International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:4, No:4, 2010

A New Extended Group Mutual Exclusion Algorithm with Low Message Complexity in Distributed Systems

S. Dehghan and A.M. Rahmani

***Abstract*—**The group mutual exclusion (GME) problem is an interesting generalization of the mutual exclusion problem. In the group mutual exclusion, multiple processes can enter a critical section simultaneously if they belong to the same group. In the extended group mutual exclusion, each process is a member of multiple groups at the same time. As a result, after the process by selecting a group enter critical section, other processes can select the same group with its belonging group and can enter critical section at the moment, so that it avoids their unnecessary blocking. This paper presents a quorum-based distributed algorithm for the extended group mutual exclusion problem. The message complexity of our algorithm is $O(4|Q|)$ in the best case and $O(5|Q|)$ in the worst case, where $|Q|$ is a quorum size.

***Keywords*—**Group Mutual Exclusion (GME), Extended GME, Distributed systems.

I. INTRODUCTION

DISTRIBUTED system is a set of processes (computers) connected by communication links. Mutual exclusion is a fundamental problem in distributed systems. When some resources (for example, a file, a communication channel, a printer) are shared among processes, two processes are not allowed to enter a critical section (CS) and use it at the same time. Recently, group mutual exclusion [1] has been proposed. There are multiple groups of processes. The processes in the same group can enter CS at the same time. An interesting application of the group mutual exclusion is presented in [2]. Consider large data sets stored in a secondary memory. A set of processes accesses to the data sets through a server. The server can be a CD jukebox. Using a regular mutual exclusive protocol, the server needs to repeatedly load and unload the data sets (e.g., the CDs) from the secondary memory to process the requests.

An efficient GME protocol would allow multiple processes to read the currently loaded data set (a CD) concurrently, while forcing processes requesting a different data set (another CD) to wait. For example, each process wants to read some data on the CDs. If CD *A* is loaded, multiple processes which want to read data on CD *A* can access it at the same time. These processes are in the same group. By contrast, the processes which want to read data on CD *B* cannot do so when *A* is loaded. These processes form a different group. In [3], the following extended definition has been introduced. Some processes might be members of multiple groups at the same time. In the CD jukebox example, the same data might be copied on CD *B* and CD *C*. In this situation, the user can read the data if either CD *B* or CD *C* is currently loaded. A different example of GME is a server that can cache locally one page of the web. Clients of this server interested in the currently cached page can read it simultaneously; clients interested in a different page must wait [4].

For group mutual exclusion, shared memory system algorithms [4-7], token-based algorithms [8-11] and quorum-based algorithms [3, 12-14] have been proposed. The extended group mutual exclusion problems are discussed in [3] and [14] – both of the solutions are quorum-based protocols. Though [3] discusses this extended group mutual exclusion problem, it just notes that when a process belongs to multiple groups, it arbitrarily selects one group. The above algorithm is not sufficient, since when the process *p* enters CS, another process *p'*, which can enter CS at the time might be blocked. Manabe et al. [14] called above situation as an unnecessary blocking.

In [14] a quorum-based extended group mutual exclusion algorithm is proposed that not have unnecessary blocking and the message complexity of this algorithm is $O(6|Q|)$ in the best case and $O(9|Q|)$ in the worst case where $|Q|$ is a quorum size. In the present study, we proposed a new algorithm for extended group mutual exclusion that avoids unnecessary blocking of processes. The message complexity of our algorithm is $O(4|Q|)$ in the best case and $O(5|Q|)$ in the worst case where $|Q|$ is a quorum size. Also the message complexity of our algorithm is less than [14].

The organization of this paper is as follows: In section two the system model and definition are presented. In section three the proposed algorithm introduced. In section four correctness of the algorithm is explained. In the fifth part message complexity of proposed algorithm is presented. In the sixth part, performance of proposed algorithm is compared with

Somaiyeh Dehghan is with department of computer engineering, Islamic Azad University, Ilkhchi Branch, Tabriz, Iran. (phone:+984123326060, e-mail: so.dehghan@iauil.ac.ir)

Amir Masoud Rahmani is with department of computer engineering, Islamic Azad University, Science and Research Branch, Tehran, Iran. (e-mail: rahmani@sr.iau.ac.ir).

former algorithms. And finally in the seventh part the conclusion is given.

II. SYSTEM MODEL AND DEFINITION

B. The System Model

We assume an asynchronous distributed system which consists of a set of n processes $V = \{p_1, p_2, ..., p_n\}$. Each process has a unique identifier selected from a set of integers $\{1,2,..., n\}$. The processes communicate with each other by passing messages through first-in, first-out (FIFO), asynchronous, and reliable (no message loss occurs) channels. We assume that the system is error-free. In our algorithm, we have two classes of processes, *director processes* and *applicant processes***.** A *director process* manages permissions and handles requests from *applicant processes*. An *applicant process* makes a request for resources and asks for permissions from *director processes*. In actual systems, one process can perform both roles simultaneously. Also, to avoid starvation and deadlock, each *director process* has a priority queue that if *director process* doesn't grant permission to *applicant process* to enter CS, then a request is inserted to queue. Priority in this queue is assigned by a timestamp, which is a pair logical clock value on request and process identifier [15-16].

B. Coteries

We describe formal definition of coterie [17] below.

Definition (Coterie [17]):

Let $U = \{u_1, u_2, ..., u_n\}$ be a set. A set C of subsets of U is a *coterie* under U if and only if the following three conditions are satisfied.

1. Non-emptiness: For each $Q \in C$, Q is not empty and $Q \subseteq U$.
2. Intersection property: For any $Q, Q' \in C$, $Q \cap Q'$ is not empty.
3. Minimality: For any $Q, Q' \in C$, Q is not a subset of Q'.

An element of C is called a *quorum***.**

Coterie is widely used for distributed mutual exclusion. Coterie is a set of quorums, and quorum is a subset of processes V such that any two quorums have non-empty intersection. A process wishing to enter its critical section sends a request to each process in a quorum. A process enters its critical section after it obtains permission from every process in a quorum, and releases the permission when it exits its critical section. Because the two quorums have not nonempty intersection, mutual exclusion is guaranteed if each process never grants permission to more than one process at a time.

II. DESCRIPTION OF THE PROPOSED ALGORITHM

This section provides a brief description of our algorithm shown in figures 1 and 2. Let $C = \{Q_1, Q_2, .., Q_n\}$ be a set of quorums, $A = \{p_1, p_2, .., p_m\}$ be a set of *applicant processes* and $G_{all} = \{G_1, G_2, .., G_k\}$ be a set of groups. When process p_i wants to enter CS, it selects its group set G_i from $G_{all} = \{G_1, G_2, .., G_k\}$, and it selects a quorum $Q \in C$ and sends its request to every $q \in Q$.

When *director processes* receive this request, they grant permission to p_i provided that no permission was granted before and CS is empty. When p_i receives permission from each $q \in Q$, selects desired group from its own group set G_i and enter CS. Other *applicant processes* that are in the same group with p_i can enter CS concurrently. As a result, extended GME is guaranteed and their unnecessary blocking is avoided.

But this definition: "allowing another process to enter CS at the same time", leads to starvation [14]. Thus, to avoid starvation, the fallowing method is used. When CS is empty, the first process which enters CS through selecting the desired group is as a *master*. Then, other *applicant processes* belonging to the same group can enter CS as *slave* simultaneously while *master* is in CS. As soon as *master* exit from CS, *director processes* do not grant permission to other processes. Then, after all *slaves* exit from CS, other processes enter and therefore starvation is avoided. In this case *director processes* select one request based on timestamp of suspended requests in queue and send permission to enter CS as *master*. After the *master* enters, *director processes* send permission to other suspended requests in queue that are in the same group with *master* to enter CS as *slave*.

A. Solutions of Proposed Algorithm for Improvement of Former Algorithm

This section presents solutions of proposed algorithm to decrease communication messages and the used memory by queues of *director processes*. In [14] each $q \in Q$ receives a request, first inserts it to queue. However only the suspended request can enter to queue and decrease used memory by queues. Also, when the *pivot* process wants to exit CS, it must waits for other process to exit from CS which cause improper wait of *pivot* process.

To eliminate improper wait of *pivot* process, the *director processes* must distinguish the exit of *master* and *slave* processes from CS. In fact guarantee of mutual exclusion is the duty of *director processes*.

For this purpose, four variables are used. *Master* is a string variable that saves *master* process name, *Master_in* is a boolean variable that shows the presence or absence of *master* in the CS, *user* is a set variable which shows the set of processes which serve as *slaves* in the CS and *scount* is a integer variable which shows the number of processes that allowed them be as slaves but have not enter CS yet.

Program *Applicant Process* (p_i: process);	***A-3.***

```
var
  Astatus: status   // { in, out, wait_grant, wait_perm }
  perm: set;              // set of quorum that send permission to p_i
  Q: quorum: set of processes;    // C ={Q_1, …, Q_n} set of quorums
  G_all: set of groups;
  G_i : set;              // group set of applicant process p_i
  ts: Time;               // timestamp of request
  Mg: string;             // master group (or current group in CS )

A-1.
  When p_i (group set is G_i) wants to enter CS:  {
       send "Request_Master(G_i, ts)" to all q∈Q ;
       Astatus = wait_grant;
       perm = ϕ ;
  }

A-2.
   At arrival of "Enter_Slave(Mg)" from q: {
       if  (Astatus = = wait_grant) ∨ (Astatus = = wait_perm)
       {
            send "Slave(Mg)" to all q∈Q ;
            Astatus = in;
               // in the CS
            Astatus = out;
            send "Release" to all q∈Q ;
       }
  }

  At arrival of  "Enter_Master" from q:  {
     if   (Astatus = = wait_ grant)  ∨
          (Astatus = = wait_perm)
     {
          perm = perm ∪ {q};
          if (perm = = Q) {
               select desired group g ∈ G_i ;
               send "Master(g)" to all q∈Q ;
               Astatus = in;
                  // in the CS
               Astatus = out;
               send "Release"  to all q ∈Q;
               perm = ϕ ;
          }
     }
}

A-4.
   At arrival of  "No_Enter" from q:  {
        Astatus = wait_perm
   }
```

Fig. 1. The pseudo code of the *applicant process* algorithm

Then *Director processes* can distinguish exit of *master* and *slave* processes from CS and therefore remove one request from queue with highest priority based on its timestamp and grant permission to it in order enter CS as a *master*.

B. The Applicant Processes Algorithm

The summary of the pseudo code for *applicant processes* shown in figure 1 is fallows:

A-1. When process p_i whose group set is G_i wants to enter CS, it selects a quorum $Q \in C$ and sends a request to every process in Q.

A-2. If any of the $q \in Q$ granted permission to enter as a slave then it can enters CS as *slave*.

A-3. If all $q \in Q$ granted permission as master, it would select the desired group g from whose group set G_i and enter CS as *master*.

A-4. If no permission is granted, it should wait.

Also when p_i exiting CS, sends release message to every $q \in Q$.

C. The Director Processes Algorithm

The summary of the pseudo code for *director processes* shown in figure 2 is fallows:

B-1. When q receives a request from p_i **:**

I. If permission is granted to other process to enter CS as a *master* before, then it inserts its request to queue and doesn't grant permission to p_i.

II. If no permission is sent to other process that enter CS as a *master* before, and if CS is empty, then it sends permission to p_i which enters CS as a *master*.

III. Otherwise if *master* is in the CS, if p_i is in the same group with *master*, it allows p_i to enter as a *slave* and increments *scount*. But if *master* isn't in the CS, it inserts its request to queue and doesn't grant permission to p_i.

B-2. When q receives message that p_i has entered CS as a *slave*, then it adds p_i to *user* set and decrements *scount*.

B-3. When q receives message that p_i has entered CS as a *master*, then $\forall$ all process p_j in queue if $Mg \in G_j$, it sends permission to enter CS as a *slave* and remove those from queue.

B-4. When q receives release message from p_i, if there is no other process in the CS and don't send permission to other process that enter CS as a *slave* before (that is *scount*= 0), then it removes one request from queue with highest priority based on its timestamp and grant permission to it to enter CS as a *master*.

```
Program Director Process (q: process);
var
  Dstatus: status;     // { Idle, wait_master, wait_slave }
  Master : string;                    // contain Master process name
  Master_in = false: boolean;    // Master is in the CS or not
  Mg: string;         // master group ( current group in CS )
  CS: string;         // CS is empty or full
  Queue = null    // priority queue of suspended requests;
  user = ϕ:set;     // number of processes that are in the CS
  scount =0;// number of processes that allowed them be as slaves

B-1.
  At arrival of "Request_Master(G_i, ts)" from p_i :
  {
     if ( Dstatus = = wait_master)
       {
            insert  (p_i , G_i, ts) to Queue;
            Send "No_Enter" to p_i ;
       }
      if (Dstatus = = Idle)

          if  (CS = = 'empty' ) {
              send "Enter_Master"  to p_i  ;
              Dstatus = wait_master;
           }
      if (Dstatus = = Idle) ∨ (Dstatus = = wait_slave) {

            if  (Mg ∈ G_i) ∧ (Master_in = true) {
                send "Enter_Slave(Mg)"  to p_i ;
                scount ++;
               Dstatus = wait_slave;
             }
            else {
                insert (p_i, G_i, ts) to Queue;
                send "No_Enter" to p_i ;
             }
       }
  }


B-2.
  At arrival of  "Slave(Mg)" from p_i : {
      if (Dstatus = = wait_slave)
      {
          user = user ∪{ p_i };
          scount -- ;
          if (scount = = 0)   Dstatus = Idle;
       }
  }

B-3.
  At arrival of  "Master(g)" from p_i : {
     if  (Dstatus = = wait_master ) {
        Master = p_i ;
        Master_in = true;
        Mg = g;     CS = 'full';
         ∀ all  processes  p_j in Queue {
            if  (Mg ∈ G_j) {
                Send "Enter_slave(Mg)"  & remove it
                                                from Queue;
                 Dstatus = wait_slave;
                scount ++;
             }
            else
            Dstatus = Idle;
        }

     }
  }

B-4.
  At arrival of  "Release" from p_i :
  {
     if  (Master = = p_i ) {
        Master_in = false;
        if ( user = = ϕ) ∧ (scount = = 0)
          {
            CS = 'empty';
            if (Queue is not empty) {
                Remove item (p_j, G_j, ts) from Queue
                    with highest Priority
                    & send "Enter_Master" to p_j ;
               Dstatus = wait_master;
             }
            else  Dstatus = Idle;
        }
     }
     else {
          user = user -{ p_i };
          if   (user = = ϕ) ∧ (Master_in = = false) ∧
              (scount = = 0)
           {
              CS =  'empty';
              if (Queue is not empty) {
                  Remove item  (p_j, G_j, ts) from Queue
                      with highest Priority
                      & send  "Enter_Master" to p_j ;
                 Dstatus = wait_master;
               }
              else  Dstatus = Idle;
           }
      }
  }
```

Fig. 2. The pseudo code of the *director process* algorithm

IV. PROOF OF CORRECTNESS

A. Safety

The mutual exclusion requirement in GME problem says that, no two processes requesting for a different group, must be in their CS simultaneously. Suppose p_1 and p_2 are applicant processes that request different groups. Let q_1 and q_1 be quorums that p_1 and p_2 select, respectively. Because any two quorums have non-empty intersection, we have $Q \cap Q' \neq 0$ then if p_j be a *director process* in the intersection, since p_j never sends permission for more than one group at a time, therefore p_1 and p_2 can not be granted by p_j simultaneously.

TABLE I PERFORMANCE COMPARISON

	Message Complexity			Extended GME	Unnecessary blocking
Algorithm	***Best case***	***Worst case***			
Joung [3]	$\sqrt{\frac{2n}{m(m-1)}}$		*n: number of processes* *m: number of resources*	Yes	Yes
Toyomura et al. [13]	$O(3\lvert Q\rvert)$	$O(n\lvert Q\rvert)$	*Q: size of Quorum* *n: number of processes*	No	Yes
Manabe et al. [14]	$O(6\lvert Q\rvert)$	$O(9\lvert Q\rvert)$	*Q: size of Quorum*	Yes	No
New Proposed Algorithm	$O(4\lvert Q\rvert)$	$O(5\lvert Q\rvert)$	*Q: size of Quorum*	Yes	No

B. Freedom from Starvation

Assume that starvation occurs. Let p_1 be the process in group g starves. We assume that $ts(p_1)$ is the smallest among starved processes. It is clear that the request of p_1 is eventually granted if no process is its critical section and no other process is making a request.

Therefore we consider a case that some processes in group $g'(\neq g)$ repeat entering and exiting their CS.

According to proposed algorithm, *director processes* don't grant permission to other processes to enter CS as a *slave* after exiting *master* process from CS. Then, all *slave* processes exit from CS finally. Also the timestamp value for each request is increasing.

As a result, when all processes exit from CS, the *director processes* remove one request from queue with highest priority base on its timestamp and grant permission to it to enter CS as a *master*. Thus because $ts(p_1)$ is the smallest among other request, is eventually granted permission and enters its critical section.

V. MESSAGE COMPLEXITY

Message complexity of the proposed algorithm is $O(4\lvert Q\rvert)$ in the best case and $O(5\lvert Q\rvert)$ in the worst case, where $\lvert Q\rvert$ is the size of the smallest quorum in a coterie.

In the best case, only one request exists to enter CS. Then four following types of messages are exchanged between an applicant process and each *director process* in a quorum:

a. *Applicant process* p_i send "Request Master(G_i, *ts*)" to every $q \in Q$.

b. p_i received "Enter_Master" from every $q \in Q$.

c. p_i selects desired group g from whose group set G_i and sends "Master(*g*)" to every $q \in Q$, and enter CS as *master*.

d. When p_i exiting CS, sends "Release" to every $q \in Q$.

Thus, message complexity is $O(4\lvert Q\rvert)$ in the best case.

The worst case happens in two conditions. The first condition is the time when *applicant process* is not the same group as CS inside processes, the next condition is the time when *applicant pro*cess is in the same group as processes CS inside but *master* process exits from CS. In these two conditions, one additional message in comparison with best case is communicated between *applicant processes* and quorum members. Then five following types of messages are exchanged between an *applicant process* and each *director process* in a quorum:

a. *Applicant process* p_i send "Request Master(G_i, *ts*)"to every $q \in Q$.

b. p_i received "No_Enter" from some of the $q \in Q$, then wait for permission.

c. After a time p_i received one of this messages:

- "Enter_Master" from every $q \in Q$, then selects desired group g from whose group set G_i and sends "Master(*g*)" to every $q \in Q$, and enter CS as *master*.
- "Enter_Slave(*Mg*)" from some of the $q \in Q$, then selects *Mg* as desired group and sends "Slave(*Mg*)" to every $q \in Q$, and enter CS as *slave*.

d. When p_i exiting CS, sends "Release" to every $q \in Q$.

Thus, the message complexity of proposed algorithm is $O(5\lvert Q\rvert)$ in the worst case.

VI. PERFORMANCE COMPARISION

In this section, the complexity and extended GME ability of proposed algorithm compared with three former algorithms: Joung [3], Toyomura et al. [13] and Manabe et al. [14]. Table I shows this comparison. The result shows that proposed algorithm has lower complexity in both best case and worst case with respect to other algorithms and has extended GME ability which after the process by selecting a group, enters critical section, other processes can select same group with its belonging group and can enter critical section at the moment, so avoid their unnecessary blocking.

VII. CONCLUSION

In the present paper, we proposed a new quorum based algorithm for the extended group mutual exclusion problem in distributed systems. The message complexity of our algorithm is $O(4|Q|)$ in the best case and $O(5|Q|)$ in the worst case. Also the message complexity of our algorithm is less than former algorithms.

REFERENCES

[1] Y.-J. Joung, Asynchronous group mutual exclusion, Distributed Computing, 13,4, pp. 189-206 , 2000.
[2] Y.-J. Joung. Asynchronous group mutual exclusion (extended abstract). In Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing (PDOC), Puerto Vallarta, Mexico, ACM Press, pages 51–60, June 28-July 2, 1998.
[3] Y.-J. Joung, Quorum-based algorithm for group mutual exclusion, IEEE Trans. on Parallel and Distributed Systems, Vol.14, No.5, pages 463-476, May 2003.
[4] V. Hadzlilacos, A note on group mutual exclusion, Proc. Of 20th PODC, pp. 100-106, 2001.
[5] P. Jayanti, S. Petrovic, and K. Tan, Fair Group Mutual Exclusion, Proc. 22nd PODC, pp. 275-284, 2003.
[6] P. Keane, M. Moir, A simple local-spin group mutual exclusion algorithm, IEEE Trans. Parallel and Distributed Systems, 12, 7, pages 673-685, 2001.
[7] K. Vidyasankar, A highly concurrent group mutual l-exclusion algorithm, Proc. of 21th PODC, 2002.
[8] S. Cantareli, A.K. Datta, F. Petit, V. Villain, Token Based group mutual exclusion for asynchronous rings, Proc. of 21st ICDCS, pages 691-694, 2001.
[9] S. Cantareli, A.K. Datta, F. Perit, V. Villain, Group Mutual Exclusion in Token Rings, Proc.of 8thColloquium Structural Information and Communication Complexity, June 2001.
[10] K.-P. Wu, Y.-J. Joung, Asynchronous Group Mutual Exclusion in Ring Networks, IEEE Proc. Computers and Digital Techniques, Vol.147, No.1, pp.1-8, 2000.
[11] Q.E.K. Mamun, H. Nakazato,, A New Token Based Protocol for Group Mutual Exclusion in Distributed System, Proceedings of The Fifth International Symposium on Parallel and Distributed Computing (ISPDC'06), 2006.
[12] R. Atreya, N. Mittel, A Distributed Group Mutual Exclusion Algorithm using Surrogate-Quorums, Technical Report, The University of Texas at Dallas, 2003.
[13] M. Toyomura, S. Kamei, and H. Kakugawa, A Quorum based Distributed Algorithm for Group Mutual Exclusion, Proc. 4th Int. Conf. on Parallel and Distributed Computing, Applications and Technologies, pp.74-74, Aug. 2003.
[14] Yoshifumi Manabe, JaeHyrk Park, A quorum-based extended group mutual exclusion algorithm without unnecessary blocking, Proceedings of the Tenth International Conference on Parallel and Distributed Systems (ICPADS'04), pp. 341 - 348, July 2004.
[15] L. Lamport. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 21(7):558-565, July 1978.
[16] M. Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. ACM transactions on Computer Systems, 3(2):145-159, March 1985.
[17] H. Garcia-Molina, D. Barbara, How to assign votes in a distributed system, Journal of the ACM, 32, 4, pp. 841-860, 1985.