Journal of Computer Science 3 (10): 829-835, 2007
ISSN 1549-3636
© 2007 Science Publications

A Token-Based Fair Algorithm for Group Mutual Exclusion in Distributed Systems

[1]Abhishek Swaroop and [2]Awadhesh Kumar Singh
[1]Department of Computer Science and Engineering,
GPM College of Engineering, Delhi 110036, India
[2]Department of Computer Engineering,
National Institute of Technology, Kurukshetra 136119, India

Abstract: The group mutual exclusion (GME) problem is a generalization of the mutual exclusion problem. In group mutual exclusion, a process requests a session before entering its critical section (CS). Processes requesting the same session are allowed to be in their CS simultaneously, however, processes requesting different sessions must execute their CS in mutually exclusive way. The paper presents a token-based distributed algorithm for the GME problem in asynchronous message passing systems. The algorithm uses the concept of dynamic request sets. The algorithm does not use any message to be exchanged in the best case and uses n+1 messages in the worst case, where n is the number of processes in the system. The maximum concurrency of the algorithm is n and synchronization delay under heavy load (worst case) is 2T, where T is the maximum message propagation delay. The algorithm uses first come first serve approach in selecting the next session type and satisfies the concurrent occupancy property. The static performance analysis and correctness proof is also included in the present exposition.

Key words: Concurrency, critical section, request set, session

INTRODUCTION

A distributed system is a collection of independent computers, which are capable of collaborating on a task. Although, mutual exclusion is a classical problem of distributed systems, group mutual exclusion (GME) is a comparatively new problem. Joung proposed GME problem as an interesting generalization of the mutual exclusion problem in[1] and modeled it as congenial talking philosophers (CTP) problem[2]. In CTP problem, there are n philosophers and m forums; however, there is only one meeting room. A philosopher may be in any one of the following three states - thinking, waiting or talking. A philosopher interested in a forum may enter the meeting room, if the meeting room is empty or some philosopher interested in the same forum is already in the meeting room, otherwise he has to wait.

The well-known readers-writer problem is a special case of GME problem, where we can use a common read session for all processes and a unique write session for each individual process. Another interesting application of GME is, when several users share large data objects stored in secondary storage (such as CD's) and only one data object can be loaded in the buffer at a time. The users interested in the data object, currently loaded in the buffer, are allowed to access it concurrently; however, users trying to access different object(s) must wait.

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 philosopher is currently attending or requesting a different session, then P can attend X without waiting for any other process to leave the session.

Kean and Moir[3] were first to introduce the term concurrent occupancy. Hadzilacos[4] redefined the term concurrent entering, though it was introduced by Joung in his seminal work[1], for shared memory model, according to which If a philosopher P requests a forum and no other philosopher is requesting a different forum, then P enters the meeting room, within a

Corresponding Author: Abhishek Swaroop, 360/7a, Street no. 6, Bhola nath nagar, Shahdra, Delhi-110032, India, Tel: 91-11-2200003

bounded number of its own steps. However, the bounded number of its own steps can not be guaranteed by any process in the message passing systems. Therefore, the idea of concurrent occupancy, defined by Kean and Moir, has been followed in most message passing GME algorithms.

The GME problem was introduced and solved by Joung[1] for shared memory model. Later on Joung[2] extended the solution for message passing systems using the idea of Ricart and Agrawala[5]. Numbers of non token-based solutions, for GME problem, have been proposed in the literature[3,6-9]. Token-based algorithms for ring networks are given in[10,11]. Token-based algorithms for fully connected networks are presented in[12-14]. Mittal-Mohan algorithm[12] is particularly suitable for those applications, where some small numbers of groups are more in demand compared to other groups. The scheduling policy of Mittal-Mohan algorithm is not fair; nevertheless, the algorithm is starvation free. The message complexity of the algorithms is 2*(n-1), where n is the number of processes. In Mamun-Nazakato algorithm[13], a session is opened for a predefined time and process are made aware about it through broadcast. Hence, the processes, interested in the currently opened session, may join it without incurring any message overhead. Further the algorithm needs that the processes maintain synchronized logical clocks. No message is propagated in the best case and n+2 messages are used in the worst case. The synchronization delay of the algorithm is 2T in the worst case.

The present paper illustrates an FCFS token-based algorithm for solving the GME problem. Our algorithm is based upon Chang-Singhal-Liu algorithm[15], which uses the concept of dynamic request sets to solve the mutual exclusion problem. In our algorithm, the process, entering first in a session, is declared captain and starts the session. A start message is sent to all other processes willing to attend the same session. The processes, allowed by captain to join the currently open session, are called followers.

The data structures used: Each process may be in any one of the six states described in table 1.

The state of a process P_i is stored in its local variable $state_i$. Each process P_i maintains a request set RS_i, which contains the process ids of all the processes to which P_i sends a request, in case P_i wishes to attend some session. Besides that, P_i maintains an array of sequence numbers SN_i. $SN_i[j] = k$ denotes that P_i knows about k requests made by P_j. In addition, P_i has another local variable $captain_i$, which stores id of the 'captain' of current session, if P_i is in its CS as follower, otherwise $captain_i$ is set to NULL.

Table 1: The states

State	Semantics
N	Not Requesting
R	Requesting
EC	Executing in CS as captain
EF	Executing in CS as follower
HS	Holding token because some follower processes are still in CS
HI	Holding token because no request is pending

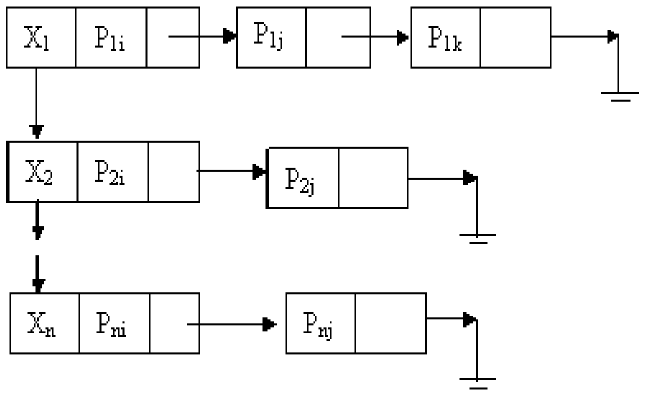

Fig. 1: The structure of token.queue

The token, in our algorithm, is a message that contains an FCFS queue, namely token.queue in order to store all pending requests. The token contains two more variables, namely token.type to store the type of current session and token.followers to store the number of follower processes to which the start message has been sent and which are still in CS. The requests for the same session are grouped together and treated as one entry in the queue. The structure of token.queue is shown below in Fig. 1.

Types of messages: Each process P_i exchanges the following types of messages.

- Request (i,SN_i,X)-It contains the id of P_i, sequence number of the request and the type of session requested. When process P_i wishes to attend a forum X and P_i is not holding the token then it sends this message to all processes in its request set.
- Start (i)-This message is sent to the followers in order to allow them to join the session, which is currently open.

- Complete (i)-This message is sent to the captain when process P_i exits from CS as follower.
- Token (token.type, token.queue, token.followers) - It is the only token existing in the system and the only process holding it can enter in its CS as captain. Whenever a session finishes and next session is selected, it is passed to the next captain.

Description of the algorithm: The code for initialization is given in Fig. 2 and the rest of the pseudo code is given in Fig 3; however, its working is described below.

```
For i = 1 to n
 {
  State_I = N; Captain_i = NULL
  RS_i = {ids of all processes except that of 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
```

Fig. 2: Pseudo code for initialization

. Initially, all processes are in state N, having their captain NULL, all entries of SN are zero and the request set of each process contains ids of all other processes except its own id. Only exception is P_1. We assume that P_1 holds the token initially, therefore, the variable state$_1$ is modified to HI and RS_1 is emptied.

When process P_i wishes to attend a session X, it increments the sequence number $SN_i[i]$ by 1. If P_i is in state HI then P_i enters the CS and sets state$_i$ equal to EC. If state$_i$ is HS, token.queue is empty and token.type = X then P_i enters in its CS and changes its state to EC. However, if state$_i$ is HS and token.queue is not empty or token.type$\neq$X, the request is added in the token.queue. Otherwise state of P_i becomes R and it sends request messages to all processes in its request set and waits for the token or start message.

A process P_i, upon receiving a request message from P_j, checks whether the request is new or old. P_i 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_i is also requesting and P_j is not in RS_i, P_i sends its request to P_j and adds P_j to RS_i. If state$_i$ is HI, it adds id of process P_j to RS_i and immediately sends the token to P_j. However, if P_i is holding token in state HS, it sends start message to P_i, if X is the currently open session and the token.queue is empty. Otherwise the request is added in the token.queue. If state of P_i is N or EF and P_j is not in RS_i, then P_j is added to RS_i.

When a follower process comes out of its CS, it sends complete message to its captain, 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 there is no request pending, 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 N and starts new session. In order to start new session, it removes front element of token.queue and appoints it the next captain. Subsequently, it removes followers of the newly appointed captain from the token.queue, sends token to the newly appointed captain and sends start message to all followers of the newly appointed captain.

On receiving every complete message, the captain decrements variable token.followers by one. If the state of the captain is HS and token.followers attains value 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 N and starts new session.

The captain process, on receiving token, changes its state to EC and enters in its CS. On receiving a start message, a process becomes follower. It changes its state to EF, stores id of its captain and enters in its CS.

Correctness proof: Our algorithm satisfies the properties, which are necessary for a correct solution of group mutual exclusion problem. We consider them one by one

Mutual exclusion: 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 captain and all 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.

Freedom from starvation: An FCFS queue is maintained by the token to store the pending requests. Whenever a session finishes, the process holding the token, passes the token to the process stored at the front of the queue. A start message is sent to all other process n the token.queue, requesting the same session.

```
Pi request for a forum X:
SNi[i]=SNi[ i]+1
If (statei=HI)
{
  token.type=X;  Statei =EC; RSi= Ø; Enter CS
}
Else if (statei=HS)
        {
          If (token.queue= Ø) && (token.type=X)
            {
             Statei=EC; Enter CS
          }
          Else Add request (i,SNi [i],X) to token.queue
        }
Else
   {
     Statei=R;
     Send request (i, SNi [i], X) to all members of RSi
   }

Pi receives request (j,SN,X):
If SN>SNi[j]        /* otherwise old request
{
  SNi[j] =SN
  If (statei=R) && ( j ∉ RSi )  /* i  is requesting for Y
   {
     Add Pj to RS;  Send request (i,SNi[i], Y) to Pj
   }
  Else If (statei=EC)
      {
           If (token.type=X) && (token.queue=Ø)
            {
             token.followers=token.followers+1
         Send start (i) to Pj
            }
           Else add request (j,SN,X) to token.queue
      }
Else If (statei=HI)
       {
        Add j to RSi; Send token to Pj
       }
Else If (statei=HS)
      {
        If (token.type=X) && (token.queue=Ø)
         {
          token.followers=token.followers+1
          Send start (i) to Pj
         }
        Else add request (j,SN,X) to token.queue
      }
     Else Add j to RSi
 }
```

```
Pi receives start (j):
Captaini=j; Statei=EF; Enter CS

Pi exits from CS:
 If statei=EF
  {
    Send complete (i) to captaini
    captaini=NULL;     Statei=N
  }
Else
  {
   If (token.followers=0) && (token.queue=Ø)
     {
      Statei=HI; token.type=NULL
     }
    If (token.followers=0) && (token.queue≠Ø)
     {
          statei =N
      Add all processes which are in token.queue and
      which can work as captain ,to RSi
Remove Process Pj and its followers from the queue
    token.type=X; token.followers=number of followers
Send token (token.queue, token.type, token.followers)
to Pj
     Send start (j) to all followers
      }
    If (token.followers≠0)  statei =HS
}

Pi receives complete(j):
token.followers=token.followers-1
If (token.followers=0) && (state=HS)
   {
         If (token.queue=Ø) statei=HI
         Else
         {
          if (i's request in token.queue) Statei=R    Else
Statei=N
 Add all processes which are in token.queue    and can
work as captain to RSi
Remove Pj and its followers from the. queue
token.type=X; toen.followers=number of followers
Send token (token.queue, token.type, token.followers)
to Pj
Send start (j) to all followers}    }

Pi receives token:
 Statei=EC; enter CS; RSi =Ø
```

Fig 3: Pseudo code of the algorithm

However, if a request of the current session type arrives at the captain, the captain checks whether the token.queue has any pending requests. 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 removes the possibility that the processes of a particular group keep on requesting for the current session and not allowing other processes to enter in their critical sections. Therefore, the sessions in the algorithm are served in a starvation free manner.

Concurrent occupancy: In the proposed algorithm, when a process starts execution in CS as a captain, it allows CS entry to all the processes, requesting for the same session, whose requests are stored in the token.queue. When the captain is in state EC or state HS and a request for the current session arrives, it checks whether the token.queue is empty. If it is so, 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.

Performance analysis: We will analyze the performance of our algorithms using following performance parameters: message complexity per CS request, average message size, forum switch complexity, maximum concurrency and synchronization delay. Forum switch complexity (defined below) and maximum concurrency are applicable for GME algorithms but not for mutual exclusion algorithms.

Forum switch complexity: The forum switch complexity[1] is measured by the maximum number of rounds of passages a process may wait, before it can access the requested resource. A passage by P_i through a session F is an interval $[t_1, t_2]$, where t_1 is the time when process P_i enters the session and t_2 is the time when P_i leaves the session. Further a set of passages S, where $t_s = \min \{t|[t, t']\in S\}$ and $t_f = \max \{t'|[t, t']\in S\}$ is a round of passage through session F, if following conditions are satisfied

- Only those passages which are in S, are initiated between t_s and t_f
- The last passage before t_s and the first passage after t_f are for a session other than F

Forum switch complexity is particularly significant in applications, where changing a session is time consuming, such as applications which require unloading and loading of disk during a session switch.

The following Table 2 describes the size of various messages used in our algorithm.

Theorem 1: The number of messages exchanged per CS entry in our algorithm is n+1 in the worst case and zero in the best case.

Proof: The messages exchanged, during the execution of the algorithm are, request, token, start and complete. The token message is used only once per session, when the current captain transfers the token to the next captain. The start message is sent to the follower and the follower process sends complete message to its captain. Therefore, besides the request messages only one start and one complete message would be required per CS entry by a follower process. Moreover, no start or complete message is needed in case of captain. The request message is 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 has to send at most n-1 request messages. Hence, in the worst case, the number of messages exchanged is n+1 in case of follower (n-1 request messages, one start message and one complete message) and n in case of captain (n-1 request messages and one token message).

Table 2: Messages and their size

Message type	Size
Request	O (1)
Token	O (n)
Start	O (1)
Complete	O (1)

However, in the best case, no messages are exchanged. If a process is in HI state and wishes 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.

Theorem 2: The average message size in our algorithm is O (n) in the worst case and O (1) in the best case.

Proof: All the messages used in the algorithm (request, start, and complete), except the token, have O (1) size. The size of the token is O (n). The token is exchanged only, when a session change occurs. The worst case will occur, when there are n-1 pending requests and each request is for a different session (if we assume that number of processes< = number of sessions). In that case, the token of size O (n) will be exchanged with every CS execution and the average message size will be O (n).

The best case will occur when there are n-1 pending requests; however, all of these requests belong to the same session. In that case, besides the request messages only one token will be transferred and n-1 start and n-1 complete messages will be exchanged. Therefore, the average message size in the best case will be O (1).

Theorem 3: The maximum concurrency of our algorithm is n.

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

Theorem 4: The forum switch complexity of the algorithm is min (n , m), where n is the number of processes and m is the number of sessions.

Proof: 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 the most min (n, m) entries in token.queue. If a process requests 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. Hence, after a process has made a request, at most min (n, m) forum switches can take place, therefore, the forum switch complexity of the algorithm is min(n,m).

Theorem 5: In the worst case, the synchronization delay of the algorithm, under heavy load, is 2T.

Proof: 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 terminates the session and passes the token to next captain. Therefore, the synchronization delay in this case will be 2T.

CONCLUSION

In the present paper, we proposes a token-based algorithm for the group mutual exclusion problem. Our algorithm uses the concept of dynamic request sets. It satisfies safety, concurrent occupancy and the strongest fairness requirement. The maximum concurrency of the algorithm is n and the forum switch complexity is min (n, m). The dynamic performance analysis of the proposed algorithm and the quantitative comparison, with other token-based GME algorithms, is being postponed for a future work. The entry policy of a GME algorithm is very critical, in order to increase the resource utilization. We plan to investigate alternate entry policies and compare them with the entry policy adopted in the present algorithm.

REFERENCES

1. Joung, Y.J., 1998. Asynchronous group mutual exclusion (extended abstract). In Proceedings of the 17th annual ACM Symposium on Principles of Distributed Computing (PODC) : 51-60.
2. Joung, Y.J., 2002. The Congenial talking philosopher problem in computer networks. Distributed Computing, 15: 155-175.
3. Kean, P. and M. Moir, 1999. A simple local spin group mutual exclusion algorithm. 18th annual ACM Symposium on Principles of Distributed Computing : 23-32.
4. Hadzilacos, V., 2001. A note on group mutual exclusion. 20th ACM Symposium on Principles of Distributed Computing: 100-106.
5. Ricart, G. and A.K. Agrawala, 1981. An optimal algorithm for mutual exclusion in computer networks, Communications of the ACM, 24 (1): 9-17.
6. Wu, K.P. and Y.J. Joung, 1999. Asynchronous group mutual exclusion in ring networks. 13th International Parallel Processing Symposium (IPPS 99): 539-543.
7. Manabe, Y. and J. Park, 2004. A quorum based extended group mutual exclusion algorithm without unnecessary blocking. 10th International Conference on Parallel and Distributed Systems (ICPADS 04).
8. Attreya, R. and N. Mittal, 2005. A dynamic group mutual exclusion algorithm using surrogate quorums. 25th IEEE conference on distributed computing systems (ICDCS 05).

9. Toyomura, M., S. Kamei and H. Kakugawa, 2003. A quorum-based distributed algorithm for group mutual exclusion. PDCAT 03: 742-746.
10. Cantarell, S., A. K.Dutta, F. Pilit and V. Villain, 2001. Token based group mutual exclusion for asynchronous rings. IEEE International Conference on Distributed Computing Systems (ICDCS): 691-694.
11. Lin, D., T.S. Moh and M. Moh, 2005. Brief announcement: improved asynchronous group mutual exclusion in token passing networks. Annual ACM Symposium on Principles of Distributed Computing (PODC 05): 275-275.
12. Mittal, N. and P.K. Mohan, 2005. An efficient distributed group mutual exclusion algorithm for non-uniform group access. International Conference on Parallel and Distributed Computing Systems.
13. Mamun, Q.E.K. and H. Nakazato, 2006. A new token based group mutual exclusion in distributed systems. 5th International Symposium on Parallel and Distributed Computing.
14. Thiare, O., M. Gueroui and M. Naimi, 2006. Distributed group mutual exclusion based on client/servers model. 7th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT 06).
15. Ye-In chang, M. Singhal and M.T. Liu, 1991. A dynamic token based distributed mutual exclusion algorithm. 10th Annual International Phoenix Conference on Computers and Communications: 240-246.