International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:1, No:8, 2007

A Distributed Group Mutual Exclusion Algorithm for Soft Real Time Systems

Abhishek Swaroop, and Awadhesh Kumar Singh

Abstract**—**The group mutual exclusion (GME) problem is an interesting generalization of the mutual exclusion problem. Several solutions of the GME problem have been proposed for message passing distributed systems. However, none of these solutions is suitable for real time distributed systems. In this paper, we propose a token-based distributed algorithms for the GME problem in soft real time distributed systems. The algorithm uses the concepts of priority queue, dynamic request set and the process state. The algorithm uses first come first serve approach in selecting the next session type between the same priority levels and satisfies the concurrent occupancy property. The algorithm allows all n processors to be inside their CS provided they request for the same session. The performance analysis and correctness proof of the algorithm has also been included in the paper.

Keywords**—**Concurrency, Group mutual exclusion, Priority, Request set, Token.

I. Introduction

THE design of protocols for distributed real time systems is more challenging than that for normal distributed systems because the real time systems must satisfy stringent response time constraints in addition to the logical correctness of the system. Nevertheless, the distributed systems are emerging as a highly promising candidate for implementing the next generation of high performance real time systems. However, the distributed system must be fine tuned before they can be used to monitor and control critical real time systems. The real time systems (RTS) are generally classified as soft real time systems (SRTS) and hard real time systems (HRTS) [22]. In the soft real-time systems, the utility of the system goes down with every unit of time elapsed after missing the deadline. However, missing a deadline does not lead to catastrophic system failure in SRTS. The hard real-time systems are those in which the utility of a system becomes zero in the event of a missed deadline and missing a deadline could lead to a catastrophic system failure. Although, both paradigms namely, shared memory and message passing exist, we have considered the message passing systems only.

Resource sharing is an important aspect of the real time distributed systems. Some resources are inherently non shareable and must be accessed in a mutually exclusive way. Many algorithms exist in the literature to solve the mutual exclusion problem [1, 10, 18, 19, 20] in message passing distributed systems. Some of these algorithms have been fine tuned to suits the needs of real time systems in [12, 13, 14, 15, 16, 21].

In [2] Joung proposed group mutual exclusion (GME) problem as generalization of classical mutual exclusion problem, and modeled it as congenial talking philosophers (CTP) problem. In group mutual exclusion a process request a session (alternatively called forum), before entering its Critical Section (CS), processes requesting for the same session are allowed to be in their CS simultaneously. However, processes requesting for different sessions must do so in a mutually exclusive way. The readers-writer problem can be considered as a special case of GME problem. In order to achieve this, we can use a common read session for all processes and a unique write session for each individual process.

The requirements for group mutual exclusion problem are:

Mutual exclusion: No two processes, requesting for a different session can be in their critical sections concurrently.

Starvation Freedom: A process attempting to attend a session will eventually succeed.

Concurrent Occupancy: If some process *P,* has requested for a session *X* and no other process is currently attending or requesting a different session, then *P* can attend *X* without waiting for any other process to leave the session.

The first algorithm for GME problem was given by Joung [2] for shared memory model. In [3] Joung proposed two algorithms RA1 and RA2 based on Ricart - Agrawala algorithm [10] to solve GME problem for message passing systems. Several non token-based algorithms for GME problem have been proposed in the literature [3,6,7,8]. Token-based algorithms for GME problem have been presented in [4,9,11,17]. However, none of these algorithms is suitable for real time systems. Mittal–Mohan algorithm [4] considers the concept of priority in selecting the next session type. However, the priority of a session is decided by the number of processes willing to attend that session. In Mittal-Mohan algorithm a requesting process can not assign priority to a request. Therefore, in its present form Mittal-Mohan algorithm can not be used for real time distributed systems.

The paper presents a token- based algorithms for solving GME problem for soft real time systems (SRTS). Our algorithm is based upon the concept of dynamic request sets.

Abhishek Swaroop is with the Department of Computer Science & Engineering of G.P.M. College of Engineering, Delhi 110036 India (phone: +91-11-22300003; fax: 91-11-27203937; e-mail: abhi_pu1@ yahoo.co.in).

Awadhesh Kumar Singh is with the Department of Computer Engineering, National Institute of Technology, Kurukshetra 136119, India (e-mail: aksinreck@rediffmail.com).

The concept have been used earlier also [1 18], but to handle some other problem that is comparatively simple. In the proposed scheme, a captain process is responsible for the session initiation and sending start message to other processes requesting for the same session, called followers, in order to allow them to enter in CS.

The rest of the paper is organized as follows. We describe the system model and assumptions in section 2, the data structures and the messages used in our algorithm are explained in section 3 and the description of the algorithm is given in section 4. The correctness proof and performance analysis of the algorithm are given in section 5 and section 6 respectively. The concluding remarks are given in section 7.

II. The System Model

We assume an asynchronous distributed system. The system has N sites, numbered as 1,2,3,..,N. The sites do not share any memory or global clock, and the only way of communication between sites, is through message passing. The system is fully logically connected, i.e. every site can send message to every other site. We assume that, at each site i, there exists exactly one process P_i. Once a process has requested for a session, it will not make new requests unless the old request is serviced. Each process P_i also announces its priority Z_i while requesting a session. A higher value of Z_i indicates higher priority level. The lowest priority level is one and the highest priority level is K.

III. Notations

Each process may be in any one of the following 6 states:

(i) *R*- requesting for a session.

(ii) *N*- not requesting.

(iii) *EC*- process is executing in its CS as captain.

(iv) *EF*- process is executing in its CS as follower.

(v) *HI*- process is holding token because, no pending request is there.

(vi) HS- process is holding token because, some followers are still in their CS.

Every process P_i stores following local variables-

state$_i$ - stores the current state of process P_i.

RS$_i$ - stores the ids of all the processes, to which P_i must send its request, in case it wishes to attend a session and not possessing the token.

SN$_{i,}$ - where $SN_i[j]=k$ denotes that P_i knows about k requests made by P_j.

captain$_i$ - stores the id of the captain of the current session, if P_i is in its CS as follower. Otherwise $captain_i$ is set to *NULL*.

The token in our algorithm contains following variables-

token.queue - *token.queue* is a priority queue to store all pending requests. The requests for the same session are grouped together, and are treated as single entry in the queue. A priority level is associated with each entry in *token.queue*. The priority level of an entry is assigned equal to the priority of the highest priority process, requesting for the session, associated with the entry. The entry with highest priority level always remains at the head of the *token.queue*.

token.type - stores the type of the current session

token.followers - stores the number of follower processes still in their CS.

In our algorithm various messages are exchanged among processes in order to solve GME problem. We briefly describe each message.

***request* (i,SN_i,X,Z_i)** – When a process P_i wishes to attend a session X with priority Z_i , and P_i is not holding the token then it sends a *request* message containing its id, sequence number of request, type of session requested and the priority of the process P_i to all processes in its request set

start (*i*) - *start* message is sent to a process to allow it to enter in CS as follower of P_i.

complete (i) - When a process P_i executing in its CS as follower, comes out of CS, it sends a *complete* message to its captain.

token (token.queue,token.type,token.followers) - A unique *token* exists in the system and only the process holding the *token* can enter in its CS as captain. Whenever a session finishes and next session is selected, the *token* is passed to the new captain.

IV. Description of the Algorithm

The complete pseudo code of our algorithm is given in Appendix A; however, brief description of the algorithm is given in this section. Initially all processes are in state *N*, having their captain as *NULL*, all entries of *SN* are zero and the Request set of each process contains ids of all other processes except itself. Only exception is process P_1. We assume that P_1 holds the token initially, therefore, the variable $state_1$ is set to *HI* and RS_1 is initialized to empty set.

A process P_i wishing to attend a session X with priority Z_i and not possessing the token, sends its request to all members in its request set, changes its state to *R* and waits for the token or *start* message. Upon receiving the token, P_i initiates a new session and enters in its CS as captain along with its followers. In case P_i receives a start message, it enters in its CS as a follower. If P_i possesses an idle token it enters in its CS as captain. However, if P_i is holding token in *HS* state, it enters in its CS again only if the requested session is the same as the current session and the *token.queue* is empty. Otherwise, the request is added in token.queue.

A procedure **add_request**(i,X,Z_i) is called to accommodate a new request in *token.queue* according to its priority level and the requested session, where i is the id of the requesting process, X is the session requested and Z_i is the priority level of the request. There exists only one entry for a session. Also, a priority level is associated with each entry in *token.queue*. If the entry for the requested session X already exists in *token.queue*, P_i is also added in the list of processes requesting for session X. If the priority level of the newly arrived request is greater than the priority level of the entry for session X, the priority level of the entry for session X is set

equal to the priority of the newly arrived request. After that the entry for session X is moved forward according to its new priority level in *token.queue*. On the other hand, if there is no entry corresponding to session X then a new entry is created and added in the *token.queue* according to its priority level.

When a process P_i receives a request (j,SN,X,Z_i), it discards the old request without taking any action. However, if the request is new, P_i updates the value of $SN_i[j]$. If P_j is not in its request set, P_i adds P_j in its request set. P_i also sends a request to P_j, if it is requesting for a session. If P_i is holding an idle token, it immediately sends it to P_j. However, if P_i is holding token in state *EC* or *HS*, it passes a *start* message to P_j only if *token.queue* is empty and the session requested is the same as the current session. Otherwise procedure **add_request** is called to add the request of P_j is in *token.queue*.

When a follower process comes out of its CS, it sends a *complete* message to its captain; it changes its state to *N* and sets its captain to *NULL*. However, when a captain process comes out of its CS, it checks the number of followers still in CS. If there are still some follower processes in their CS, the captain changes its state to *HS*. If no follower process is in CS and no pending requests are there, the captain process changes its state to *HI*. However, if there are pending requests in the *token.queue*, the captain process changes its state to *R* or *N* , depending upon whether its request is in *token.queue* or not, removes next captain and its followers from the *token.queue*, sends token to the next captain, and sends *start* messages to all followers. Before sending the token to the next captain the priority level of all entries in *token.queue* are incremented by one.

Upon receiving a *complete* message the captain decrements the variable *token.followers* by one. If the state of the captain is *HS* and *token.followers* is zero, the captain changes its state to *HI* if *token.queue* is empty. However, if *token.queue* is not empty, the captain process changes its state to *R* or *N* depending upon whether its request is in *token.queue* or not, removes the next captain and its followers from *token.queue*, sends *token* to the next captain, and sends *start* messages to all followers. The priority level of each entry is also incremented by one before transferring the token in order to remove the possibility of starvation.

The captain process on receiving *token* changes its state to *EC* and enters in its *CS*. Upon receiving a *start* message, a process changes its state to *EF*, sets the variable captain and enters in its CS.

V. Correctness of the Algorithm

In this section we will show that our algorithm satisfies all requirements, which are necessary for a solution of group mutual exclusion problem.

A. Safety

The mutual exclusion requirement in GME problem says that, no two processes requesting for a different session, must be in their CS simultaneously. There exists only one *token* in the system, and only the process holding the *token* can initiate a session as a captain. The process holding the *token* can send the *start* message to only those processes requesting for the same session. Further the *token* is not transferred to another process, until the current captain and all its followers have come out of their CS. Therefore, no two processes requesting for a different session, can be in their CS at the same time.

B. Freedom from Starvation

A priority queue is associated with the *token* to store the pending requests. A priority level and a session type are associated with each entry in the *token.queue*. The entry with the highest priority level is always at the front of the *token.queue* in order to favor sessions associated with higher priority levels. However, an FCFS approach is used to select a session, among sessions having same priority levels. Further, the priority of long waiting processes is gradually enhanced using the idea of *aging* [Silberschatz] in order to completely remove the possibility, if any, of starvation. Whenever a new session is selected the priority level of all sessions, whose requests are stored in *token.queue*, is incremented by one. Therefore, the process having lowest priority level will also be able to attain highest priority level after *K*-1 session switches.

If a request for the current session type arrives at the captain, it first checks whether the *token.queue* has any pending requests or not. The captain sends *start* message to the requesting process, only if the *token.queue* is empty. However, if the *token.queue* is not empty, the request is added in the *token.queue*. This entry policy reduces the concurrency and hence the resource utilization, however, it removes the possibility that the processes of a particular group keep on requesting for the currents session and not allowing other processes to enter in their critical sections. Therefore, we can say that, the sessions in our algorithm are served in a starvation free manner.

C. Concurrent Occupancy

In the proposed algorithm, when a process starts a session as a captain, it captures all the processes (requesting for the same session), whose requests are stored in the *token.queue*, at the time of entry in its CS. When the captain process is in state *EC* or state *HS* and a request for the current session arrives, it checks whether the *token.queue* is empty or not. If the *token.queue* is empty, it immediately sends a *start* message to the requesting process. The requesting process enters in its CS upon receiving the *start* message. Hence, it is proved that our algorithm satisfies the concurrent occupancy property.

VI. Performance Analysis of the Algorithm

In this section we analyze the performance of our algorithms using following performance parameters: message complexity /CS request, average message size, forum switch complexity, maximum concurrency, and synchronization delay. Forum switch complexity [2] and maximum concurrency are applicable only for GME algorithms, not for mutual exclusion algorithms.

Message complexity: The messages exchanged, during the

execution of the algorithm are, *request*, *token*, *start* and *complete*. The *request* messages are sent by a requesting process to all processes in its request set. The maximum cardinality of a request set can be n-1; therefore a requesting process can send at the most n-1 request messages. Therefore, if a process enters in CS as captain, in the worst case, *n* messages (*n*-1 'request' messages and one token message), needs to be exchanged. However, in case of a follower process, in the worst case, *n*+1 message are required (*n*-1 'request' messages, one 'start' message, and one 'complete' message). However, in the best case no message needs to be exchanged. If a process holding token in *HI* state, wish to attend a session, in that case a new session will be started immediately and the state of the process changes from *HI* to *EC*. No message exchange is required in this case.

Average message size: The Table I describes the various messages used in our algorithm and their sizes.

TABLE I
MESSAGES AND THEIR SIZE

Message Type	Size
'request'	$O(1)$
'token'	$O(N)$
'start'	$O(1)$
'complete'	$O(1)$

Among the messages used in the algorithm, only the token has the size $O(N)$. However, the token is exchanged, only when a new session is initiated. Therefore, in the best case (all processes requesting for the same session), the average message size will be $O(1)$, because one token, N-1 'start', N-1 'complete' and some 'request' messages (depending upon the cardinality of the request sets at each site), will be exchanged. However, in the worst case (all processes requesting for a different session); N token messages will be exchanged, besides the 'request' messages. In this case the average message size will be $O(N)$.

Maximum concurrency: In our algorithm the request of a process requesting for the current session can be fulfilled by the captain process, if no request for some other session is pending in the token.queue. Therefore, if all the processes are requesting for the same session, they can be in their CS concurrently. Hence, the maximum concurrency of our algorithm is n.

Forum switch complexity: The pending requests for a particular session in *token.queue* are grouped together and the requests for one session are treated as a single entry in *token.queue*. Therefore, at any point of time there can be at most **min***(n,m)* entries in *token.queue*. If a process requests for a new session, which has no entry in *token.queue* till now, then a new entry is created and added at the tail of the queue. If we assume only one priority level, after a process has made a request, at most **min**(*n*,*m*) forum switches can take place. However, in a prioritized environment where *k* priority levels exist, a process with higher priority can be placed ahead of a lower priority process, even if the lower priority process entered the queue before the higher priority process. However, due to aging the priority of lower priority process(es) will increase with each forum switch and would succeed in attaining the highest priority level after at most K-1 session switches. Therefore, the forum switch complexity of the algorithm is **max**{**min**(*n*,*m*),(*K*-1)}.

Synchronization delay: The heavy load synchronization delay of the algorithm is $2T$ in the worst case and T in the best case, where T is the maximum message propagation delay.

Under heavy load conditions, there will always be some pending requests in *token.queue*, therefore, as soon as a captain comes out of CS and no follower is in its CS, the *token* is passed to the next captain , and the, heavy load synchronization delay is T. However, if the last process to come out is a follower, it will first send a *complete* message to the captain, which in turn finish the session and passes the *token* to next captain. Therefore, the synchronization delay in this case will be $2T$.

VII. CONCLUSION AND FUTURE WORK

In the present paper, we proposed a token-based algorithm for the group mutual exclusion problem which favors the requests with higher priority levels. This feature of the algorithm makes it suitable for soft real time distributed systems also. The introduction of priority makes a system susceptible to starvation problem. It has been taken care by using the idea of aging while maintaining the strongest fairness requirement that is FCFS, among sessions having same priority levels. The algorithm satisfies the mutual exclusion and concurrent occupancy. The algorithm has reduced forum switch complexity keeping maximum concurrency as n. To the best of our knowledge, the proposed work is the first algorithm on group mutual exclusion that allows a process to declare a priority level along with its request for a session and the algorithm favors the sessions with higher priority levels. Although, Mittal-Mohan algorithm [4] uses the concept of priority to enhance the resource utilization, they do not allow individual processes to assign priority level to their request. However, our algorithm allows individual processes to assign priority level to their request. This characteristic makes our algorithm suitable for use in soft real time environment. An interesting extension of the work could be making it suitable for hard real time systems, which have more stringent deadlines to meet.

APPENDIX

Appendixes, if needed, appear before the acknowledgment.

A. The pseudo code of the algorithm

Code for initialization*:*
For i = 1 to n
{
$state_i$=N; $captain_i$ =NULL
RS_i=ids of all other processes except P_i
For j = 1 to n $SN_i[j]$=0;
}

$state_1$=*HI*; RS_1=Ø
token.type=NULL; *token.queue*=Ø
token.followers=0

P_i request for a forum X with priority Z_i:
$SN_i[i]=SN_i[i]+1$
If ($state_i$=*HI*)
{
token.type=X; $state_i$ =*EC*
RS_i= Ø; Enter CS
}
Else if ($state_i$=*HS*)
{
If (*token.queue*= Ø) && (*token.type*=X)
{
$state_i$=*EC*; Enter CS
}
Else call Add_ request (i,X,Z_i)
}
Else
{
$state_i$=*R*;
Send request (i, SN_i[i], X,Z_i) to all members of RS_i
}

P_i receives request (j,SN,X,Zj):
If $SN>SN_i[j]$ /* otherwise old request
{
$SN_i[j]=SN$
If ($state_i$=*R*) && ($j \notin RS_i$)
{
Add j to RS_i
Send request (i,$SN_i[i]$, *Y*) to j
}
Else If ($state_i$=*EC*)
{
If (*token.type*=X) && (*token.queue*=Ø)
{
token.followers=*token.followers*+1
Send start (i) to P_j
}
Else call add_ request (j,X,Z_j)
}
Else If ($state_i$=*HI*)
{
Add j to RS_i; Send token to P_j
}
Else If ($state_i$=*HS*)
{
If (*token.type*=X) && (*token.queue*=Ø)
{
token.followers=*token.followers*+1
Send start (i) to P_j
}
Else call add_ request (j,X,Z_j)
}
Else Add j to RS_i
}

P_i receives start (j):
$captain_i$=j; $State_i$=*EF*; Enter CS

P_i exits from CS:
If $state_i$=*EF*
{
Send complete (i) to $captain_i$
$captain_i$=NULL; $state_i$=*N*
}
Else
{
If (*token.followers*=0) && (*token.queue*=Ø)
{
$state_i$=*HI*; *token.type*=NULL
}
If (*token.followers*=0) && (*token.queue*≠Ø)
{
If (i's request in *token.queue*) $state_i$ =*N* **else** $state_i$=*R*
Increment priority level of all entries in token.queue by one
Add all processes which are in *token.queue* and
which can work as captain to RS_i
Select new captain P_j
Remove Process j and its followers from the front of the queue (requesting for a session X)
token.type=X
token.followers=number of follower processes
Send *token* (*token.queue*, *token.type*, *token.followers*) to P_j
Send start (j) to all followers
}
Else $state_i$ =*HS*
}

P_i receives comlete(j):
token.followers=*token.followers*-1
If (*token.followers*=0) && ($state_i$=*HS*)
{
If (*token.queue*=Ø) $state_i$=*HI*
Else
{
If (i's request in *token.queue*) $state_i$ =*N* **else** $state_i$=*R*
Increment priority level of all entries in token.queue by one
Add all processes which are in *token.queue* and can work as captain to RS_i
Remove Process P_j and its followers from the front of the *token.queue* (requesting for X)
token.type=X
token.followers=number of follower processes
Send token (*token.queue*, *token.type*, *token.followers*) to P_j
Send start (j) to all followers
}
}

P_i receives token
$state_i$=*EC*; enter CS; RS_i =Ø

Proceedure Add_request(*i*,X,*Z*)

If (entry for session X already in *token.queue*
Add current request also in the list of requests for X
If (X.priority<*Z*) X.*priority*=*Z*
else
{
create a new entry in *token.queue* for session X
add this request at the rear of the token.queue
X.*priority*=Z_i
}
Y=session in entry ahead of entry of X in *token.queue*
While (Y.*priority*<X.*priority*)
{
Swap entry corresponding to session X with

```
    entry corresponding to session Y
    Y=session in entry ahead of entry of X in token.queue
  }
}
```

REFERENCES

[1] Ye-In chang, M.Singhal and M. T. Liu, "A dynamic token based distributed mutual exclusion algorithm", In proc. of 10th annual international phoenix conference on computers and communications, Pages 240 – 246, 1991.

[2] Y.J.Joung, "Asynchronous group mutual exclusion (extended abstract)", In proceedings of the 17the annual ACM symposium on principles of distributed computing (PODC), Pages 51 –60, 1998.

[3] Y.J.Joung, "The Congenial talking philosopher problem in computer networks", distributed computing, Vol. 15, Pages 155-175, 2002.

[4] N.Mittal and P.K.Mohan, "An efficient distributed group mutual exclusion algorithm for non-uniform group access", In proceedings of the international conference on parallel and distributed computing systems, 2005.

[5] P.Kean and M.Moir, "A simple local spin group mutual exclusion algorithm", In proceedings of the 18th annual ACM symposium on principles of distributed computing, pages 23-32, 1999.

[6] Y.Manabe and J.Park, "A quorum based extended group mutual exclusion algorithm without unnecessary blocking", In proceedings of 10th international conference on parallel and distributed systems (ICPADS'04), 2004.

[7] R.Attreya and N.Mittal, "A dynamic group mutual exclusion algorithm using surrogate quorums", In proceedings of the 25th IEEE conference on distributed computing systems (ICDCS'05), 2005.

[8] M.Toyomura, S.Kamei and H.Kakugawa, "A quorum–based distributed algorithm for group mutual exclusion", PDCAT'03 Pages 742-746, 2003.

[9] D.Lin, T.-S.Moh and M.Moh, "Brief announcement: improved asynchronous group mutual exclusion in token passing networks", In proceedings of PODC'05, Pages 275-275, 2005.

[10] G.Ricart and A.K.Agrawala, "An optimal algorithm for mutual exclusion in computer networks", communications of the ACM 24(1), Pages 9-17, 1981.

[11] Q.E.K Mamun, H.Nakazato, "A new token based group mutual exclusion in distributed systems", In the proc. of the Vth international symposium on parallel and distributed computing, 2006.

[12] F.Mueller, "Prioritized token-based mutual exclusion for distributed systems" 9th Symposium on parallel and distributed processing, Pages 791-795, 1998.

[13] A. Housini, M.Trehel, "Distributed mutual exclusion token-permission based by prioritized groups", Proc. of ACS/IEEE international conference, Pages 253-259, 2001.

[14] Y.Chang, "Design of mutual exclusion algorithm for real time distributed systems", Journal of Information science and engineering, Vol.10, Pages 527-548, 1994.

[15] A. Goscinski, "Two algorithms for mutual exclusion fro real time distributed systems", Journal of Parallel and Distributed Computing, Vol. 34(1), Pages 1-13, 1996.

[16] Y.I.Chang, "Comments on two algorithms for mutual exclusion in real-time distributed computer systems", Journal of Parallel and Distributed Computing, Vol. 23, 449-454 (1994).

[17] O.Thiare, M.Gueroui, and M.Naimi, "Distributed group mutual exclusion based on client/servers model", In the proceedings of 7th international conference on parallel and distributed computing, applications and technologies (PDCAT'06), 2006.

[18] M. Singhal, "A heuristically–aided Algorithm for mutual exclusion in distributed systems", IEEE Transactions on Computers, vol. 38, no.5, pp. 651 – 662, 1989.

[19] I. Suzuki, T. Kasmi, "A distributed mutual exclusion algorithm", ACM Transactions on Computer Systems, vol. l3, no. 4, pp. 344 – 349, 1985.

[20] M. Maekawa, "A √n algorithm for mutual exclusion in decentralized systems", ACM Transactions on Computer Systems, vol. 3, no. 2, pp. 145 – 159, 1985.

[21] S. Karnar, and N. Chaki, "Modified Raymond's Algorithm for priority(MRA-P) based mutual exclusion in distributed systems", ICDCIT 2006,LNCS 4317, pp. 325-332, 2006.

[22] Siberschatz, A., Galvin, P. B., Gagne, G. "Operating Systems Concepts", 6th edition, John Wiley & Sons, Inc., 2002.