A High Efficiency Distributed Mutual Exclusion Algorithm

Dan Liu, Xinsong Liu, Zhijie Qiu, and Gongjun Yan

8010 Division
Department of Computer Science
University of Electronic Science and Technology
Chengdu, Sichuan 610054, P.R.China
liudan@uestc.edu.cn
http://teacher.uestc.edu.cn/teacher/teacher.jsp?TID=yrewy

Abstract. A high efficiency Distributed Mutual Exclusion (DMX) algorithm based on *RA* algorithm, is presented. It puts different mutual exclusion operations for reading and writing requests. The algorithm, belonging to nontoken-based type, saves the message complexity while has *T* synchronization delay. A read/write globe clock stamp which based on the Lamport clock stamp is put forward for the read/write operations. Using the read/write globe clock stamp, reading and writing requests can access Critical Sections (*CS*) with fairness. Furthermore, a dynamic detection mechanism is adopted in the algorithm to realize self-stability.

Keywords: distributed mutual exclusion, read/write clock stamp, self-stability

1 Introduction

DMX algorithms have been studied intensively in the last 20 years. Several taxonomic research papers of the algorithms have been published[1,2,3,11,12], which can be grouped into two classes, nontoken-based and token-based. The performance of DMX algorithms is generally measured by two metrics: first, the message complexity, which is the number of messages necessary per *CS* invocation; second, the synchronization delay, which is the time required after a process leaves the *CS* and before the next process enters the *CS*. The token-based algorithms can reduce a message complexity to *O(logN)*, and their synchronization delay can reduce to *O(logN)T*. The nontoken-based algorithms generally have a message complexity between *N* and *3(N-1)*, and have a synchronization delay between *T* and *2T*.

The proposed algorithm belongs to nontoken-based type. Based on Ricart-Agrawala(*RA*) algorithm[8], it reduces the message complexity and keeps the same synchronization delay as *RA* algorithm. The algorithm provides different methods for reading mutual exclusion and writing mutual exclusion to reduce the message complexity. According to the Lamport globe clock stamp, a read/write

X. Zhou et al. (Eds.): APPT 2003, LNCS 2834, pp. 75–84, 2003.

globe clock stamp suitable for read/write mutual exclusion, is presented. Subnumbering for reading requests, is brought in to avoid deadlock and starvation.

Many DMX algorithms try to reduce message complexity [6,7,8], that, maybe lead to a less stable system. So these algorithms are always under the assumption: the processes communicate through the asynchronous message passing over an error-free underlying communication network, while the message transit times may vary. Adopting a reply and timeout mechanism, the proposed algorithm is self-stability[4,5,10]. It can tolerate transient failures of communication network, and be more reliable and practical.

The structure of the paper is arranged as following: Section 2 introduces the system model of *RA* algorithm; Section 3, the proposed algorithm is presented in details. The performance analysis of the algorithm is given in Section 4, and in Section 5 conclusions of the algorithm are provided.

2 *RA* Algorithms Review

In this section, the general system model of *RA* algorithm, which is the basis of the proposed algorithm, is described.

2.1 System Model

The *RA* algorithm runs under the following system model. There are N nodes in the system, which are connected with a communication network. The system has no shared memory, that, the nodes exchange information only via message transfer. The network guarantees error-free *FIFO* delivery with bounded message delay.

Two types of messages are exchanged among nodes: *REQUEST* and *REPLY* . While wanting to enter *CS*, a node sends the *REQUEST* messages to other nodes and waits for their *REPLY*. Each *REQUEST* for per *CS* invocation is assigned a priority *p*, which is implemented by the Lamport globe clock stamp[7]. To achieve fairness and to prevent deadlock and starvation, the *REQUESTs* should be ordered by priority for per *CS* invocation.

Define *p* as *p=(SN,SiteID)*, where *SN* is a unique locally assigned sequence number to the request and *SiteID* is the process identifier. *SN* is determined as follows. Every node maintains the highest sequence number seen so far in a local variable *Highest-M-Seen*. While making a request, a node uses a sequence number which is one larger than the value of *Highest-M-Seen*. When a *REQUEST* is received, *Highest-M-Seen* is updated as follows.

Highest-M-Seen=Maximum (Highest-M-Seen, sequence number in the REQUEST)

Priorities of two *REQUESTs* , p_1=$(SN_1, SiteID_1)$ and p_2=$(SN_2, SiteID_2)$ are compared as follows. Priority of p_1 is greater than priority of p_2 if $SN_1 < SN_2$ or $(SN_1 = SN_2$ and $SiteID1 < SiteID2)$.

Definition 1. S_i and S_j are concurrent if S_i's *REQUEST* is received by S_j after S_j has made its *REQUEST*, and S_j's *REQUEST* is received by S_i after S_i has made its *REQUEST*. Where S_i represents node i.

2.2 Ricart-Agrawala Algorithm

Each node S_i uses the following local integer variables: my-seq-m$_i$, replycount$_i$, highest-m-seen$_i$, and also uses the following vectors:

RD$_i$[1:N] of Boolean. *RD*$_i$[j] indicates if S_i has deferred the *REQUEST* sent by S_j

The *RA* algorithm is outlined in Fig.1. The *REPLY* messages sent by a process are blocked only by processes which request the *CS* with higher priority. Thus, when a process sends *REPLY* messages to all deferred requests, the process with the next highest priority request receives the last needed *REPLY* message and enters the *CS*. The execution of *CS* requests in this algorithm is always ordered by decreasing priority. For each *CS* invocation, there are exactly *2(N-1)* messages: *(N-1) REQUESTs* and *(N-1) REPLYs*.

1) *Initial local state for node S_i*
   - *int my-seq-m$_i$ = 0*
   - *int replycount$_i$ = 0*
   - *int array of Boolean RD$_i$[j] = 0, $\forall j \in \{1...M\}$*
   - *int highest-m-seen$_i$ = 0*
2) *InvMulEx: S_i executes the following to invoke mutual exclusion*
   - *my-seq-m$_i$ = highest-m-seen$_i$ + 1*
   - *Make a REQUEST(P_i) message: where P_i = (my-seq-m$_i$,i)*
   - *Send REQUEST(P_i) message to all the other processes*
   - *replycount$_i$ = 0*
   - *RD$_i$[k] = 0, $\forall k \in \{1...N\}$*
3) *RcvReq: node S_i receives message REQUEST(P_j), where P_j = (SN_j, j), from node S_j*
   - *a. If S_i is requesting then there are two cases*
     - *S_i's REQUEST has a higher priority than S_j's REQUEST. In this case, S_i sets RD$_i$[j] = 1 and highest-m-seen$_i$ = max(highest-m-seen$_i$, SN_j)*
     - *S_i's REQUEST has a lower priority than S_j's REQUEST. In this case, S_i sends REPLY to S_j*
   - *b. If S_i is not requesting then send REPLY to S_j*
4) *RcvReply: node S_i receive REPLY from node S_j*
   - *replycount$_i$ = replycount$_i$ +1*
   - *if (CheckExecuteCS) then execute CS*
5) *FinCS: node S_i finish executing CS*
   - *Send REPLY to all nodes S_k, such that RD$_i$[k]=1*
6) *CheckexecuteCS*
   - *If (replycount$_i$ = N-1), then return true else return false*

Fig 1. Ricart-Agrawala algorithm

3 The Proposed DMX Algorithm

3.1 Basic Idea

The system model that the proposed algorithm requires is almost the same as the section 2. But it is self-stability, and is able to recover from transient errors by itself. It is realized by lock mechanism. A process must get the lock of a CS before entering it, and release the lock after finishing to execute the *CS*. Each node S_i runs a lock module which is realized by the proposed algorithm. The lock module has no independent threads, and some skills such as sleeping processes and waking up processes are used in it. The module has two queues. One is Request Queue(RQ), which maintains all the requests that have not been finished. Another is Lock Queue(LQ), which maintains all the locks in the system.

There are two types of net locks, Reading(R) lock and Writing(W) lock. A process requests a R-lock when only executing reading operations in *CS*, while it requests a W-lock when executing writing operations in *CS*. No mutual exclusion requirements are required between R-locks. But R-lock and W-lock, W-lock and W-lock need do mutual exclusion operations. A R-lock is recorded at local while a W-lock is recorded at all the nodes in the system.

Four types of messages are used in the proposed algorithm: *REQUEST*, *REPLY*, *CREPLY* and *RELEASE*. The functions of *REQUEST* and *REPLY* are the same as their functions in *RA* algorithm. *CREPLY* serves as a collective reply and *RELEASE* serves as a release request message. Their functions will describe in detail in the following context.

The proposed algorithm refers to the idea of the Lodha, S. and Kshemkalyani, A. (*LK*) algorithm[9] to reduce the message complexity. But *LK* algorithm does not consider that reading and writing requests have different requirements for mutual exclusion. In the proposed algorithm, the idea is as follows. Without loss of generality, S_i represents site i, and P_i represents a process which is running at S_i, and R_i represents a *REQUEST* message which is sent by P_i.

When P_j receives R_i for a lock, at the same time P_j wants the same lock and sends R_j. If R_j has a higher priority, then R_i serves as a reply to P_j and P_i need not send *REPLY* to P_j. If P_j has a lower priority, then R_j serves as a reply to P_i. So, when there are concurrent requests, *REQUEST* messages can serve as *REPLY* to reduce message complexity.

When P_i receives R_j for a lock while it does not want the lock, it sends *REPLY* to P_j. When exiting *CS* and releasing the lock it has got, P_i will select the next highest priority R_k, which is not replied, from the requests set, and send *CREPLY* to P_k as a collective reply from all processes that had made higher priority requests than R_k. When receiving *CREPLY*, that means all the requests, which with higher priorities than R_k, have finished to access *CS*, and P_k can delete the requests. From this point, the number of *REPLY* is reduced.

To reduce the message complexity and design complexity, the reading requests are not sent to other nodes. But that will bring some problems as follows. If P_i has only reading requests, then there are only local reading requests and remote writing requests in RQ_i. Because other nodes have not $P_i's$ reading requests, no node will send *CREPLY* to P_i when it releases lock, and the other nodes' requests will exist in RQ_i forever. So these requests in RQ_i will not be processed and their owner will wait for the reply forever. To avoid the case, while exiting a *CS*, P_i sends *RELEASE* messages to the nodes which have not messages in RQ_i, that, means the nodes have not writing request. When receiving the *RELEASE* messages, the nodes can delete corresponding requests R_i from local RQ.

3.2 Read/Write Globe Clock Stamp

When generating a request, the priority of it is also generated. *RA* algorithm uses the Lamport clock stamp to define the priority of a request. For the proposed algorithm, while the reading requests not being sent to all the nodes, some abnormal cases as follows will happen if using the Lamport clock stamp.

Assuming P_i has many continuous reading requests but no writing requests, and other nodes have a few requests. By reading requests not spreading to net, while using Lamport clock stamp, the *SN* of P_i is large but the other processes in other nodes have small *SN*. When a writing request is generated on other nodes after $P_i's$ reading requests, it's priority may be higher than $P_i's$ reading request which is generated earlier. To avoid this, the priority of a request is implemented by the read/write globe clock stamp.

Definition 2. read/write globe clock stamp is defined as $(SN_w, SN_r, SiteID)$, the SN_w is the same as *SN* in Lamport clock stamp. SN_r is the supplement serial number. All the continuous reading requests between two writing requests have the same SN_w, and they use different SN_r to represent their different priorities.

Priorities of two requests
p_1=(SN_w1,SN_r1,$SiteID_1$) and p_2=(SN_w2,SN_r2,$SiteID_2$) are compared as follows.

If $SN_w1 < SN_w2$ or ($SN_w1 = SN_w2$ and $SiteID_1 < SiteID_2$) or ($SN_w1 = SN_w2$ and $SiteID_1 = SiteID_2$ and $SN_r1 < SN_r2$) then $p_1 > p_2$.

3.3 Realization of a Self-Stability System

In order to realize DMX, all the nodes must negotiate through communication network. If a message is abnormal when negotiating, the request process may wait forever and deadlock happens. So, in the proposed algorithm, a reply and timeout mechanism is used to implement the reliability and avoid deadlock.

When sending a message, two timeout clocks are generated, $t_1 = \tau, t_2 = k\tau$.

If a message is sent but does not receives its reply after t_1, it is considered as lost and will be sent again per t_1 interval, until getting the reply of it or sent for k times. If it has been sent for k times while not getting the reply, the destination node is considered as failed node, and will be deleted from the group. So, the source node will not wait a reply from it. Apparently, transient failures not exceeding t_2 can be tolerated by the mechanism.

When recovering, a node sends initial messages to all other nodes, so the other nodes can add it into the group. Using this mechanism, the system can tolerate transient failures in the communication network and implementation automatic recovering.

3.4 Implementation of the Proposed Algorithm

Declaration and Data Structure. The lock of a *CS* is marked as $L_{(j,t,m)}$, $t \in$ {r,w} is the state of a lock, *t=r* represents a *R*-lock which is hold by a reading process, and *t=w* represents a *W*-lock which is hold by a writing process. For *R*-lock, m represents the share number.

To realize fairness, each process has a priority p which is implemented by read/write clock stamp. Assuming P_i wants to access CS_j, it will generate a request R_i, marked as $R_{(i,j,p,t,s)}$, where $t \in$ {r,w} and $s \in$ {l,n}. For a reading request *t=r* and for a writing request *t=w*. The symbol s represents the state of a request. It may be local block state(l) or net block state(n). L state represents the request having the qualification to get the lock but it will wait until the other local process, which has hold the lock, to release it, and n state represents that the request is waiting for the nodes negotiating to permit the qualification to get the lock.

A reading request $R_{(i,j,p,r,l)}$ is not sent to net, so *s=l*. $R_{(i,j,p,r,l)}$ is inserted into RQ_i. If there is no other concurrent request for the lock of CS_j, the requester can hold the lock. Otherwise while the lock released by other process and $R_{(i,j,p,r,l)}$ becoming the highest priority request, the requester will hold the lock of CS_j.

A writing request $R_{(i,j,p,w,n)}$ must be sent to all the nodes in the system. Firstly, $R_{(i,j,p,w,n)}$ is inserted into RQ_i and waits for net negotiating to permit the qualification to get the lock. When all the nodes approve the requester to get the lock, the requester can get the lock of CS_j if there is no other higher priorities requests for the lock, otherwise changing $R_{(i,j,p,w,n)}$to $R_{(i,j,p,w,l)}$ and waiting for the lock being freed. When the lock is released and $R_{(i,j,p,w,l)}$ becomes the highest request in RQ_i, the requester will hold the lock of CS_j.

For *REPLY* or *CREPLY*, marked as $REP_{(i,j,p)}$ or $CREP_{(j,p)}$, where i represents the source process, and j represents to request the lock of CS_j, and p represents the priority of corresponding request being replied. When receiving $REP_{(i,j,p)}$, the receiver process checks whether all the replies of the corresponding requests are received. If so, it changes the corresponding request's state from n to l . When receiving $CREP_{(j,p)}$, the receiver deletes all the requests whose priorities are higher than p from RQ and changes the corresponding request's state from n to l.

RELEASE is marked as $REL_{(i,p)}$, where i represents the source process, p represents the priority of corresponding request. When receiving $REL_{(i,p)}$, a process deletes the corresponding request.

Algorithm Implementation. The proposed algorithm is composed of applying *R*-lock, releasing *R*-lock, applying *W*-lock, releasing *W*-lock, receiving

REQUEST, receiving *REPLY*, receiving *CREPLY*, receiving *RELEASE* procedures.

Define the requests set for CS_j in RQ_i as $\Omega_j=\{R \mid R \in R_{(k,j,p,t,s)}$, k $\in [1...N]\}$.

Define the nodes set $\omega_j = \{S_k \mid S_k$ is active, k $\in [1...N]\}$.

(1) Applying R-lock. *While wanting to enter CS_j and doing reading operations, P_i applies R-lock.*

- *1. Generates $R_{(i,j,p,r,l)}$ and inserts it into RQ_i.*
- *2. If $L_{(j,t,m)}$ exists in LQ_i goes to 3, else goes to 6.*
- *3. If $L_{(j,t,m)}$ is W-lock, blocks P_i on $R_{(i,j,p,r,l)}$, when it is waked up, goes to 2.*
- *4. If a request $R_{(k,j,p',w,l)}$ exists in RQ_i where $\forall k \in [1...N]$,and $p' > p$, then blocks P_i on $R_{(i,j,p,r,l)}$, when it is waked up, goes to 4.*
- *5. m=m+1, P_i holds the R-lock, return.*
- *6. If a request $R_{(k,j,p',w,l)}$ exists in RQ_i, where $k \in [1...N]$,and $p' > p$, then blocks P_i on $R_{(i,j,p,r,l)}$,when it is waked up, goes to 6.*
- *7. If there is no lock $L_{(j,r,m)}$ in LQ_i, generates $L_{(j,r,m)}$, P_i holds $L_{(j,r,m)}$, return.*

(2) Releasing R-lock. *While leaving CS_j which is entered by reading mode, P_i will release R-lock*

- *1. When P_i releases $L_{(j,r,m)}$, deleting the corresponding request $R_{(i,j,p,r,l)}$ from RQ_i, m=m-1, if $m \neq 0$ then return.*
- *2. If $\Omega_j=\phi$ then deletes $L_{(j,r,m)}$ from LQ_i, return.*
- *3. Get the highest priority $R_{(k,j,p',w,s)}$ from Ω_j, while it is a writing request, if k=i then goes to 4 else goes to 5.*
- *4. Changes $L_{(j,r,m)}$ to $L_{(j,w,m)}$, and wakes up the blocked process on it, return.*
- *5. Changes $L_{(j,r,m)}$ to $L_{(j,w,m)}$ and sends $CREP_{(i,p')}$ as the reply of $R_{(k,j,p',w,n)}$ to P_k.*

(3) Applying W-lock. *While wanting to enter CS_j and doing writing operations, P_i will apply W-lock*

- *1. Generates $R_{(i,j,p,w,n)}$.*
- *2. Sends $R_{(i,j,p,w,n)}$ to the nodes set ω_j except local node, and inserts it into RQ_i, blocked P_i on $R_{(i,j,p,w,n)}$. When it is waked up, goes to 3.*
- *3. P_i gets $L_{(j,w,m)}$, return.*

(4) Releasing W-lock. *While leaving CS_j which is entered by writing mode, P_i will release R-lock*

- *1. When P_i releases $L_{(j,w,m)}$, the corresponding $R_{(i,j,p,w,l)}$ is deleted from RQ_i. If $\Omega_j = \phi$ then deletes $L_{(j,w,m)}$ from LQ_i and goes to 6, else goes to 2.*
- *2. Getting the highest priority request $R_{(k,j,p',t,s)}$ from Ω_j, if $k = i$ then goes to 3, else goes to 5.*
- *3. If it is writing request $R_{(k,j,p',w,l)}$, wakes up the process blocked on it, return, else goes to 4.*

- *4. If no writing request in Ω_j, changes $L_{(j,w,m)}$ to $L_{(j,r,m)}$,wakes up all the blocked processes blocked on the reading requests, goes to 6. If there is writing request in Ω_j, supposing the highest priority writing request is $R_{(k',j,p'',w,s)}$, then changes $L_{(j,w,m)}$ to $L_{(j,r,m)}$, and wakes up all the processes which are blocked on the reading requests whose priority $p > p$", goes to 6.*
- *5. Sends $CREP_{(i,p')}$ as the reply of $R_{(k,j,p',w,n)}$ to P_k*
- *6. Selects the nodes $S_k(k \neq i)$ who has no request in RQ_i.*
- *7. Sends $REL_{(i,p)}$ to S_k, return.*

(5) Receiving REQUEST

- *1. When P_j receives $R_{(i,j,p,w,n)}$, inserts it into RQ_j.*
- *2. If there is no local request $R_{(j,j,p',t,s)}$ in RQ_j, P_j sends $REP_{(j,p)}$ to P_i as a reply to $R_{(i,j,p,w,n)}$,return.*
- *3. Assuming $R_{(j,j,p',t,s)}$ is the highest priority request in Ω_j, if $p' < p$ then goes to 4 else goes to 5.*
- *4. If t=w, it means that $R_{(j,j,p',t,s)}$ has been sent to P_i, this message should be regard as the reply, return. If t=r, sends $REP_{(j,p)}$ message to P_i as the reply to $R_{(i,j,p,w,n)}$,return.*
- *5. If t=w, $R_{(i,j,p,w,n)}$ will be regarded as the reply from P_i to $R_{(j,j,p',t,s)}$,goes to (6). If t=r, return.*

(6) Receiving REPLY or CREPLY

- ***P_i receives $CREP_{(j,p)}$***
  - *1. Sets the receiving mark that $REP_{(i,j,p)}$ has been received.*
  - *2. If P_i receives $REP_{(k,j,p)}(1 \leq k \leq N)$ from all other nodes then changes $R_{(i,j,p,w,n)}$ to $R_{(i,j,p,w,l)}$, goes to next step, otherwise return.*
  - *3. If $R_{(i,j,p,w,l)}$ is not the highest priority request in Ω_j, return.*
  - *4. Generates $L_{(j,w,m)}$, inserts it into LQ_i, wakes up the processes blocked on $R_{(i,j,p,w,l)}$, return.*
- ***P_i receives $CREP_{(j,p)}$***
  - *1. Removes $R_{(k,j,p',w,l)}$ from Ω_j whose priority $p' > p$,$k \in [1...N]$.*
  - *2. Changes $R_{(i,j,p,w,n)}$ to $R_{(i,j,p,w,l)}$.*
  - *3. If $R_{(i,j,p,w,l)}$ is the highest priority request in Ω_j, then generates $L_{(j,w,m)}$, inserts it into LQ_i, wakes up processes blocked on $R_{(i,j,p,w,l)}$, return.*
  - *4. Generates $L_{(j,r,m)}$,m=0,inserts it into LQ_i, wakes up the processes blocked on the $R_{(i,j,p',r,l)}$ whose priority $p' > p$, return.*

(7) Receiving RELEASE

- *1. While receiving $REL_{(j,p)}$, P_i removes $R_{(i,j,p,w,n)}$ from RQ_i.*
- *2. If $\Omega_j = \phi$, deletes $L_{(j,t,m)}$ from LQ_i, return, else gets the highest priority request $R_{(k,j,p',t,s)}$ from Ω_j, if k=i goes to next step, else return.*
- *3. As assumption, $R_{(k,j,p',t,s)}$ must be reading request, so generates $L_{(j,r,m)}$, m=0,inserts it into LQ_i. If no writing request in Ω_j, wakes up all the processes blocked on the messages in RQ_i, return. Otherwise gets the highest priority write request $R_{(k',j,p'',w,s)}$ from Ω_j, wakes up all the processes blocked on the messages whose priority is higher than $R_{(k',j,p'',w,s)}$,return.*

Fig 2. The proposed algorithm

4 A Performance Analysis

Definition 3. P_i and P_j are concurrent if R_i is received by P_j after P_j has made R_j, and R_j is received by P_i after P_i has made R_i. $CSet_i$=$\{R_j \mid R_j$ is concurrent with $R_i\} \cup \{R_i\}$.

Traditionally, the performance of DMX algorithms is compared on the basis of synchronization delay and the message complexity. In the proposed algorithm, the synchronization delay is an average message delay which is the same as that of *RA* algorithm, while message complexity is reduced.

The proposed algorithm needs additional communication spending to realize self-stabilizing. In order to compare the performance at the same condition, the proposed algorithm's performance is evaluated without considering the communication spending to realize self-stabilizing.

1) For writing requests
   The request process P_i will send *(N-1)* requests $R_{(i,j,p,w,n)}$, and receives $(N-\mid CSet_i \mid)$ replies.
   - $\mid CSet_i \mid \geq 2$, there are two cases here
     (1) There is at least one writing *REQUEST* whose *p'*$<$ *p* in $CSet_i$, and the number of nodes who only has reading *REQUEST* is *k*. So, P_i will send one *CREPLY* and *k RELEASE* messages. The message complexity is $2N- \mid CSet_i \mid +k$. When all *REQUESTs* are concurrent, there are $N + k (0 \leq k < N)$ messages.
     (2) There is no writing request whose $p' < p$ in $CSet_i$, and the number of nodes which only have reading request is *k*. So, P_i will not send *CREPLY*. The message complexity is $2N - 1- \mid CSet_i \mid +k$. When all requests are concurrent, there are $N + k - 1 (0 \leq k < N)$ messages.
   - $\mid CSet_i \mid= 1$.This is the worst case, implying that all requests are serialized. In this case the message complexity is *2(N-1)*, same as *RA* algorithm.
2) For reading requests
   The requester will generate reading request R_i but not send it to other nodes, the message complexity is as follows.
   - $\mid CSet_i \mid \geq 2$,there are two cases here
     (1) There is at least one writing request whose $p' < p$ in $CSet_i$. So,P_i will send one *CREPLY*. The message complexity is 1.
     (2) There is no writing request whose $p' < p$ in $CSet_i$. So P_i will not send *CREPLY*. The message complexity is 0.
   - $\mid CSet_i \mid= 1$. Imply that all requests are serialized. In this case the message complexity is 0.

5 Conclusions

In this paper, a high efficiency and self-stabilized DMX lock algorithm is presented. It can recover from transient failures of the system. Unlike most DMX algorithms, different DMX methods for reading and writing operations, is used to reduce the message complexity and system design complexity. A new globe clock stamp—read/write clock stamp, which is suitable for the proposed algorithm, is presented. The algorithm is proved to be high-performance by performance analysis.

References

1. Y.-I. Chang, "A Simulation Study on Distributed Mutual Exclusion" J. Parallel and Distributed Computing, vol. 33,pp. 107–121, 1996
2. M.Singhal, "A taxonomy of Distributed Mutual Exclusion" J.Parallel and Distributed Computing, vol. 18, no.1, pp.94–101, May 1993
3. Raudal C.Burns. "Semi-Preemptible Locks for a Distributed File System" Performance, Computing, and Communications Conference, 2000. IPCCC'00. Conference Proceeding of the IEEE International , 2000 pp. 397–404
4. Mizuno, M.; Nesterenko, M.; Kakugawa, H. "Lock-based self-stabilizing distributed mutual exclusion algorithms" Distributed Computing Systems, 1996, Proceedings of the 16th International Conference on, 1996, pp. 708–716
5. Dijkstra, E.W. "Self-stabilizing systems in spite of distributed control" Communications of the ACM, 17(11),pp. 643–644, November 1974.
6. O. Carvalho and G. Roucairol, " On Mutual Exclusion in Computer Networks" Technical Correspondence, Comm. ACM, vol. 26, no. 2, pp. 146–147, Feb. 1983.
7. L. Lamport, Time, "Clocks and the Ordering of Events in Distributed Systems" Comm. ACM, vol. 21, no. 7, pp. 558–565, July 1978.
8. G. Ricart and A. K. Agrawala, "An Optimal Algorithm for Mutual Exclusion in Computer Networks" Comm. ACM, vol. 24, no. 1, pp. 9–17, Jan. 1981.
9. Lodha, S.; Kshemkalyani, A. "A fair distributed mutual exclusion algorithm" Parallel and Distributed Systems, IEEE Transactions on , Volume. 11, Issue. 6 , June 2000, pp. 537–549
10. Gouda, M.G.; Multari, N.J. "Stabilizing communication protocols" Computers, IEEE Transactions on , Volume.40, Issue.4 , April 1991, pp. 448–458
11. Sanders, B. "The information structure of distributed mutual exclustion algorithms" ACM Transactions on Computer Systems, 5(3),pp.284–299, 1987
12. Raynal,M. "A simple taxonomy for distributed mutual exclustion algorithm." ACM Operating Systems Review, 25(2),pp.47–50, 1991