Group Mutual Exclusion Based on Priorities

Karina M. Cenci
Laboratorio de Investigación en Sistemas Distribuidos
Universidad Nacional del Sur
Bahía Blanca, Argentina
kmc@cs.uns.edu.ar
and
Jorge R. Ardenghi
Laboratorio de Investigación en Sistemas Distribuidos
Universidad Nacional del Sur
Bahía Blanca, Argentina
jra@cs.uns.edu.ar

***Abstract*—We propose a distributed solution for the group mutual exclusion problem based on priorities, in a network with no share memory whose members only communicate by messages. The proposed algorithm is composed by two players: groups and processes, groups are passive players while processes are active players. For the coordination access to the resource, each group has assigned a quorum. The groups have associated a base priority in each stage, meanwhile the processes have the same level priority. An important feature is that processes have associated a time to participate in the group in each stage. The message complexity obtain, in the best case, where the group does not yield the permission, is $3l + 3(q - 1)$ messages, where l denotes the processes linked and q denotes the quorum size. The maximum concurrency of the algorithm is n, which implies that all processes have linked to the same group.**

***Keywords*-Mutual Exclusion - Group Mutual Exclusion - Concurrency - Distributed Systems**

I. Introduction

In distributed systems there are processes that compete in using resources and others that cooperate and share resources for solving a task. The main problem to solve is to recognize it as a mutual exclusion one. This problem arises in multiprogramming environments because processes require exclusive access for using resources, e.g. printers, database. Different solutions have been proposed to solve this problem, e.g. [1], [6]. When some processes cooperate and others compete, a difference from the original problem appears, known as group mutual exclusion. The concept of group mutual exclusion (GME) can be applied to a variety of areas, e.g., concurrent data structures, distributed data bases, communication, video conferences, wireless systems. In GME two properties are important: exclusion among competing processes and concurrence among cooperative processes. There are different approaches for this problem using different paradigms and implementations.

GME problem is first proposed by Joung [3], in which an asynchronous algorithm for shared memory parallel computer system is proposed. Wu-Joung [13], proposed a solution to the group mutual exclusion on ring network. Several quorum-based algorithms [4] [12] [7] have been proposed for asynchronous message passing. The Manabe-Park [7] algorithm prevents the unnecessary blocking, defined as the case that two processes are prevented from entering a critical section simultaneously even if they are capable of doing so. Kakugawa et al [5] proposed a privileged token approach, with two classes of tokens -main and subtoken-, and used coteries for communication structure to reduce message complexity. Singh-Su [9] proposed a solution to the region synchronization problem (such as mutual exclusion, group mutual exclusion, readers/writers) using message and satisfying the property of absence of unnecessary blocking. There are also solutions for mobile ad hoc networks like [11], [8], [10].

In this paper, we propose a distributed solution to the problem of group mutual exclusion coordination, considering that the processes require a time to share the resource in a group. We consider that every group has an associated priority.

II. Preliminaries

Let be a set of n processes P_0, P_1, ..., P_{n-1}; a set of m groups G_0, G_1, ..., G_{m-1} and a unique, non shareable, resource among the m groups. The processes may work alone or in cooperation with others processes in a group. Any of the n processes is able to participate in a group. Only one group at a time is allowed to use the resource.

Initially each process works alone. When the process wants to work in a team, it selects a group. Each process may select any of the different groups with a finite time of work in the team. Figure 1 shows an example of relation between the groups and the

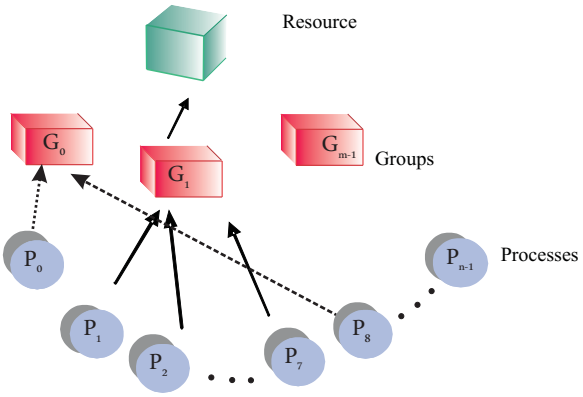


Fig. 1. Example of Relation between the players

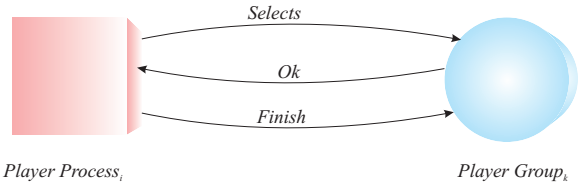


Fig. 2. Communication between the players

processes; where P_1, P_2 and P_7 are linked to the group G_1, this is active and has the permission to use the resource. That means that all the processes are using the resource concurrently. Processes P_0 and P_8 are linked to the group G_0 that is competing to gain the access to the resource.

The model of two players, posed in [2], proposed a general solution to this problem using two players: *groups* and *processes*. Figure 2 shows the communication between the players. The *processes* are active players and the *groups* are passive players. The relation between the players is temporary. When the player group is activated, the competition to access the resource begins.

The design of a solution for this problem requires an algorithm that satisfies the followings requirements.

- Mutual Exclusion: if some process is in a group, then no other process can be in a different group simultaneously.
- Bounded Delay: a process attempting to participate in a group will eventually succeed.
- Progress: when the resource is available (the critical section is empty), and some groups are waiting. At some later point one group gain the access to the resource.
- Concurrent Entering: if some processes are interested in a group and no process is interested in a different group. Then the processes can participate in the group concurrently.

A. Coteries

Definition 2.1: Coterie [1]. Let U = $\{G_1, ..., G_m\}$ 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 $S_i \in C$, S_i is not empty and $S_i \subseteq U$.
2) Intersection property: for any S_i, $S_j \in C$, $S_i \cap S_j$ is not empty.
3) Minimality: for any S_i, $S_j \in C$, S_i is not a subset of S_j.

Coterie is a set of quorums, and a quorum is a subset of processes. By the intersection condition, the coterie can be used to develop algorithms for mutual exclusion in a distributed system. To enter the critical section, a node is required to receive permissions from all the members of some quorum in the system. Since any pair of quorums have at least one member in common, mutual exclusion is then guaranteed.

III. Algorithm GBP (Group Base Priority)

This section presents a solution to the problem of group mutual exclusion including time associated with the players (groups and processes), using messages for the communication. Applying the model of the two players [2] to this situation, we obtain the following:

- When the player process wants to participate in a group, first specifies his time and then selects the group. Waits until the group allow the access.
- At the moment the player group activates, its assigns the time of the first process to use the resource.
  While the player group is waiting to access to the resource (entry section):

  When a request from a player process arrives, adds the request to the active queue and compares the duration of the process with the group duration. If it is greater, then sets the duration to the maximum duration of the new player process.

  While the player group is using the resource (critical section):

  When a request from a player process arrives, compares if the duration of the process in not greater than the remainder (group duration - elapsed duration). Then adds the request to the active queue and accepts the request. Otherwise adds the request to the waiting queue until the next stage.

The time associated with each player does not represent deadline time but represents its duration in the critical section. In distributed environment, we have to consider the communication time (time delay). We assume a reliable network, with a estimated

communication time tc, and a finite time of use of the resource. The communication time is necessary to adjust the remainder time, to accept or not a new player process while the player group is in the critical section. We define the following variables:

- $tc_{i,k}$: Delay estimation of the communication between the group G_k and the process P_i
- $tpodur_i$: Process time associated to the group in a stage
- $gtpo_k$: Group time in a stage

When the player group receives a request from a process and it is in the critical section, the acceptance control for a new process is the following: $tpodur_i$ $<$ (remainder $time_k$ - $tc_{i,k}$), where remainder $time_k$ = ($gtpo_k$ - $tpouse_k$)

When the time group finishes, we consider the following options:

1) Wait until all the associated processes release the group.
2) Inform the associated processes in order to finish their associations. We could consider different possibilities.
   a) Waiting for all of the process acknowledgment. The delay time could be unpredictable.
   b) Release the critical section (release the resource) and allow another group to access. This option avoids the waiting time, but the notification could be delayed to a associated process and still continue using the resource while a different group gains the access to the critical section. This situation does not guarrantee the group mutual exclusion property.

In accordance with the constrains imposed for allowing the active association in the group, we assume that the group does not take into account its own time. When each process finishes, the group is inform. When the group is empty of participant processes it releases the critical section. This option simplifies the solution and it is acceptable because the associated time is not critical.

Figure 3 shows the actions of the player process. The process, in each stage, sends two messages to the group and receives one message from the group.

- Req-Process (G_k, P_i, $topdur_i$): the process P_i sends a request message to the group G_k to participate in during a period $topdur_i$.
- Rep-Process (G_k, P_i): the process P_i receives the reply of his request from G_k, that allows the access to the resource.
- Rel-Process (G_k, P_i): the process P_i sends a message to the group G_k to inform that the period in the group has finihed and it is unlinked.

The time that the process stays in the critical section is $tpodur_i$ and then releases its association

process P_i
  $RemainderSection$
    ...
  $EntrySection$
    G_k = chosen group
    $tpodur_i$ = chosen the time to use the resource
    send Req-Process (G_k, P_i, $topdur_i$)
    receive Rep-Process (G_k, P_i)
  $CriticalSection$
    ... *duration* $tpodur_i$
  $ExitSection$
    send Rel-Process (G_k, P_i)

Fig. 3. Actions of Player Process P_i

Variables
  $state$ (INACTIVE, ACTIVE, CS, EXIT)
  LP: keeps information of all the linked process.
  LG: keeps information of all waiting requests of $lock$.
  $gtpo_k$: keeps the time to use the resource.
  priori: keeps the base priority of the group in the stage.

Fig. 4. Variables Group G_k

with the group.

Figure 4 shows the variables of the group G_k. Figure 5 shows the actions of the player group associated with the process and figure 6 shows the actions with the other groups. The states of the group are the followings: INACTIVE: is waiting for participating processes, ACTIVE: is waiting to access the resource, CS is using the resource and EXIT is releasing the resource. Each linked process has the same priority, and each group has an associated priority. Two different groups could have the same priority. The proposed protocol is based on priority without prompt meanwhile the group with lower priority is using the resource.

The group communicates with the associated processes and with the other groups. The messages received from the process are:

- Req-Process(G_k, P_i, $tpodur_i$) this message is received from a process, if the group is INACTIVE then changes its state to ACTIVE, adds the request to the list LP and sets the length of time of the group ($gtpo_k$) with the length of time of the process ($tpodur_i$). If the group is ACTIVE the request is added, and the length time of the group is checked with the length of time of the process. If it is lower then it sets its current time length with the process time length. If the group is in CS it adds the request and checks the remaining time group with the length of time process. If it is greater it accepts the process request and allows to participate in this stage. Otherwise the process has to wait for the next stage.
- Rel-Process(G_k, P_i) this message comes from a process to release his link with the group. Removes the request from the list LP. If the

```
group G_k
  ♢ Receive Req-Process(G_k, P_i, tpodur_i)
    case state of
      "Inactive": gtpo_k = tpodur_i;
                  state = "Active";
                  conj = ∅;
                  priori = chosen priority;
                  AddLp(LP, P_i);
                  AddLG(LG, G_k, priori);
                  send multicast Req-Grupo(G_k,
                  prior);
      "Active": if tpodur_i > gtpo_k then
                  gtpo_k = tpodur_i;
                  AddLp(LP, P_i);
      "SC": AddLp(LP, P_i);
            if tpodur_i ≤ (gtpo_k - tpouse_k - tc_{i,k})
            then
              send Rep-Process(G_k, P_i);
      "Exit": AddLp(LP, P_i);

  ♢ Receive Rel-Process(G_k, P_i)
    DeleLp(LP, P_i)
    if activeempty(LP) then
      state = "Exit";
      send multicast Lib-Group(G_k);
      G_l = SelectGroup(LG);
      send Rec-Group(G_l, G_k);
      state ="Inactive";
      if not empty(LG) then
        state = "Active";
        conj = ∅;
        priori = chosen priority;
        send multicast Req-Group(G_k,
        priori)
```

Fig. 5. Actions of Player Group G_k with Player Process

```
♢ Receive Req-Group(G_l, priori)
  if empty(LG) then
    AddListGroup(LG, G_l, priori);
    send Rec-Group(G_k, G_l)
  else
    if HigherPriori(LG, G_l, priori) then
      G_s = findHigh(LG);
      send Rel-Group(G_s, G_k);
      AddListGroup(LG, G_l, priori);
      upgrade(priori);
    else
      AddListGroup(LG, G_l, priori);

♢ Receive Rec-Group (G_l, G_k)
if G_l ∉ conj then
  conj = conj ∪ {G_l}
  if |conj| = |S_k| then
    state = "CS"
    For each process in LP do
      send Rep-Process(G_k,P_i)

♢ Receive Rel-Group (G_l, G_k)
  if state ≠ "CS" then
    conj = conj - {G_l};
    send Rep-Rel-Group(G_k, G_l);

♢ Receive Rep-Rel-Group (G_l, G_k)
  G_s = findHigher(LG);
  send Rec-Group(G_k, G_s)

♢ Receive Lib-Group (G_l)
  DeleListGroup(LG, G_l)
  if not emptyListGroup(LG) then
    G_s = findHigher(LG);
    send Rec-Group(G_k, G_s)
```

Fig. 6. Actions of Player Group G_k

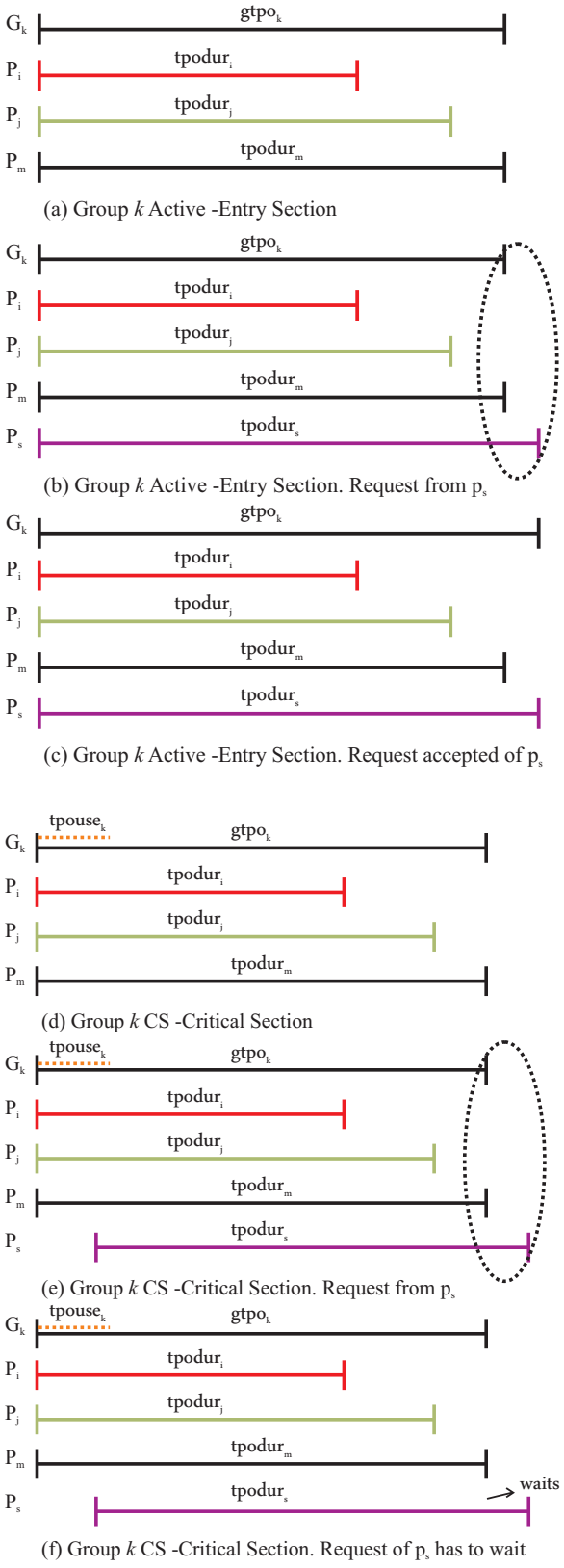
list LP is empty of active process then releases the resource. If waiting processes exists in the list then the group begins a new stage.

The messages received from the others groups are:

- Req-Group(G_l, priori) this message comes from group G_l that requires the *lock*. The group G_k grants the lock if available. If the lock is not available two different cases may occur: (a) The priority of the received message is less than the priority of the message given the lock then the request is delayed. (b) If the priority is greater then calls the lock to the appropriate group and then grant it the highest priority.
- Rec-Group(G_l, G_k) this message comes from group G_l, affirmative response to the message *Req-Group* of requirement lock. If the group G_k have all the locks then change the state to CS and tell all the processes that are linked to the group.
- Rel-Group(G_l, G_k) this message comes grom group G_l requiring the lock, this will be successful if the group G_k is not in the critical section.
- Rep-Rel-Group(G_l, G_k) this message comes from group G_l releasing the lock that had given the group G_k. The lock is granted the highest priority requirement.
- Lib-Group(G_l) this message comes from the group G_l that releases the lock. If there are outstanding requirements, then choose the highest priority and gives the lock.

Figure 7 (a) shows the state of the group G_k (ACTIVE) with their linked processes P_i, P_j and P_m, the time of the group is equal to the time of process P_m. Figure 7 (b) shows when arrives a new request from process P_s to link the group with a time ($tpodur_s > gtpo_k$), since the group is in the ACTIVE state, sets the value of its time with the time of P_s, figure 7 (c) shows this modification. Figure 7 (d) shows the state of the group G_k (CS) with their linked processes P_i, P_j and P_m, and the $tpouse_k > 0$. Figure 7 (e) shows when arrives a new request from process P_s to link the group with a time $tpodur_s$. Since the time $tpodur_s$ is greater than ($gtpo_k$ - $tpouse_k$ - $tc_{s,k}$) and the group is in SC then the process P_s has to wait for the next stage, figure 7 (e) shows this case.

The messages among the groups correspond to the competition to gain the access of the resource. The algorithm uses messages to obtain the permissions



(a) Group *k* Active -Entry Section

(b) Group *k* Active -Entry Section. Request from p_s

(c) Group *k* Active -Entry Section. Request accepted of p_s

(d) Group *k* CS -Critical Section

(e) Group *k* CS -Critical Section. Request from p_s

(f) Group *k* CS -Critical Section. Request of p_s has to wait

Fig. 7. Concurrency

from the other groups. Each group has associated a quorum (set of groups) to request the permission of access (S_k). To select the quorum, we use the Maekawa method [6], the size of the quorum is $\sqrt{m}$, where m is the number of groups. When the group obtains all the permissions the resource can be use and this is informed to his associated processes.

IV. Correctness

In this section, we show the correctness of the proposed algorithm. The algorithm satisfies the properties of mutual exclusion, progress and concurrent entering.

Theorem 4.1: The proposed algorithm ensures mutual exclusion.

Suppose processes P_i and P_j can access the critial section at the same time, where $P_i \in G_k$, $P_j \in G_l$, $G_k \neq G_l$. Thus, two processes are in the critical section linked with different groups. If this occurs, then the group G_k receives all the locks from his quorum S_k and the group G_l receives all the locks from his quorum S_l. For the condition (2) of the definition of coterie, $S_k \cap S_l \neq \emptyset$, then occurs that the member of the two quorums grants the lock to 2 requirements. This is a contradiction.

Theorem 4.2: The proposed algorithm ensures bounded delay.

Suppose a process P_i makes a request to the group G_k and is waiting indefinitely. Each request has associated a priority (G_k, ts, priori). This priority eventually will be the higher and grants the access to the critical section. Besides, we consider that each group does not stay indefinitely in the critical section for the arrival of new processes, through the time of the process (topdur$_i$) and the remainder time of the group (gtpo$_k$ - tpouse$_k$ - tc$_{i,k}$).

To reduce the waiting time, we consider:

- When a group with a lower priority is using the resource, the other groups with higher priority that wants to access must wait.
- When a group is using the resource (state = "CS") accepts new requests of processes only if the time associated is less than his remainder time. With this consideration, none of the groups stays indefinitely using the resource.

Theorem 4.3: The maximun concurrency of the proposed algorithm is n.

When each process makes a request for the same group simultaneously, all of the requests are added to the active queue. When the group grants the locks of his quorum can access concurrently the n processes.

V. Complexity

The complexity of the algorithm can be measured using different topics, like the number of access to

shared memory, the delay time between entries in the critical section and the number of exchanged messages. The election of the measure depends on the type of the algorithm.

The complexity of the algorithm is measured in function of the number of the messages requires.

Let $q = |S_k|$, in the best case, each group requires for gain the access and release $3(q-1)$ messages, where $(q-1)$ for request the permission, $(q-1)$ for grant the permission, $(q-1)$ for release the permission. If it has associated: one process, in total requires $3+3(q-1)$; l processes, in total requires $3l+3(q-1)$. If in average, each group has to yield once so, the number of messages requires are $5(q-1)$, where $(q-1)$ for yield the permission, $(q-1)$ for grant the permission, with l associated processes in total requires $3l+5(q-1)$. If each group has to yield the permission at most p times, then requires with l associated processes $3l+3(q-1)+2p(q-1)$ messages. With the maximum concurrency, n request for the same group simultaneously, in total requires $3n+3(q-1)$ messages.

Although the number of messages for a request that is unique in a group is larger than $3q+1$ in [4], the proposed algorithm requires less messages for l simultaneously requests to the same group.

VI. Conclusion

In this paper we proposed a distributed solution for group mutual exclusion considering that processes have an associated time to share the group. This should be the duration they will cooperatively work in the group in each stage. The algorithm is based on priorities over the groups with no prompt. The communication among the processes and groups uses messages. The groups have assigned a quorum, that is use in the competition to get the permissions to access the resource.

The algorithm guarantees mutual exclusion, progress, bounded delay and concurrency. In the best case, where the group does not yield the permission, with l processes linked, requieres $3l+3(q-1)$ messages. The maximum concurrency of the algorithm is n, which implies that all processes have linked to the same group.

References

[1] D. Barbara and H. García-Molina. Mutual exclusion in partitioned distributed systems. *Distributed Computing*, 1:119–132, 1986.

[2] K. M. Cenci and J. Ardenghi. Modelos dos actores para grupos de procesos. In *XIV Congreso Argentino de Ciencias de la Computación (CACIC 2008)*, 2008.

[3] Y. J. Joung. Asynchronous group mutual exclusion (extended abstract). In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing (PODC'98)*, pages 51–60, June 1998.

[4] Y. J. Joung. Quorum-based algorithms for group mutual exclusion. *IEEE Transactions on Parallel and Distributed Systems*, pages 463–476, May 2003.

[5] H. Kakugawa, S. Kamei, and T. Masuzawa. A token-based group distributed group mutual exclusion algorithm with quorums. *IEEE Transactions on Parallel and Distributed Systems*, 19(9):1153–1166, September 2008.

[6] M. Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, May 1985.

[7] Y. Manabe and J. Park. A quorum-based extended group mutual exclusion algorithm without unnecessary blocking. In *Proceedings of th Tenth International Conference on Parallel and Distributed Systems*, pages 341–348, 2004.

[8] J. F. Myoupo, M. Naimi, and O. Thiare. A clustering group mutual exclusion algorithm for mobile ad hoc networks. In *IEEE Symposium on Computers and Communications, 2009. ISCC 2009.*, pages 693–696, 2009.

[9] Gurdip Singh and Ye Su. Efficient synchronization in message passing systems. In *22nd International Conference on Advanced Information Networking and Applicaciones*, pages 219–226, 2008.

[10] A. Swaroop and A. K. Singh. A token-based group mutual exclusion algorithm for cellular wireless networks. In *Annual IEEE India Conference (INDICON)*, pages 1–4, 2009.

[11] O. Thiare, M. Gueroui, and M. Naimi. Distributed groups mutual exclusion based on clients/servers model. In *Proceedings of the Seventh Internacional Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 67–73, 2006.

[12] M. Toyomura, S. Kamei, and H. Kakugawa. A quorum-based distributed algorithm for group mutual exclusion. In *Proceedings of the Fourth International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 742–746, 2003.

[13] K. P. Wu and Y. J. Joung. Asynchronous group mutual exclusion in ring networks. In *13th International Parallel Processing Symposium / 10th Symposium on Parallel and Distributed Processing (IPPS / SPDP '99). Proceedings. IEEE Computer Society*, pages 539–543, April 1999.

Received: December 2010. Accepted: February 2011.