have been proposed. The three most often used performance measures are the number of messages exchanged between the nodes per Critical Section (CS) execution, the response time, and the synchronization delay. In this paper, we present a new fully distributed mutual exclusion algorithm. A node requesting the CS sends out the request message which will roam in the network. The message will be forwarded among the nodes until the requesting node obtains enough permissions to decide its order to enter the CS. The decision is made by using Relative Consensus Voting (RCV), which is a variation of the well-known Majority Consensus Voting (MCV) scheme. Unlike existing algorithms which determine the node to enter the CS one by one, in our algorithm, several nodes can be decided and ordered for executing the CS. The synchronization delay is minimal. Although the message complexity can be up to $O(N)$ in the worst case in a system with N nodes, our simulation results show that, on average, the algorithm needs less number of messages and has less response time than most of those existing algorithms which do not require a logical topology imposed on the nodes. This is especially true when the system is under heavy demand. Another feature of the proposed algorithm is that it does not require the FIFO property of the underlying message passing mechanism.

1. Introduction

Solving the mutual exclusion problem in a distributed system imposes more challenges than in a centralized system. The mutual exclusion problem states that to enter a Critical Section (CS), a process must first obtain the lock for it and ensure that no other processes enter the same CS at the same time. When competing processes are distributed on the nodes over a network, how to achieve mutual exclusion efficiently still remains a difficult problem to solve in distributed systems. Over the last two decades, many algorithms for mutual exclusion in distributed computing systems have been proposed. Three performance measures are often used to evaluate their performance. They are *message complexity*, *response time* and *synchronization delay* [16]. The message complexity is measured in terms of the number of messages exchanged between the nodes per CS execution. The response time is the time interval a request waits for its CS execution to be over after its request messages have been sent out. The synchronization delay is the time interval between two successive executions of the CS. The response time and synchronization delay both reveal how soon a requesting node can enter the CS and are measured in terms of the average message propagation delay T_n.

Distributed mutual exclusion algorithms can be divided into two categories: structured and non-structured. Structured algorithms impose some logical topologies, such as tree, ring and star, on the nodes in the system. These algorithms usually have good message complexity when the load is "heavy", i.e., there is always a pending request for mutual exclusion in the system. For example, Raymond's tree-based algorithm [12] requires only 4 messages exchanged per CS execution at heavy loads. However, these algorithms increase average response time delay as high as $O(\log(N))$. Meanwhile, the organization and maintenance of the specified topology also lead to a large overload. Furthermore, most structured algorithms work well only under their specified topologies, and may be inefficient in some other environments [20]. In this paper, we are concerned with non-structured algorithms which are generic in the sense that they are suitable for arbitrary network topologies.

For non-structured algorithms, the message complexity can be as low as $O(\sqrt{N})$ or $O(\log(N))$. The response time can be $2T_n$ at light loads and $N*(T_n+T_c)$ at heavy loads, where T_c is the average CS execution time. But either the reduction of the message complexity is achieved at the cost of long synchronization delay or the decrease in response time is gained at the cost of high message complexity. In other words, they either cause high message complexity or result in long response time. More importantly, most of the algorithms require the FIFO (First In First Out) property as prerequisite for the underlying message passing communications. If this property can not be satisfied, extra messages or mechanisms needed to be employed to solve possible deadlock [5].

In this paper, we present a novel non-structured algorithm that can solve distributed mutual exclusion efficiently and resiliently. A node requesting the CS sends out the request message which will roam in the network. The message will be forwarded among the nodes until the requesting node obtains enough permissions to decide its order to enter the CS. The decision is made by using Relative Consensus Voting (RCV), which is a variation of the well-known Majority Consensus Voting (MCV) scheme [18]. In RCV, the request of a node can be granted if it either can eventually obtain the largest number of permissions against other currently competing requests, or the node has the smallest id among the requesting nodes potentially with the same number of permissions. Since nodes are not always required to collect permissions from the majority of all the nodes in the system, the number of messages exchanged can be reduced.

The proposed algorithm requires no pre-configuration on the system but only needs to know the total number of the network nodes that are involved. It possesses several other advantages. First, it does not require the FIFO property of the underlying message passing mechanism. Even when messages are delivered out of order, there is no impact on the algorithm's correctness and performance. Second, unlike existing algorithms which determine the node to enter the CS one by one, in our algorithm, several nodes can be decided and ordered for executing the CS so that the delay time before entering the CS can be reduced. The algorithm generates a sequence of requesting nodes that describes their order to execute the CS. Each node executes the CS directly if it stands on the top the sequence or waits for a message from its immediate preceding node in the sequence informing it to enter the CS, so the synchronization delay is minimal, i.e., T (T is the average delay of passing a message between two nodes). Another advantage introduced by the RCV scheme is resiliency which is inherited from the MCV. Since the correct operation of the algorithm does not depend on any specific node, crash of nodes will not affect the algorithm's execution. Although the message complexity can be up to $O(N)$ in the worst case, our simulation results show that, on average, the algorithm needs less number of messages and has less response time than most of those existing algorithms which do not require a logical topology imposed on the nodes. This is especially true when the system is under heavy demand. We argue that performance of distributed mutual exclusion algorithms under light load is not as critical as under the heavy loads, because system resources are rich under light load, thus algorithms with higher overhead can work well.

The remainder of this paper is organized as follows: Section 2 overviews related work. Section 3 describes our system model. In Section 4, we present the design of the proposed algorithm. Sections 5 and 6 contain the correctness proof of and performance evaluation of the proposed algorithm, respectively. Finally we conclude the paper in Section 7.

2. Background and related works

Some of the non-structured algorithms employ a logical token to achieve mutual exclusion [3, 14, 17]. In the token-based algorithms, a unique token is shared among the nodes and only the node which possesses the token is able to enter the CS. The most representative algorithm that uses token is *broadcast* [17]: a requesting node sends token requests to all other nodes and the token holder then passes the token to the requesting node after it finishing executing the CS or it no longer need the token. An optimization on the broadcast is that a node only sends its token requests to nodes that either has the token or is going to get it in near future [14] so that the number of messages exchanged per CS execution can be reduced from *N* to *N/2* on the average at light loads, and the response time keeps $2T_n$ at light loads and $N*(T_n+T_c)$ at heavy loads. [3] proposes an interesting algorithm where the token contains an ordered list of all requesting nodes that have been determined the order to enter the CS. The messages needed to exchange per CS execution is *3-2/N* at heavy load. But when calculating the response time, an extra "request collect time" must be considered. Another drawback of the algorithm is that it is not a fully distributed algorithm because at any time, there is an "arbiter" acting as the coordinator in the system. In addition, it is difficult for token-enabled algorithms to detect loss of the token and regenerate a new unique one. Although some efforts have been made to tackle this problem [2, 6, 10], solutions always induce extra high overloads.

For algorithms without using token, usually several rounds of message exchanges among the nodes are required to obtain the permission for a node to enter the CS. Lamport's logical timestamp [7] is often adopted in this type of non token-based algorithms. Ricart and Agrawala proposed an algorithm [13] as an optimization of Lamport's algorithm. In their algorithm, a node grants multiple permissions to requesting nodes immediately if it is not requesting the CS or its own request has lower priority. Otherwise, it defers granting permission until its execution in the CS is over. Only after receiving grants from all other nodes, can the requesting node enter the CS. Ricart-Agrawala's mutual exclusion algorithm has low delays because of parallelism in transfer of messages. The response time is $2T_n$ and $N*(T_n+T_c)$ under light load and heavy load, respectively. But the number of messages exchanged per CS execution is a constant of *2*(N-1)*, which is quite large. Under light load, the average number of messages can be reduced to *N-1* by using a dynamic algorithm [15]. A more recent work described in [8] reduces message traffic of the Ricart-Agrawala type algorithms to somewhere between *N-1* and *2(N-1)* by making use of the concurrency of requests and some other methods. Nevertheless, the message complexity remains $O(N)$.

Another type of non-structured algorithms that does not need to use token is the quorum-based algorithms. A quorum is a set of nodes associated with each node in the system and every two quorums have a nonempty intersection. The commonality of quorum-based algorithms lies in that a requesting node can enter the CS with permissions from only the nodes in its quorum. Obviously, messages needed to be exchanged are decided by the size of the quorum. A well known example is Maekawa's algorithm [9], where nodes issue permission only to one request at a time and a requesting node is only needed to receive permissions from all members of its quorum before it is able to enter the CS. In [9], the quorum size is $\sqrt{N}$ while in the Rangarajan-Setia-Tripathi algorithm [11], the size is reduced to $\frac{G+1}{2}\sqrt{\frac{N}{G}}$, where G is the subgroup size. [1] organizes all *N* nodes to a binary tree and a quorum is formed by including all nodes along any path that starts from the root of the spanning tree and terminates at a leaf. So the quorum size is log*(N)* in the best case and *(N+1)/2* in the worst case. However, the algorithm will degenerate to a centralized algorithm because the root node is included in all quorums when it is always available.

As to the response time, it is comparatively high under heavy load in Maekawa's algorithm because the synchronization delay is $2T_n$. Some improvements have been made to the Maekawa type algorithms [4, 19] by introducing more types of messages and exchanging a few more messages so that the synchronization delay can be reduced to T_n. Despite its good performance, the quorum-based mutual exclusion algorithms still have two disadvantages. First, the overhead of generating quorum for each node must be taken into account especially when the number of network nodes tends to change dynamically. Second, if the FIFO property can not be satisfied, which means that messages between two nodes are not always delivered in the same order as being sent, extra mechanism should be employed to avoid possible deadlock, and when conflicts occur frequently, more than *N* messages may need to be exchanged [5].

3. System model and data structures

A distributed system consists of *N* nodes that are numbered from N_0 to N_{N-1}. The term *node* used here refers to a process as well as the computer on which the process is executing. There is no shared memory or global clock and the nodes communicate with each other only through message passing. In this paper, we do not consider fault tolerance issues. We assume that the nodes do not crash underlying communication medium is reliable so that the messages will not be lost, duplicated.

It is assumed that each node can issue a request for entering the CS only when there is no outstanding request issued from the same node. Figure 1 shows the structure of a node. On each node, a MPM (Message Processing Model) is deployed. It processes messages cached in the Incoming Message Queue of that node and sends messages to other nodes when necessary. Also, every node maintains a table recording the system information (SI). Figure 2 illustrates the data structure used for SI. It contains three fields:

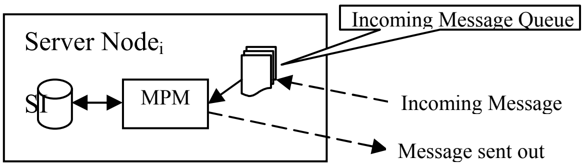

Figure 1. Node Structure

- *Next* indicates which node, if any, will enter the CS immediately after this node.
- *NONL* (Node Ordered Node List) is a sequence of ordered tuples. A tuple, in the form of < *NodeID*, *TS* >, records the requesting node's ID and the timestamp at which moment the corresponding request message was firstly initialized.
- *NSIT* (Node System Information Table) consists of *N* rows, one for each node in the system (including the node itself). Each row records the information about a node known to it, including the *ID*, the timestamp *TS,* and a tuple list *MNL* of that node. *MNL* is a list of tuples like <*NodeID*, *TS*>, showing all the nodes from which a request message has been received. *TS* represents how up-to-date the information about the node is. Since the status of the node's information is updated whenever the node issues a request message or receives a request message, *TS* is implemented as a counter recording the number of request messages that have been initialized at or sent to the node.

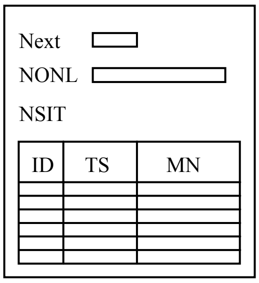

Figure 2. Data structure of SI

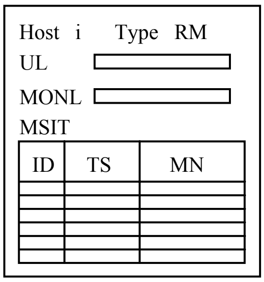

Figure 3. RM initialized by Node$_i$

In the remaining part of the paper, we denote a node with ID "i" as N_i, and the SI maintained by N_i as SI_i.

Only three types of messages are employed in our proposed algorithm. They are:

- Request Message (RM)
- Enter Message (EM)
- Inform Message (IM)

Since a RM message can be forwarded by different nodes, we call the node that initially sends the message the *home node* of the message. Each message is associated with a flag which indicates the type of the message. Data structures contained in messages are similar to that used for *SI*. As an example, figure 3 shows the data structure used in RM. "*MONL"* (Message Ordered Node List) is a sequence of ordered tuples. "*MSIT"* (Message System Information Table) records the newest system information updated during the roaming of the message. In addition, a field *Host* indicates the home node of the message. "*UL*" records unvisited nodes' ID.

The EM and IM messages do not have the *UL* and *Host* fields. IM messages have a field "*Next*" recording the id of the node that will enter the CS after the message's destination node.

4. The algorithm

When a node wants to enter the CS, it initializes a RM message and sends it to some other node. As described before, the field *MNL* in the data structure maintained by a node records all the nodes that have sent RM to the node. When a RM message is processed by the MPM on a node, a tuple is generated and appended to the *MNL*. After exchanging with the information carried in the incoming messages (using the *Exchange* procedure), the MPM will calculate whether the RM message's home node has gained enough information to determine its rank among all the competing nodes (using the *Order* procedure). If not, the RM will be forwarded to other nodes that the message has not visited. Otherwise, if the rank can be determined, we say that the requesting node, or its RM, or the corresponding tuple is ordered. An ordered tuple knows the order for its home node to enter the CS and will not be forwarded among the nodes. If the MPM finds that a tuple has the highest rank, it immediately sends an EM message to the tuple's home node. If the tuple hasn't the highest rank, its immediate preceding tuple's home node will be informed by an IM the immediate next node to execute the CS. After a node finishes executing the CS, it will send an EM message to its successor.

In the following subsections, we will describe the algorithm executed by MPM, the Exchange procedure, and the Order procedure.

4.1. The MPM Algorithm

Once a node N_i wants to enter a CS, it increases its timestamp t_i by one, and appends the tuple *<i,* t_i*>* to *SI_i.NSIT[i].MNL* (Line 4, 5). After doing so, it generates a RM message and sends it out for roaming over the network to confer with other nodes on its order of entering the CS. The RM message is initialized with a partial SI copy of the home node (Line 6-13).

If node N_i has been ordered, it will receive an EM message when it is on top of the *Ordered Nodes List*, or its immediate preceding node "*k*" in the list will receive an IM informing it to update the *Next* field to "i". When on top of the Ordered Node List, N_i will either receive an EM from N_k which just exits the CS or from N_j where its order is determined. On getting enough permissions to enter the CS, N_i will first invoke the *Exchange* procedure to update its SI with the incoming EM and then enter the CS (Line 14-16).

Whenever finishing executing the CS, N_i must send an EM to the node represented by *$Next_i$*, if any, and delete its own tuple from the top of *SI_i.NONL* (Line 17-24).

At times N_i will receive an IM indicating that N_j is the next node to it that enters CS. If N_i hasn't entered the CS or is currently executing the CS (this can be determined by whether tuple <i, t_i> is still in *SI_i.NONL*), the only thing left to do reset the value *SI_i.Next* to "j". Otherwise, which means that N_i has finished executing CS, N_i should generate an EM with a copy of its SI and send it to N_j immediately (Line 25-32).

Upon receiving a RM originally initialized in N_j, the MPM in N_i must increase its timestamp and register tuple <j, t_j>, then (1) call *Exchange* procedure to update its SI, (2) call *Order* procedure to determine several node's order to enter the CS (if they can be ordered) employing RCV algorithm (Line 33-37). Obviously, the information included in the message is collected from the nodes along its forwarding path. If N_j is ordered, then its immediate preceding node in the *NONL* N_k will receive an IM or N_j itself will receive an EM from N_i (if N_j is on top of *NONL*) (Line 38-45). Otherwise, when N_i cannot be determined its access order (that is to say the information carried by the request message is not enough for determining its home node's access order), N_i will regenerate an RM with newest system information but remains the "Host" to be "j" and forward it to some other node which exists in the RM's *UL* (any of the unvisited nodes) (Line 46-53).

The MPM Algorithm

```
1.  Initialization:
2.   //Omitted
3.  Upon requesting the CS:
4.      $SI_i$.NSIT[i].TS++;
5.      Append tuple <i, $SI_i$.NSIT[i].TS> to
        $SI_i$.NSIT[i].MNL;
6.      Create a message with following content:
7.          //initialize RM, copying information needed
8.          Host = i;
9.          UL = {$N_x$ | $0 \le x \le N-1$}-$N_i$;
10.         MONL = $SI_i$.NONL;
11.         MSIT = $SI_i$.NSIT;
12.     Select an unvisited node randomly and delete
        corresponding id from message UL;
13.     Send the message to the selected node;
14. Upon receiving the EM:
15.     Call Exchange Procedure to update the $SI_i$.$NSIT_i$ ;
16.     Enter the CS;
17. Upon releasing the CS:
18.     $SI_i$.NSIT[i].TS++;
```

```
    Delete i from $SI_i$.NONL$_i$
    If ( $SI_i$ . Next<> NULL ) then  // send EM message
    informing the next node to enter CS if any
       Initialize an EM with newest MONL and MSIT
         copy from $SI_i$ ;
       Send the EM to the $SI_i$.Next;
       $SI_i$.Next = NULL;
    End if
Upon receiving an IM:
    (IM indicating next to be node j)
    If ( tuple <i, $t_i$> which is immediate precedes tuple
    <j, $t_j$> is not in the list of $SI_i$.NONL) then
       // this node has finished executing the CS
       initialize an EM with newest MONL and MSIT
      copy from $SI_i$;
       send the EM to node j;
    else
       set $SI_i$.Next = j;
    end if
Upon receiving/processing a RM: (Assume that the
  message initialized at node k arrives in node i)
    Call Exchange Procedure to update $SI_i$.NSIT;
    Append tuple <k, $t_k$> to $SI_i$.NSIT[i].MNL;
    $SI_i$.NSIT[i].TS=max($SI_i$.NSIT[h].TS)+1
    ($h \in [0, N-1]$);
    Call Order procedure;
    if BeOrdered = true  then
       if Highest_Priority = true then
          Initialize an EM with newest MONL and
           MSIT copy from $SI_i$ ;
          send the EM to node k;
       else
          // informing k's preceding node to reset its
           Next field
          send an IM to node k's immediate preceding
           node according to NONL;
       end if
    else  // forward this RM with updated information
       generate a new RM' with following content:
          Host' = k;
          MONL' = $SI_i$.NONL;
          MSIT' = $SI_i$.NSIT;
          choose one unvisited node (assume node h)
           from the UL of RM;
          UL' = UL - $N_h$;
       send the message to node h;
    end if
```

4.2. The Order specifications

When request message originally initialized in N_j is delivered to N_i, in this procedure, it will determine whether N_j can be ordered by employing the RCV scheme. First, all tuples existing in the *NSIT* will be organized as a sequence {TP_i} temporarily. The rank of a tuple in the sequence is defined by two parameters: the number of *MNL*s in which the tuple is placed on the top and the value of *NodeID*. The latter is used to resolve any tie: when more than one tuple are placed on the same number of *MNL*s, the tuple with smallest *NodeID* wins and will be assigned the highest rank (Line 12). Afterwards, the first tuple in {TP_i} is tested to determine whether it can be ordered (Line 13).

All ordered tuple will be appended to *NONL* and removed from all *MNL*s of the *NSIT* (Line 14, 15). The boolean variables *BeOrdered* and *Highest_Priority* will be set to true if node N_i is ordered (Line 16-19) and is on top of the *NONL* (Line25).

The Order Procedure

```
1.  Continue = true;
2.  BeOrdered = false;
3.  if (tuple <$j$, $t_j$> is in ONL) // already ordered when
    processing other RM
4.     Continue = false
5.     BeOrdered = true;
6.     delete <$j$ ,$tj_k$> from any entry of NSIT;
7.  end if
8.  while Continue = true Do
9.     begin
10.     //calculate upon the SI stored in node j after
         updating with the incoming message.
11.     // RCV scheme, some node(s) can be ordered
         simultaneously in this procedure
12.     finds all the $M$ ($M$ <= $N$) different tuple in the
         NIST to build the sequence {$TP_h$}: here, each $TP_h$
         reaches the top of $S_h$ ($S_h$>=1) rows of MNL in
         NSIT, and $\forall k,l(1 \le k,l \le M)$, if
         $(k < l)$ then{ ( $S_k > S_l$ or [( $S_k = S_l$ ) and
         $TP_1.NodeID < TP_2.NodeID$)]}; if there is only one
         tuple in the sequence, then $S_2$=0, $S_2.NodeID$=1;
13.     if $\left(S_1 - S_2 > N - \sum_{h=1}^{M} S_h\right)$ or ( $S_1 - S_2 = N - \sum_{h=1}^{M} S_h$
         and ($TP_1.NodeID < TP_2.NodeID$ ))  then
14.        append the $TP_1$  to NONL$_j$ ;
15.        delete $TP_1$ from any row of NSIT$_i$ ;
16.        if ($TP_1.NodeID$ = j ) then
17.           Continue = false;
18.           BeOrdered = true;
19.        endif
20.     else
21.        Continue = false;
22.     endif
23.   end
24. endif
25. if OnTopOf(NONL$_i$) then Highest_Priority = True ;
26. // node j can enter CS immediately
```

4.3. The Exchange specifications

In this procedure, MPM updates the node's SI with the incoming message by comparing the content of tuples. After executing this procedure, newest information will be append and outdated data will be deleted. (Assume that a message arrives at node i)

First, the information in *SI_i.NONL* and *MONL* are synchronized: outdated tuples are deleted from *MONL* (line 1, 2). According to line 1-2, if a tuple <j, t_j> is in *MONL* but not in *SI_i.NONL* and *SI_i.NSIT[j].MNL*, it can be inferred that node N_j has been ordered. However, to the current node i, it may be the case that N_j had finished executing the CS or that no message containing information about N_j has never visited N_i before. The two cases can be distinguished by the difference between the timestamp of N_j maintained by the message (*MSIT[j].TS*) and that of the current node (*SI_i.NSIT[i].TS*). If the timestamp of N_j maintained by the message is smaller than that of the current node, j must be outdated and can be removed.

When N_i receives an EM, it knows that all the nodes whose tuple is preceding it's tuple in the Ordered Node List have finished executing the CS and can be safely deleted from its own *NONL*. So, in this procedure, tuples that precede <i, t_i> in Ordered Node List also can be deleted (line 3, 4). Otherwise, it will be added to *SI_i.NONL* in the rest of the procedure. Next, *SI_i.NONL* and *MONL* are combined if needed and relevant tuples are deleted from *SI_i.NSIT* (line 5-12).

Last, *SI_i.NSIT* and *MSIT* are synchronized (line 13-22). If the N_j's timestamp in *MSIT* of the message and the current node is the same, then no information need to be exchanged because the message and the current node maintain the same state about N_j. Otherwise, information update is needed. First, outdated tuples, if any, are deleted from *SI_i.NSIT* and *MSIT* (line 15-18). Then *SI_i.NSIT[j]* is set as *MSIT* if its *TS* is smaller (line 19, 20).

The Exchange Procedure

```
1.  if $\exists a \in MONL$ and $a \notin SI_i.NONL$ and
      $a \notin SI_i.NSIT[Host].MNL$ and
      $MSIT[a.NodeID].TS < SI_i.NSIT[a.NodeID].TS$ then
2.      delete tuples which precede a and a from MONL;
3.  if $\exists b \in SI_i.NONL$ and $b \notin MONL$ and
      $b \notin MSIT[Host].MNL$ and
      $MSIT[a.NodeID].TS > SI_i.NSIT[a.NodeID].TS$ then
4.      delete tuples which precede a and a from
          $SI_i$.NONL;
5.  if Length (MONL) >Length ($SI_i$.NONL) then
6.      for every tuple c in MONL and not in $SI_i$.NONL
7.          delete c from any entry of $SI_i$.NSIT;
8.      Set $SI_i$.NONL = MONL;
9.  else
10.     for every tuple c in in $SI_i$.NONL but not in MONL
11.         delete c from any entry of MSIT
12. end if
13. for k = 1 to N do
14.     if $SI_i$.NSIT[k].TS <> MSIT[k].TS then
15.         if there exists a tuple <k, $t_k$> in
              $SI_i$.NSIT[k].MNL but is not in MSIT[k].MNL
              and ($SI_i.NSIT[k].TS < MSIT[k].TS$) then
16.             delete <k, $t_k$> from any entry of $SI_i$.NSIT;
17.         if there exists a tuple <k, $t_k$> in MSIT[k].MNL
              but is not in $SI_i$.NSIT[k].MNL and
              ($SI_i.NSIT[k].TS > MSIT[k].TS$) then
18.             delete <k, $t_k$> from any entry of MSIT;
19.         if $SI_i$.NSIT[k].TS < MSIT[k].TS then
20.             set $SI_i$.NSIT[k] = MSIT[k];
21.     end if
22. end for
```

5. Correctness proof

In this section, we present the correctness argument which shows that the proposed algorithm achieves mutual exclusion and is also free of deadlock and starvation. We first give some lemmas.

Lemma 1. $\forall$i $\forall$j, $|S_i.NSIT[j].MNL| < N$ (|*MNL*| is the length of *MNL*)

Proof: In the assumed distributed system consisting of *N* nodes, each node contains only one process that makes a request to mutual exclusively access the CS and each process initiates at most one outstanding request at any time. So, a node will never issue a new request message until it finishes executing the CS for the previous request. In other words, there won't be two different tuples for a node itself in the node's own *NSIT*.

A node' knowledge about other nodes is reflected in its *NSIT*. The *NSIT* stores tuples in the field of *MNL*, representing who and when sent request message to that node. It is obviously that the tuples come from the information carried by an incoming message. The message copies information from the node where it is generated. When a message (RM, EM or IM) arrives in a node, it will firstly processed by the Exchange procedure, which ensures that a *MNL* does not contain any pair of tuples that has the same *NodeID*: if there exist two tuples with the same *NodeID* but different *TS* in *MSIT* and *NSIT* respectively, the one with smaller timestamp must be outdated and will be deleted in the procedure. As only one tuple survives for any other nodes in each entry of the *NSIT*, there will be at most *N* tuples in a *MNL*.

Lemma 2. When every *MNL* in *NSIT* is nonempty, at least one node can be ordered after the execution of the Order procedure.

Proof: During the execution of Order, a sequence of tuples regarding their order is generated. The order is determined by the number of *MNL*s of which a tuple stands on the top and the value of its *NodeID*. If every *MNL* in the *NSIT* is nonempty, we then have $N - \sum_{h=1}^{M} S_h = 0$. At least, the first tuple in the sequence can be ordered after the execution of the Order procedure because either the first tuple in the sequence either holds more first positions or it has smaller *NodeID* value.

Lemma 3. A node N_i's rank to enter the CS can be determined after its corresponding RM message has been forwarded at most *N-1* times.

Proof: Since the RM^t_i (RM^t_i means the request message was initialized at node N_i at timestamp t) will not be sent to a node that it has already been forwarded to, after *N-1* times of forwarding and information exchange, tuple $<i, t>$ will appear in each *MNL* of *NSIT* of the last visiting node (we assume the node is N_k). It's obviously that every *MN*L isn't empty. By **Lemma 2**, at least one node can be ordered, so its corresponding tuple $<j, t'>$ will be deleted from $NSIT_k$ and appended to $NONL_k$. If $j \neq i$, another tuple will be deleted from $NSIT_k$ and appended to $NONL_k$. This procedure will be repeated according to **Lemma 2** and the specification of Order. By **Lemma 1**, there are at most *N* tuples in each *MNL*, which means that after finite times of iterations, node N_i will be ordered at some node N_k.

Lemma 4. When a RM^t_i is ordered and is appended to the *NONL* at certain node, the tuple for the nodes which precede N_i in *NONL* and haven't finished executing the CS, if any, must still exist in that *NONL*.

Proof: When a node N_j precedes N_i in entering the CS was ordered, if any, then according to Order procedure, tuple $<j, t'>$ must have been ranked the highest among the *M* ($M<N$) tuples competing for entering the CS at that time. Semantically, a tuple $<j, t'>$ will not be deleted from any *NONL* and *NSIT* until N_j hasn't finished executing the CS. Otherwise in any node, $<j, t'>$ will either exists in the *NONL* or is still in the *NSIT* keeping unordered. When RM^t_i is processed at certain node N_k, if tuple $<j, t'>$ hasn't been ordered, the rank that tuple $<i, t>$ can achieve should be no higher than that of tuple $<j, t'>$. Consequently, RM^t_i cannot determine its own order because it needs to wait until RM^t_j is ordered. Thus, when RM^t_i is able to be ordered and appended to *NONL* at some node, RM^t_j is already in that *NONL* if N_j hasn't exited from the CS.

Lemma 5. Two different tuples cannot achieve the same rank.

Proof: By definition, a tuple is ranked according to the number of *MNL*s in which it stands on the top and the value of the *NodeID* field. It is straightforward that two different tuples cannot achieve the same rank.

Lemma 6. In the Exchange procedure, after outdated tuples are deleted, either $MONL \subseteq NONL$ or $MONL \supseteq NONL$ and tuples on the top of the two lists are the same if both are nonempty.

Proof: Assume the contrary, neither $MONL \subseteq NONL$ nor $MONL \supseteq NONL$ is true. It is clear that $| NONL | >0$ and $|MONL | > 0$. So there must exist at least one tuple, which can be denoted as A, and we have $A \in MONL$, $A \notin NONL$. Also, there must exist tuples, which is denoted as B, we have $B \in NONL$ but $B \notin MONL$. We assume that $MONL=\{A_k\}$ $(1 \leq k \leq M)$, $NONL =\{B_k\}(1 \leq k \leq M')$.

Firstly, we consider $NONL \cap MONL =\varnothing$. According to the **Order** Procedure, A_{k1} is ranked higher than A_{k2} if $k1 < k2$ in *MONL*, and $B_{k'1}$ is ranked higher than $B_{k'2}$ if $k'1 < k'2$ in *NONL*. By **Lemma 5**, two different tuples could not achieve the same order. Since $NONL \cap MONL=\varnothing$, by **Lemma 4**, either A_M precedes all tuples in *NONL* or $B_{M'}$ precedes all tuples in *MONL*. But in the first case, since B_1 must know A_M is ordered before it got ordered, A_M is sure to exist in *NONL*. In the second case, A_1 must know $B_{M'}$ is ordered before it got ordered, then $B_{M'}$ should exist in *MONL*. This is obviously contrary to $NONL \cap MONL =\varnothing$.

When $NONL \cap MONL \neq\varnothing$, we consider the minimum *k* where $A_k \neq B_k$. When $k>1$, $A_{n-1}=B_{n-1}$ for any $n \in [1, k)$. But we have the instance that different A_k and B_k achieve the same rank (they both rank k^{th} in the *NONL* and *MONL*), which is contrary to **Lemma 5**. In case of $k=1$, we can find the minimum k', where $A_{k'}= B_{k'}$. And if $k' >1$, $A_{n-1} \neq B_{n-1}$ for any $n \in [1, k')$. According to the proposed algorithm, both $A_{k'-1}$ and $B_{k'-1}$ precede $A_{k',}$ but by **Lemma 5**, $A_{k'-1}$ and $B_{k'-1}$ which are different tuples cannot achieve the same rank. Assume they are of different rank, by **Lemma 4**, if $A_{k'-1}$ precedes $B_{k'-1}$, $B_{k'-1}$will be appended to *MONL* after $A_{k'-1}$ but before $A_{k'}$. Or if $B_{k'-1}$ precedes $A_{k'-1}$, $A_{k'-1}$ will be appended to *NONL* after $B_{k'-1}$ but before $B_{k'}$. It is contrary to the assumption. In each case, there is contradiction.

Lemma 7. Tuples in any *NONL*s are ranked in the same order.

Proof: We assume that a RM^t_i is sent from node N_j to N_j, so $MONL = NONL_i$ and $MSIT= NSIT_i$ (here, we use $NONL_i$ and $SI_i.NONL$ equally for simplicity). We re-denote *MONL* and $NONL_k$ as *MONL'* and $NONL_k'$ after outdated ordered tuples being deleted. By **Lemma 6**, either $MONL' \subseteq NONL_k'$ or $MONL \supseteq NONL_k'$ is true, and tuples on the top of the two lists are the same if both are nonempty. This means tuples in *MONL* and $NONL_k$ are ranked in the same order except outdated ordered tuples.

Without losing generality, we assume that $MONL' \subseteq NONL_k'$. What is left to be proved now is that an outdated ordered tuple, before it is deleted, is ordered equally in *MONL* and $NONL_k$. According to the definition of outdated ordered tuple given in section 3.2.2 and the specification of **Order** procedure, such a tuple precedes all the ordered tuples that is not outdated and there is no such tuple existing in both $NONL_k$ and *MONL*.

If both *MONL'*and $NONL_k'$ are nonempty, we only need to prove that all of outdated tuples must exist either in $NONL_k$ or in *MONL*. Assume the contrary, there is an outdated ordered tuple $A \in MONL$ and $A \notin NONL_k$ and another outdated ordered tuple $B \in NONL_k$ and $B \notin MONL$. By **Lemma 5**, tuple *A* and *B* cannot achieve the same rank. But by **Lemma 4**, if tuple *B* precedes *A*, *A* precedes all tuples in $NONL_k'$, *A* will be appended to $NONL_k$ after *B* and before any other tuples in $NONL_k'$. Otherwise, i.e. tuple *A* precedes *B*, *B* will be appended to *MONL* after *A* and before any other tuples in *MONL'*. We can see that both cases result in contradiction distinctly.

If *MONL'* is empty, we only need to prove that any outdated tuples in *MONL* precedes all tuples in $NONL_k$. Also assume the contrary, there is an outdated ordered tuple $A \in MONL$, $A \notin NONL_k$ while another outdated ordered tuple $B \in NONL_k$, $B \notin MONL$ and *B* precedes *A*. By **Lemma 4**, *A* will be appended to $NONL_k$ after *B* and before any other tuples in $NONL_k'$. It is contradiction to $A \notin NONL_k$.

Thus, tuples in any two different *NONL*s are ranked in the same order.

Lemma 8. When a node N_i is executing the CS, tuple *<i, t>* must stand on the top of $NONL_i$.

Proof: From the algorithm presented above, there are only two cases in which N_i can enter the CS. One is that when N_i is ordered at some node N_k after executing the Order procedure, its corresponding tuple *<i, t>* is nicely on the top of $NONL_k$. Since *<i, t>* gets *Highest_Priority*, N_k will send an EM message to N_i informing it to enter the CS immediately. When N_i receives the EM, it will invoke the Exchange procedure. Since tuple *<i, t>* is on the top of *MONL* of the incoming EM, after executing the Exchange procedure, *<i, t>* will stand on the top of $NONL_i$ by **Lemma 6**.

In another case, the order of N_i is determined without *Highest_Priority*. Only when N_i receives an EM message from its directly preceding node N_j can it enter the CS. Since N_j has finished executing the CS, N_j will delete its corresponding tuple from $NONL_j$ so that *<i, t>* will stand on the top. When N_i receives the EM, it will delete all tuples which precede *<i, t>* from $NONL_i$. Thus tuple *<i, t>* will be also on the top of $NONL_i$.

Theorem 1. Mutual exclusion is achieved.

Proof: Mutual exclusion is achieved when any pair of nodes is never simultaneously executing the critical section. Assuming the contrary that two nodes N_i and N_j are in the CS at the same time, so tuple $<i, t_i>$ and $<j, t_j>$ must reside on top of $NONL_i$ and $NONL_j$ respectively according to **Lemma 8**.

Now, let's assume that a message (RM, IM or EM) is sent from node N_i to N_j with $MONL=NONL_i$ and $MSIT=NSIT_i$. Because both N_i and N_j are simultaneously in their CS, neither $<i, t_i>$ or $<j, t_j>$ can be considered as outdated tuples and be deleted as in the Exchange procedure. Thus the facet that tuples in $NONL_i$ and $NONL_j$ are not ranked in the same order is contrary to **Lemma 6** and **Lemma 7**.

Theorem 2. Deadlock is impossible.

Proof: The system is deadlocked when no node is in its critical section and no requesting node can ever proceed to its own critical section. Assume the contrary that the deadlock is possible, in our algorithm, it will result in two cases. First case, no node could determine its order to enter the CS. This is contrary to **Lemma 2** and **Lemma 3** because every requesting node could determine its order to enter the CS after its corresponding RM message is forwarded no more than *N*-1 times. In the second case, there exist three node N_A, N_B and N_C, where N_A is waiting for EM message from N_B directly or indirectly, N_B is waiting for EM message from N_C directly or indirectly and N_C is waiting for EM message from N_A directly or indirectly. Then N_A precedes N_C, N_C precedes N_B and N_B precedes N_A. It is contradiction to **Lemma 7**, so in our algorithm, deadlock is impossible.

Theorem 3. Starvation is impossible.

Proof: Starvation occurs when one node must wait indefinitely to enter its critical section even though other nodes are entering and exiting their own critical section. Assume the contrary, that starvation is possible. In our algorithm, time need to execute the algorithm and CS and the time for message transfer are all finite. Since a requesting node's order to enter the CS is determined after its corresponding RM message has been forwarded at most *N*-1 times (**Lemma 3**), the reason that cause a node to be in starvation must be waiting for its preceding nodes infinitely. But by **Lemma 4** and **Lemma 7**, the sequence of ordered nodes which precede it is determined once one node gets ordered. So, the node in starvation will receive an EM message from its directly preceding node and enter the CS in finite time. Thus a contradiction occurs and the theorem must be true.

6. Performance Evaluation

As mentioned before, there are three measures to evaluate the performance of a mutual exclusion algorithm: message complexity, response time and synchronization delay. The performance of an algorithm depends upon loading conditions of the system and has been usually studied under two special loading conditions: light load and heavy load. Under the light load condition, there is seldom more than one request for mutual exclusion simultaneously in the system while under the heavy load condition, there is always a pending request in a node. We first present an analysis of the performance of the proposed algorithm under the two different cases. Then we describe our simulation study, which focus on the heavy load condition, and discuss the simulation results.

6.1. Performance Analysis

6.1.1. Message Complexity. When a node wants to enter the CS, it must firstly initialize a RM message with a copy of its SI. The RM message will be forwarded among nodes carrying up-to-date system information until its home node's ID stands on top of relative majority of *MNL*s, meaning that it gains enough permissions. Under the light load condition, if there is no outdated information, the RM's host ID will be on top of each *MNL* the message travels. So, after being forwarded *[N/2]+1* times (the RM will be updated on each forwarding), the currently processing MPM will find that the RM's host ID has been ordered with *Hightest_Priority* and immediately send an EM message to that server. Hereby, the message complexity is *[N/2]+2*. (Here, we use the square brackets denoting a function that gets the integer part of a digit.)

When there exists outdated information, if the outdated information can be deleted within *[N/2]+1* forwarding times, the message complexity will be *[N/2]+2* all the same. Otherwise, in the worst case, the RM will be forwarded *N-1* times visiting all system nodes until it can be ordered. In this case, the message complexity is *O(N)*.

Under the heavy load condition, when a total of *m* nodes are competing for the same CS, a node who is granted the privilege must have its ID standing on the top of at least *[N/m]+1 MNL*s. The minimum times the RM message is forwarded is *[N/m]+2*. So the message complexity is calculated by counting how many nodes in the system are competing the CS simultaneously, and the number of competing nodes *m* is decided by the distribution model that describes how frequently a node requests the CS.

6.1.2. Synchronization delay. It is obvious that the synchronization delay of our algorithm is *T* where *T* is average message propagation delay, because only one enter message is needed to be passed between two successive executions of the CS.

6.1.3. Response Time. Under low load condition, before a node can enter the CS, its corresponding request message needs to be forwarded *N/2* to *N-1* times to determine its order and one enter message for entering the CS. So the response time will be $([N/2]+2)*T_n$ to $(N-1)*T_n$. Under high load condition, if each node will wait for an enter message to enter the CS, the response time is sure to be $N*(T_n+T_c)$ under heavy load condition where T_c is the average CS executing time.

6.2. Simulation Results

Although our algorithm has high message complexity and long response time in the worst case, it has good performance in general. In fact, the heaver the system load is, the better our algorithm performances. To demonstrate that, a simulation is conducted to evaluate our algorithm against several other algorithms including the Maekawa which is low in message complexity and the Broadcast, Ricart which are low in response time.

We adopt a simulation model similar to the one used in [14]: requests for CS execution arrive at a site according Poisson distribution with parameter λ; message propagation delay between any pair of nodes T_n and CS execution time T_c are all constant to be 5 and 10 time units (although this condition is not necessary, we still take it for ease). The whole system is free of node crash and communication failure. Two measure results are collected, viz. number of message exchanged (*NME*) and response time (*RT*) per CS execution. Here, we use the first method mentioned in [9] to generate quorums for Maekawa's algorithm.

We first consider the situation that all nodes are requesting the CS simultaneously as soon as the system is initialized. Every node only requests once. When the system is initialized, each node knows nothing about others in our algorithm. So in this situation, we can see how soon and sufficient the system information exchanges. Figure 4 and 5 plot the average *NME* and *RT* against the number of nodes in the system. It is shown that when the number of nodes increases, the messages exchanged and time delay both increase. Moreover, our algorithm has the least messages exchanged of the four algorithms while its average response time is similar to the other three's.

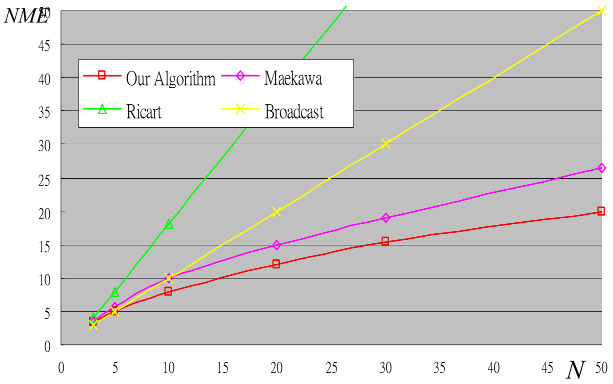

Figure 4. **Message number Vs node number**

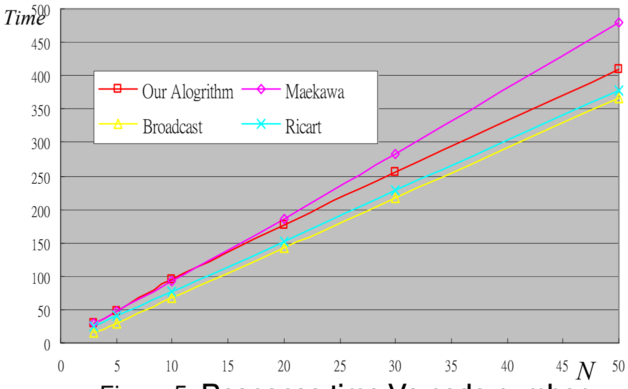

Figure 5. **Response time Vs node number**

Afterwards, a system of 30 nodes is simulated with different request arrival rate λ. We run the simulation for enough long time (100000 time units) repeatedly and record the *NME* and *RT* for all successful requests. Figure 6 and Figure 7 illustrate the performance comparison of the four simulated algorithms. As the load increases, messages needed for per mutual exclusion decreases in our algorithm. It is clear that the heaver the system load is, the better our algorithm outperforms the Maekawa in average *NME*. Although the average response time of our algorithm is a little higher than that of the Broadcast and the Ricart, it is much lower than the Maekawa's. Since the Broadcast and the Ricart need much more messages exchanged to achieve mutual exclusion, our algorithm performances better than the other three in general.

7. Conclusion

In this paper, we described an efficient fully distributed algorithm for mutual exclusion. Our algorithm imposes no specified structure on the system topology and doesn't force messages to be delivered in FIFO order. These two merits make our algorithm more attractive in applications. In addition, we adopt the RCV scheme to schedule nodes that are intended to execute the critical section. Since the RCV scheme comes from the famous MCV scheme, the algorithm gains high resiliency. We have presented the proof of correctness of the algorithm, with respect to guaranteed mutual exclusion, deadlock freedom and starvation freedom. Both analysis and simulation are used

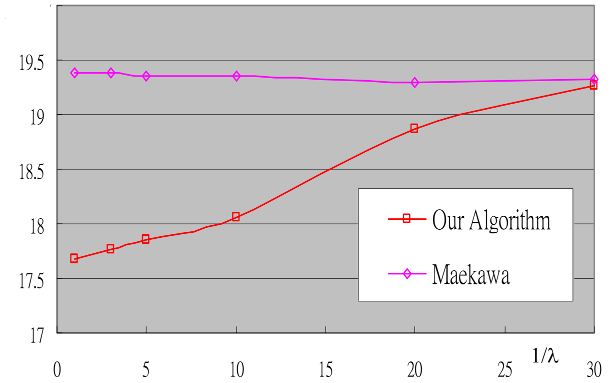

Figure 6. Message number Vs λ

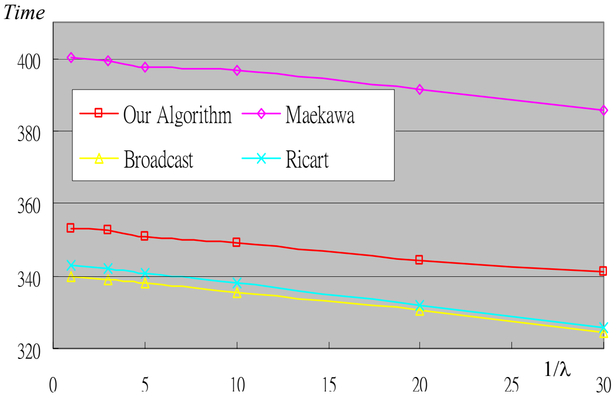

Figure 7. Response time Vs λ

to evaluate the algorithm's performance. Simulation results compare our algorithms with some existing algorithms and show that the proposed algorithm outperforms other algorithms especially under high load condition.

In our future work, we will conduct simulation studies to compare with more existing algorithms. We will also investigate how to improve the algorithm by designing different methods for forwarding the request messages.

Acknowledgement

This work is partially supported by Hong Kong Polytechnic University under HK PolyU research grants G-YD63 and G-YY41, and The National 973 Program of China under grant 2002CB312002.

Reference

[1] D. Agrawal, and A. E. Abbadi, "An Efficient and Fault-Tolerant Solution for Distributed Mutual Exclusion", *ACM Transactions on Computer Systems*, Vol. 9(1), Feb。 1991, pp. 1-20.

[2] D. Agrawal, and A. E. Abbadi, "A Token-Based Fault-Tolerant Distributed Mutual Exclusion Algorithm", *Journal of Parallel and Distributed Computing*, Vol. 24(2), 1995, pp. 164-176.

[3] S. Banerjee , and P. K. Chrysanthis, "A New Token Passing Distributed Mutual Exclusion Algorithm", In *Proceedings of 16th International Conference on Distributed Computing Systems*, 1996, Hong Kong, May. 27-30, 1996, pp. 717 -724.

[4] G. Cao, and M. Singhal, "A Delay-Optimal Quorum-Based Mutual Exclusion Algorithm for Distributed Systems", *IEEE Transaction on Parallel and Distributed Systems*, Vol. 12(12), Dec. 2001, pp. 1256-1267.

[5] Ye-In Chang, "Notes on Maekawa's O(√ N) Distributed Mutual Exclusion Algorithm", In *Proceedings of the 5th IEEE Symposium on Parallel and Distributed Processing*, Dallas, USA, Dec. 1-4, 1993, pp. 352-355.

[6] M. Choy, "Robust Distributed Mutual Exclusion", In *Proceedings of the 16th International Conference on Distributed Computing Systems*, Hong Kong, May. 27-30, 1996, pp. 760 -767.

[7] L. Lamport, "Time, Clocks and Ordering of Events in Distributed Systems", *Communications of the ACM*, Vol. 21(7), Jul. 1978, pp. 558-565.

[8] S. Lodha, and A. Kshemkalyani, "A Fair Distributed Mutual Exclusion Algorithm", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 11(6), June 2000, pp. 537-549.

[9] M. Maekawa, "A √N Algorithm for Mutual Exclusion in Decentralized Systems", *ACM Transaction on Computer Systems*, Vol. 3(2), May. 1985, pp. 145-159.

[10] S. Nishio, K. F. Li, and Manning E. G. "A Resilient Mutual Exclusion Algorithm for Computer Networks", *IEEE Transactions on Parallel Distributed Systems*, Vol. 1(3), Jul. 1990, pp. 344-355.

[11] S. Rangarajan, S. Setia, and S.K. Tripathi, "A Fault-Tolerant Algorithm for Replicated Data Management", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 6(12), Dec. 1995, pp. 1271-1282.

[12] K. Raymond, "A Tree-based Algorithm for Distributed Mutual Exclusion", *ACM Transactions on Computer Systems*, Vol. 7(1), Feb. 1989, pp. 61-77.

[13] G. Ricart, and A. K. Agrawala, "An Optimal Algorithm for Mutual Exclusion in Computer Networks", *Communications of the ACM*, Vol. 24(1), Jan. 1981, pp. 9-17.

[14] M. Singhal, "A Heuristically-Aided Algorithm for Mutual Exclusion in Distributed Systems", *IEEE Transactions on Computers*, Vol. 38(5), May. 1989, pp. 651-662.

[15] M. Singhal, "A Dynamic Information Structure Mutual Exclusion Algorithm for Distributed Systems", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 3(1), Jan. 1992, pp. 121-125.

[16] M. Singhal, "A Taxonomy of Distributed Mutual Exclusion", *Journal of Parallel and Distributed Computing*, Vol. 18, 1993, pp. 94-101.

[17] I. Suzuki, and T. Kasami "A Distributed Mutual Exclusion Algorithm", *ACM Transactions on Computer Systems*, Vol. 3(4), Nov. 1985, pp. 344 - 349.

[18] R. H. Thomas, "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases", *ACM Transactions on Database Systems*, Vol. 4(2), Jun. 1979, pp. 180–209.

[19] T. Tsuchiya, M. Yamaguchi, and T. Kikuno, "Minimizing the Maximum Delay for Reaching Consensus in Quorum-Based Mutual Exclusion Schemes", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 10(4), Apr. 1999, pp. 337-345.

[20] J. E. Walter, J. L. Welch, and N. H. Vaidya, "A Mutual Exclusion Algorithm for Ad Hoc Mobile Networks", *Wireless Networks*, Vol. 7(6), Nov. 2001, pp. 585-600.