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 \(T_p\).

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 \(2T_p\) at light loads and \(N*(T_p+T_c)\) 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 $N*(T_c+T_i)$ 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 overheads.

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 $N*(T_c+T_i)$ 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 \(G + \frac{N}{2G}\), 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:

- **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.

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.
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 Ni wants to enter a CS, it increases its timestamp t by one, and appends the tuple <i, t> to SIi.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 Ni 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, Ni will either receive an EM from Nk which just exits the CS or from Nj where its order is determined. On getting enough permissions to enter the CS, Ni 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, Ni must send an EM to the node represented by Nexti, if any, and delete its own tuple from the top of SIi.NONL (Line 17-24).

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

Upon receiving a RM originally initialized in Nj, the MPM in Ni must increase its timestamp and register tuple <j, t>, 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 Ni is ordered, then its immediate preceding node in the NONL Ni will receive an IM or Ni itself will receive an EM from Nj (if Ni is on top of NONL) (Line 38-45). Otherwise, when Ni 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), Ni will regenerate an RM with newest system information but remains the “Host” to be “i” 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. SIi.NSIT[i].TS++;
5. Append tuple <i, SIi.NSIT[i].TS> to SIi.NSIT[i].MNL;
6. Create a message with following content:
7. //initialize RM, copying information needed
8. Host = i;
9. UL = {Nk | 0 ≤ x ≤ N-1}-Ni;
10. MONL = SIi.NONL;
11. MSIT = SIi.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 SIi.NSIT;
16. Enter the CS;
17. Upon releasing the CS:
18. SIi.NSIT[i].TS++;
19. Delete i from SIi.NONLi.
20. If ( SIi.Next<> NULL ) then // send EM message informing the next node to enter CS if any
21. Initialize an EM with newest MONL and MSIT copy from SIi;
22. Send the EM to the SIi.Next;
23. SIi.Next = NULL;
24. End if
25. Upon receiving an IM: (IM indicating next to be node j)
26. If ( tuple <i, ti> which is immediate precedes tuple <j, tj> is not in the list of SIi.ONL) then
27. // this node has finished executing the CS
28. initialize an EM with newest MONL and MSIT copy from SIi;
29. send the EM to node j;
30. else
31. set SIi.Next = j;
32. end if
33. Upon receiving/processing a RM: (Assume that the message initialized at node k arrives in node i)
34. Call Exchange Procedure to update SIi.NSIT;
35. Append tuple <<k, tk>> to SIi.NSIT[i].MNL;
36. SIi.NSIT[i].TS=max(SIi.NSIT[i].TS)+1
37. Call Order procedure;
38. if BeOrdered = true then
39. if Highest_Priority = true then
40. Initialize an EM with newest MONL and MSIT copy from SIi;
41. send the EM to node k;
42. else
43. // informing k’s preceding node to reset its Next field
44. send an IM to node k’s immediate preceding node according to NONL;
45. end if
46. else // forward this RM with updated information
47. generate a new RM’ with following content:
48. Host’ = k;
49. MONL’ = SIi.NONL;
50. MSIT’ = SIi.NSIT;
51. choose one unvisited node (assume node h) from the UL of RM;
52. UL’ = UL - Nh;
53. send the message to node h;
54. end if
4.2. The Order specifications

When request message originally initialized in Ni is delivered to Nh in this procedure, it will determine whether Nh can be ordered by employing the RCV scheme. First, all tuples existing in the NSIT will be organized as a sequence {TPi} temporarily. The rank of a tuple in the sequence is defined by two parameters: the number of MNLs 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 MNLs, the tuple with smallest NodeID wins and will be assigned the highest rank (Line 12). Afterwards, the first tuple in {TPi} is tested to determine whether it can be ordered (Line 13).

All ordered tuple will be appended to NONL and removed from all MNLs of the NSIT (Line 14, 15). The boolean variables BeOrdered and Highest_Priority will be set to true if node Ni 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, tj> is in ONL) // already ordered when processing other RM
4. Continue = false
5. BeOrdered = true;
6. delete <j, tj> 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 {TPh}; here, each TPk reaches the top of Sh (Sh >=1) rows of MNL in NSIT, and ∀k, l(1 ≤ k, l ≤ M), if
13. (k < l) then{ (Sh > Sl) or ([Sh = Sl] and TPk.NodeID < TPk.NodeID))}; if there is only one tuple in the sequence, then Sh=0, Sx.NodeID=1;
14. if (Sh > Sl) or (Sh = Sl) and (TPk.NodeID < TPk.NodeID)) then
15. append the TPk to NONLk;
16. delete TPk from any row of NSITk;
17. if (TPk.NodeID = j) then
18. Continue = false;
19. endif
20. else
21. Continue = false;
22. endif
23. end
24. endif
25. if OnTopOf(NONLj) 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 $Sl, NONL$ 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 $Sl, NONL$ and $Sl, NSIT[i].MNL$, 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 and the current node’s $Sl, NSIT[i].TS$. 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 $Sl, NONL$ in the rest of the procedure. Next, $Sl, NONL$ and $MONL$ are combined if needed and relevant tuples are deleted from $Sl, NSIT$ (line 5-12).

Last, $Sl, NSIT$ 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 $Sl, NSIT$ and $MONL$ (line 15-18). Then $Sl, NSIT[j]$ is set as $MSIT$ if its $TS$ is smaller (line 19, 20).

**The Exchange Procedure**

1. if $\exists a \in MONL$ and $a \notin Sl, NONL$ and $Sl, NSIT[Host].MNL$ and $MSIT[NodeID].TS < Sl, NSIT[NodeID].TS$ then
   2. delete tuples which precede $a$ and $a$ from $MONL$;
   3. if $\exists b \in Sl, NONL$ and $b \notin MONL$ and $b \notin Sl, NSIT[Host].MNL$ and $MSIT[NodeID].TS > Sl, NSIT[NodeID].TS$ then
   4. delete tuples which precede $a$ and $a$ from $SI, NONL$;
5. if Length ($MONL$) $>$ Length ($Sl, NONL$) then
   6. for every tuple $c$ in $MONL$ and not in $Sl, NONL$
   7. delete $c$ from any entry of $Sl, NSIT$;
8. Set $Sl, NONL = MONL$;
9. else
   10. for every tuple $c$ in $Sl, NONL$ and not in $MONL$
   11. delete $c$ from any entry of $Sl, NSIT$;
12. end if
13. for $k = 1$ to $N$ do
14. if $Sl, NSIT[k].TS < MSIT[k].TS$ then
15. if there exists a tuple $<k, t_k>$ in $Sl, NSIT[k].MNL$ but not in $Sl, NSIT[k].MNL$ and $Sl, NSIT[k].TS < MSIT[k].TS$ then
   16. delete $<k, t_k>$ from any entry of $Sl, NSIT$;
17. if there exists a tuple $<k, t_k>$ in $MSIT[k].MNL$ but not in $Sl, NSIT[k].MNL$ and $Sl, NSIT[k].TS > MSIT[k].TS$ then
18. delete $<k, t_k>$ from any entry of $MSIT$;
19. if $Sl, NSIT[k].TS < MSIT[k].TS$ then
20. set $Sl, NSIT[k] = MSIT[k]$;
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.** $\forall i \forall j, |Sl, NSIT[j].MNL| < N$ ($MNL$ 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’s 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 $N - \sum_{k=1}^{N} s_k = 0$. 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$_i^j$ (RM$_i^j$ 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 NS$iT_i$ of the last visiting node (we assume the node is N$_j$). It’s obviously that every MNL isn’t empty. By Lemma 2, at least one node can be ordered, so its corresponding tuple $<j, t>$ will be deleted from NS$IT_i$ and appended to NONL$_i$. If j $\neq i$, another tuple will be deleted from NS$IT_i$ and appended to NONL$_i$. 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$_j$ will be ordered at some node N$_i$.

Lemma 4. When a RM$_i^j$ 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 NS$IT_i$ 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 NS$IT_i$ keeping unordered. When RM$_i^j$ is processed at certain node N$_k$, if tuple $<j, t>$ hasn’t been ordered, the rank that tuple $<j, t>$ can achieve should be no higher than that of tuple $<j, t>$. Consequently, RM$_i^j$ cannot determine its own order because it needs to wait until RM$_i^j$ is ordered. Thus, when RM$_i^j$ is able to be ordered and appended to NONL at some node, RM$_i^j$ 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 MNLS 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 = $\emptyset$. According to the Order Procedure, A$_k$ is ranked higher than A$_{k+1}$ if $k1 < k2$ in MONL, and B$_{k+1}$ is ranked higher than B$_{k1}$ if $k1' < k2'$ in NONL. By Lemma 5, two different tuples could not achieve the same order. Since NONL $\cap$ MONL = $\emptyset$, by Lemma 4, either A$_k$ precedes all tuples in NONL or B$_{k'}$ precedes all tuples in MONL. But in the first case, since B$_1$ must know A$_k$ is ordered before it got ordered, A$_{k'}$ is sure to exist in NONL. In the second case, A$_1$ must know B$_{k'}$ is ordered before it got ordered, then B$_{k'}$ should exist in MONL. This is obviously contrary to NONL $\cap$ MONL = $\emptyset$.

When NONL $\cap$ MONL $\neq \emptyset$, we consider the minimum k where A$_{k1} \neq B_{k1}$. When k $\geq$ 1, A$_{k1} \neq B_{k1}$ 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 $k^n$ 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 $\geq$ 1, A$_{k1} \neq B_{k1}$ for any n $\in$ {1, k}'. According to the proposed algorithm, both A$_{k1}$ and B$_{k1}$ precede A$_k$; but by Lemma 5, A$_{k1}$ and B$_{k1}$ which are different tuples cannot achieve the same rank. Assume they are of different rank, by Lemma 4, if A$_{k1}$ precedes B$_{k1}$, B$_{k1}$ will be appended to NONL after A$_{k1}$ but before A$_{k2}$. Or if B$_{k1}$ precedes A$_{k1}$, A$_{k1}$ will be appended to NONL after B$_{k1}$ but before B$_{k1}$. It is contrary to the assumption. In each case, there is contradiction.

Lemma 7. Tuples in any NONLs are ranked in the same order.

Proof: We assume that a RM$_i^j$ is sent from node N$_j$ to N$_i$, so NONL = NONL$_i$ and NS$IT_i$ = NS$IT_j$ (here, we use NONL$_i$ and SL.NONL equally for simplicity). We re-denote MONL and NONL$_i$ as MONL’ and NONL’ after outdated ordered tuples being deleted. By Lemma 6, either MONL’ $\subseteq$ NONL’ or MONL’ $\supseteq$ NONL’ 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’ are ranked in the same order except outdated ordered tuples.

Without losing generality, we assume that MONL’ $\subseteq$ NONL’. What is left to be proved now is that an outdated ordered tuple, before it is deleted, is ordered equally in MONL and NONL. 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 and MONL.

If both MONL and NONL’ are nonempty, we only need to prove that all of outdated tuples must exist either in NONL or in MONL. Assume the contrary, there is an outdated ordered tuple A $\in$ MONL and A $\notin$ NONL. Another outdated ordered tuple B $\in$ NONL 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, A will be appended to NONL after B and before any other tuples in NONL. 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 MON’ is empty, we only need to prove that any outdated tuples in MONL precedes all tuples in NONL. Also assume the contrary, there is an outdated ordered tuple A $\in$ MONL, A $\notin$ NONL while another outdated ordered tuple B $\in$ NONL, B $\notin$ MONL and B precedes A. By Lemma 4, A
Lemma 6. When a node Ni is executing the CS, tuple \(<i, t>\) must stand on the top of NONLi.

Proof: From the algorithm presented above, there are only two cases in which Ni can enter the CS. One is that when Ni is ordered at some node Nk after executing the Order procedure, its corresponding tuple \(<i, t>\) is nicely on the top of NONLi. Since \(<i, t>\) gets Highest_Priority, Nk will send an EM message to Ni informing it to enter the CS immediately. When Ni 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 NONLi by Lemma 6.

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

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 Ni and Nj are in the CS at the same time, so tuple \(<i, t>\) and \(<j, t>\) must reside on top of NONL, and NONLi respectively according to Lemma 8.

Now, let’s assume that a message (RM, IM or EM) is sent from node Ni to Nj with MONL=NONL and MSIT=NSIT. Because both Ni and Nj are simultaneously in their CS, neither \(<i, t>\) nor \(<j, t>\) can be considered as outdated tuples and be deleted as in the Exchange procedure. Thus the fact that tuples in NONL and NONLi are not ranked in the same order is contrary to Lemma 6 and Lemma 7.

Theorem 2. Deadlock is impossible.

Proof: The system is deadlock 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 \(N/2\) times. In the second case, there exist three node \(N_A\), \(N_B\) and \(N_C\), where \(N_B\) is waiting for EM message from \(N_A\) 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 \(N_B\), \(N_C\) 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 MNLS, 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 \(\lfloor N/2\rfloor+1\) 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 Highest_Priority and immediately send an EM message to that server. Hereby, the message complexity is \(\lfloor N/2\rfloor+1\). (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 \(\lfloor N/2\rfloor+1\) forwarding times, the message complexity will be \(\lfloor N/2\rfloor+2\) 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 $[N/m]+1$ MNLs. 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 $((N/2)+2)*T_{m}$ to $(N-1)*T_{m}$. Under high load condition, if each node will wait for an enter message to enter the CS, the response time is sure to be $N*(T_{m}+T_{c})$ 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 heavier 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 $\lambda$; message propagation delay between any pair of nodes $T_{p}$ and CS execution time $T_{e}$ 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.

Afterwards, a system of 30 nodes is simulated with different request arrival rate $\lambda$. 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 heavier 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 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


