PERMISSION BASED GROUP MUTUAL EXCLUSION ALGORITHMS FOR A CLUSTER TREE NETWORK

Touomguem Nzeumbeu Arlette Sylvie[1], Kamla Vivient Corneille[2], and Tcha'wu Gatsing Elvire Cheryl[1]

[1]Departement of mathematics and Computer Science, Faculty of Science, the University of Ngaoundere, Cameroon
[2]Departement of mathematics and Computer Science, National School of Agro-industrial sciences, the University of Ngaoundere, Cameroon

ABSTRACT

Due to the growing application of peer to peer computing, distributed applications are continuously spreading over an extensive number of nodes. To cope with this large number of participants, various hierarchical cluster based solutions have been proposed. Cluster or group based solutions are scalable for a large number of participants. As far as group mutual exclusion solutions are concerned, some of them have good complexity but do not take into account the growing number of participants. Others take into account the previous aspect but do not have good complexity. In this paper we present two group mutual exclusion algorithms namely; ***TBGMEACα*** *and* ***TBGMEAC$_\beta$*** *to deal both with the growing number of nodes and the matter of having a good complexity. The proposed logical structure is a cluster tree. The first solution uses the partial flooding method, which is the partial propagation of informations available at root level. In the second solution, informations available at root level are propagated only towards the processes which issued a request for a session. Both algorithms have complexities of O(p) and O(log p) respectively (p is the number of clusters in the system).*

KEYWORDS

Distributed system, resource management, group mutual exclusion, cluster tree network & partial flooding.

1. INTRODUCTION

1.1 Preliminaries

In a computer system, a process executes one part of his program called critical section (CS) when it accesses a shared or critical resource. For certain shared resource, the critical section of a process can be associated to a type or a group. Critical sections belonging to the same group can be executed concurrently while critical sections belonging to different groups must be executed in a mutual exclusive manner [1]. Thus, the problem of shared resource appear, when different groups of processes have to access exclusively to a critical resource. That problem is known since many decades in computer science research community as mutual exclusion problem. In terms of the maximal number of processes in a group, two classes of mutual exclusion problem are distinguished in the literature. The first class, called the basic mutual exclusion problem [2, 3, 4,5, 6, 7, 8, 9], where the maximum size of groups is one, and the second class called group mutual exclusion (GME) problem, where every group can have more than one process [1,10,11,12, 13, 14, 15, 16].

The focus of this paper is the GME problem. The issue of this problem is to manage a mutually exclusive resource among n processes. Processes are allowed to be in critical section simultaneously only if they belong to the same group. A conference room, classroom, CD-jukebox, Internet server, and railway are examples of GME problems. At least one conference can take place in the conference room at the same time. In this first example, participants are processes attending the conference, conferences are groups, and the conference room is the critical resource. With regard to a classroom, only one lesson can be given at the same time. In this second example, students are processes attending the lesson, lessons are groups, and the classroom is the critical resource. The example of CD-juke (resp. Internet server) is well described in [11, 12] (resp. [13]). Regarding the example of the railway, it has only two sides and trains can leave from one side to another, but only one side is opened at the same time. In this last example, and in relation with the GME problem, trains are processes, the two sides of railway are groups, and the railway is the critical resource.

1.2 Related work

Several solutions of the GME problem in computer system exist. The main challenge of these solutions resides in the number of exchanged messages through the computer system in order to avoid concurrent use of a critical resource by different groups. Many existing solutions for the GME problem can be classify in permission based and token based categories. Algorithms for the first category can be found in [11, 17] for the shared memory model and in [12, 18, 19, 20, 21] for the message passing model. In [20, 21] solutions are provided for fully connected network while in [18, 19] they are provided for ring network. Solutions have been also proposed for tree network [12] where processes are assumed to be arranged in the form of a tree and groups of processes are called sessions. Any process interested in session y can enter and exit the session any number of time till a session $x \neq y$ is requested. In [12], three algorithms have been proposed namely GMEα, GMEβ and GMEγ. GMEα has a message complexity of $3\ (N-1) + h_1$ where h_1 is the initial tree height. GMEβ is an improvement of GMEα. The message complexity here has been reduced to $4h_1$. GMEγ, which is an improvement of GMEβ has a message complexity of $4h_{max}$ (h_{max} is the Maximum height of the tree). The trick used here is that the root is no longer a fixed root like in GMEα, and GMEβ; any process can become the root. For the token based category, solutions can be found in [19, 22, 23]. The solution in [22] uses the notion of priority. To ensure the GME two tokens are used; a process with the primary token can send the secondary token to a process wishing to access the same session, while the process holding the secondary token cannot do so. The algorithm requires $2(N-1)$ messages per entry in the critical section where N is the number of nodes. The algorithm in [19] is for an asynchronous ring and its message complexity is N^2. A solution has also been provided for hierarchical structure [23]; here processes are grouped to form clusters which in turn are used to build a two-level hierarchy. The algorithm uses the notion of coordinator and requires that these are all connected. Its message complexity is O (p), where p is the number of clusters.

1.3 Our contributions

We present two permission based group mutual exclusion algorithms for cluster tree networks (***TBGMEAC***) ; both solutions have an unbounded degree of concurrency and use fixed root. The first solution ***TBGMEAC***$_\alpha$ uses the partial flooding method, which is the partial propagation of informations available at root level. The message complexity is $(4h_0 + h_{max} + 3\ (P-1))$ where p is the number of clusters, h_{max} the maximum height of the cluster tree and h_0 the distance between a node and its coordinator. In the second solution ***TBGMEAC***$_\beta$, informations available at root level are propagated only towards the processes which issued a request for a session. The message complexity is $4\ (h_{max} + h_0)$.

1.4 Paper organisation

We present our system model and formally describe the group mutual exclusion problem in section 2. We describe our group mutual exclusion algorithm for a cluster tree network in section 3. In section 4, we give the correctness proof of our GME algorithms. Finally, we present our conclusion, and future works in section 5.

2. MODEL AND PROBLEM DEFINITION

2.1. System model

We assume an asynchronous distributed system that consist of *N* nodes numbered 1, 2 …*N*; the only way of communication between different nodes is message exchange. We also assume that each node can communicate directly with every other and that the links are both FIFO (First In First Out) and infallible. There is neither overall time in the system or shared memory. The propagation delay of messages between two nodes is finite but unpredictable. A process should not issue a new request before the old query is satisfied. In the following we assume that the terms node, site and process designate one and the same thing; the nodes are also assumed not to crash.

In the network, nodes are logically partitioned into separated groups and each group is called cluster; clusters are then arranged to form a tree. *N* nodes are thus divided into *P* clusters numbered p_1, p_2…p_p, each cluster containing N_i nodes such as the following equality is verified:$\sum_{i=1}^{p} Ni = N$. Each node is represented by its identifier and there is no need to mention the reference of the cluster to which it belongs, in the course of algorithm. In every cluster, a node is responsible for the task of leading and has as its parent the cluster leader at the top level, except for the case of the root cluster leader. A hierarchy of three levels consisting of ten clusters and sixty-one nodes is presented in the figure 1 below.

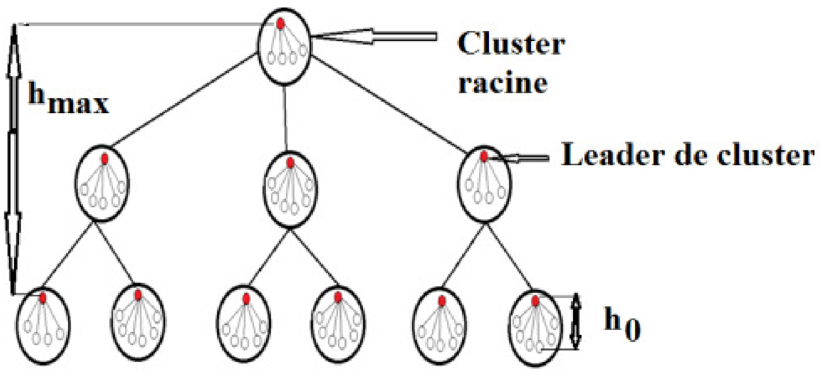

Figure 1. A three level hierarchy cluster tree network

2.2. The Group Mutual Exclusion Problem

The group mutual exclusion (GME) problem is an extension to the basic mutual exclusion problem. It sets the problem of sharing m resources in mutual exclusion by n processes; a group is then associated with each resource type and critical sections belonging to the same group can run concurrently, while critical sections belonging to different groups must be executed in mutual exclusion. The properties of GME algorithms are:

Group mutual exclusion (safety): At any time, no two processes that have requested critical sections belonging to different groups are in their critical sections simultaneously.
Starvation freedom (liveness): A process requesting entry into its critical section should eventually be able to enter the critical section.
Concurrent entry (non-triviality): If all requests are for critical sections belonging to the same group, then a requesting process should not be required to wait for entering its critical section until some other process has left its critical section.
To measure the performance of a group mutual exclusion algorithm, we use the following metrics:
The message complexity: it is a measure of the number of messages generated per entry to the critical section.
The synchronization delay: the time elapsed between when some process leaves its critical section and some other process can enter its critical section of different type.
The waiting time: the time elapsed between when a process issues a request for critical section and when it actually enters the critical section.
The system throughput: the number of critical section requests fulfilled per unit time.
The concurrency: the number of processes that are in their critical sections at the same time.

3. OUR ALGORITHMS

Our algorithms are derived from the GME_β solution proposed in [12]. We modified GME_β extending it to take into account the growth in the number of participants while maintaining good message complexity.

3.1 Overall System Architecture

We assume that there are two layers in the system such as, the application layer (upper layer) and the GME layer (lower layer). The interface between these two layers is implemented using three types of messages: Request_Session, Grant_Session and Exit_Session. Thus, when a process *p* running at the application layer solicits any *X* session, it sends *Request_session (X)* to the GME layer. Eventually, the GME layer grants access to it by sending *Grant_session (X)*; at the end of the use of the session *X*, *p* sends *Exit_session (X)* to GME layer.

To maintain the hierarchy processes are divided into two classes such as the non-leaders and leaders processes. Each process maintain two pointer variable. The non-leaders processes of each cluster are all connected to their leaders which are their parents. They each maintain a pointer variable par_p that contains a value in N_p. Since these processes are leaves in each cluster, they therefore have no descendants. Leader processes in turn maintain a variable par_p that contains this time a value in $N_p \cup Nil$. If $par_p \in N_p$ then, par_p contains the link label of a particular neighbor of p, called the parent of p. Another variable, D_P, is a subset of $N_p \setminus \{Par_p\}$, called the set of descendants of p. The parent pointer of processes form an oriented spanning tree rooted in a particular process, referred to as the root of the spanning tree, such that $Par_{racine} = nil$.

3.2. The first algorithm: $TBGMEAC_\alpha$

The algorithm proceeds in two phases that are opening and closing of sessions. These phases are initiated by the root.

Suppose a process *P* makes a first request (the first in the system) for a session *X*, which is sent to the cluster leader through the message *ASK (X)*. The leader records the session *X* as the requested session and, sends the request to the next level cluster leader; so, from leader to leader the message finally reach the root. Once the root receives the *ASK (X)*, it starts the opening operation of session *X* by sending *OS (X)* not only towards the process that is causing the *ASK (X)*, but also to any leader which received no request from their descendants. It is this technique that we call partial flooding. To achieve this, we introduce a new variable *New_Openset* which will contain both all processes that requested a session (the set *Openset*), and any leader in which did not receive any request from its descendants. By doing so, those of such leaders who will later receive requests will in turn serve only those of their descendants which issued requests for the current session.

To identify the leaders which have not requested (or received requests from their descendants) session, we must distinguish in a leader's descent, leaders of lower- level cluster from proper members of the cluster. So a leader must have as descent $D = D_c \cup D_{inf}$ where D_c contains the descendants belonging to the cluster of the corresponding cluster leader and D_{inf} contains the descendants which are leaders of lower-level clusters so, $New\_Openset = Openset \cup D_{\text{inf}}$. From the moment the root propagated the opening session message for *X*, process *p* or other processes can enter and exit the critical section as many times as they wish.

Now we suppose that another session, say $Y \neq X$ is requested by another process, say *q*; the request is routed to the root, and it begins the second phase which is that of the closing of *X* by sending the message *CS (X)* to all its descendants contained in *New_Openset.* Leaders do the same for their descendants. Upon receipt of *CS (X),* p (or other processes) and the leaders which received no request from their descendants must send a message to confirm that they have exited session *X*. At this point, it is possible that the message *CS (X)* reaches a process while it is still in critical section; in this case, it will delay sending his confirmation and will only do that once out of the critical section. Once the root has received all the confirmations of the closing of session *X,* it organizes the opening operation of session *Y.*

Since ***TBGMEAC***$_\alpha$ is derived from GME_β, as a description of ***TBGMEAC***$_\alpha$, we are going to show the parts which have been modified to obtain our algorithm. For this we present the used procedures and the used messages firstly for the leader side, then for the non-leader side.

Procedures from the leader's side

```
    Procedure OpenSession (S)
 1: Open := true ;
 2: Current S :=S ;
 3: I : = Get (ReqQ, S) ;
 4: OpenSet := I.NS ;
 5: New_OpenSet := $OpenSet \cup \{p \in D_{inf} / p \notin OpenSet\}$ ;
 6: for all q $\in$ New_OpenSet do
 7:       Send OS (S) to q ;
 8: end for
 9: if ReqS= Current S then
10:    send Grant_Session to App layer ;
11: end if ;
12: Remove (ReqQ, I) ;
    Procedure CloseSession
 1: Open := false ;
 2: for all $q \in$ New_OpenSet do
 3:    send CS(S) to q ;
 4: end for
 5: TestDone ;
```

Procedures from the non-leader's side

```
Procedure C_OpenSession (S);
1: Open := true;
2: CurrentS := S;
3: Send Grant_Session to App layer;

Procedure C_CloseSession (S)
1: Open : = false;
2: C_TestDone;
```

Message section executed by the leaders

```
Upon receipt of ASK(S) from q
1: if $Open \wedge CurrentS = S$ then
2:     $New\_OpenSet := New\_OpenSet \cup \{q\}$;
3:     send OS(S) to q;
4: else
5:     ReqFlow$(S, \{q\})$;
6: end if

Upon receipt of DONE from q
2: $New\_OpenSet := New\_OpenSet \setminus \{q\}$;
3: TestDone;
```

Message section executed by the non-leaders

```
Upon receipt of C_Request_Session (S) from App layer
1: ReqS := S;
2: if CurrentS= S then
3:     send Grant_session to App layer;
4: else
5:     send ASK (S) to par;
6: end if

Upon receipt of C_Exit_Session from App layer
1: $ReqS := \perp$;
2: C_Test Done;

Upon receipt of OS (S) from q
1: C_OpenSession (S);

Upon receipt of CS (S) from q
1: C_CloseSession (S);
```

Complexity analyses

Message complexity: We will consider two cases to compute the message complexity. Assume that a process *P* requests the current opened session *X*; in this case it needs $2h_0$ messages because the opening session message have also been sent to the all the leaders in the network. Assume now that *P* requests a session $Y \neq X$; this generates $(4h_0 + h_{max} + 3\ (P-1))$ messages that is, $h_0 + h_{max}$

messages to send the request of *q* to the root, 2($(p\text{-}1) + h_0$) messages for the closing of the current opened session (*X*) and ($(p\text{-}1) + h_0$) for the opening of session *Y*.

The synchronization delay: this complexity measure is considered in our algorithm from the point of view of sending and receiving messages and generates $3(h_0 + (p\text{-}1))$ messages that is, ($(p\text{-}1) + h_0$) messages from the root to the last process to exit the current opened session, ($(p\text{-}1) + h_0$) messages from that process to the root, and ($(p\text{-}1) + h_0$) messages from the root to the waiting process to enter the new opened session.

The waiting time: this complexity is also considered as for the previous complexity measure. Assume that *P* wants to access the current opened session; it will have to wait for $2h_0$. Assuming now that it wants to access a different session *Y*, it will have to wait T $(4h_0+ h_{max} +3 (P\text{-}1))$, where T is the transfer time of a message.

The degree of concurrency: the degree of concurrency is unbounded because, since a session is opened processes can enter and exit the critical section as many time as they want.

3.3. The second algorithm: $TBGMEAC_\beta$

Here we present an algorithm which reduces the message complexity from $(4h_0+ h_{max} +3 (P\text{-}1)$ to $2(h_{max} + h_0)$, that is from $O (P)$ to $O (log$ p$)$. The astuteness used to do this is that once the root receives a request, it sends the consequent messages only towards the processes which issued a request. This algorithm also proceeds in two phases. Assume a process *P* requests a session *X*; the request is sent towards the root as in the first algorithm. Then root then starts the opening of session *X* by sending the *OS (X)* message only towards the process which issued a request for *X*; this time, *P* can enter and exit session *X* as many times as it wants. Assume now that *q* wants to access a session $Y \neq X$; the request normally reaches the root which starts the closing of session *X* by sending the message *CS (X)* towards the process which issued the request for *X*. Here also, process *p* can delay sending the exit message from session *X* and send it only once out of the critical session. Once the closing session message now reaches the root, it starts the opening of session *Y*.

As the description of the pseudo code of $\boldsymbol{TBGMEAC_\beta}$ the leaders are going to execute entirely GME_β, while the non-leaders are going to execute the same code presented previously for the non-leaders side.

Complexity analyses

Message complexity: We will consider two cases to compute the message complexity. Assume that a process *P* requests the current opened session *X*. in this case it needs $2h_0$ messages because the leader has already recorded *X* as the current session. In the case the request is the first in the system, this generates 2 $(h_{max} + h_0)$. Assume now that *P* requests a session $Y \neq X$; this generates 4 $(h_{max} + h_0)$ messages that is, $(h_{max} + h_0)$ messages for the request to reach the root, 2 $(h_{max} + h_0)$ messages to close the current opened session and $(h_{max} + h_0)$ messages to open the new requested session.

The synchronization delay: this complexity measure is considered in our algorithm from the point of view of sending and receiving messages and generates $3(h_0 + h_{max})$ messages that is, $(h_0 + h_{max})$ messages from the root to the last process to exit the current opened session, $(h_0 + h_{max})$ messages from that process to the root, and $(h_0 + h_{max})$ messages from the root to the waiting process to enter the new opened session.

The waiting time: this complexity is also considered as for the previous complexity measure. Assume that *P* wants to access the current opened session; it will have to wait for $2h_0$. Assuming now that it wants to access a different session *Y,* it will have to wait $4T\ (h_0+ h_{max})$ where T is the transfer time of a message.

The degree of concurrency: the degree of concurrency is unbounded because, since a session is opened processes can enter and exit the critical section as many time as they want.

4. CORRECTNESS PROOF

For a GME algorithm to be deemed correct, it must satisfy the properties of safety, liveness and concurrent entry.

4.1 Safety

Here we want to show that if two processes *p* and *q* are running simultaneously their critical sections S_p and S_q then $S_p= S_q$. for this we designed the lemma and theorem that follow.

Lemma 4.1: If two leaders say C_{pi}, and C_{pj} grant access to a session in their respective clusters, then they do so for the same session.

Proof: Only the root is responsible of both the opening and the closing of sessions; thus in one case or another, the root sends the same information to all its descendants in *New_Openset*. By this way the *lemma 4.1* holds.

Theorem 3.1: If two processes *p* and *q* are simultaneously in critical section, then they are for the same session.

Proof: The only condition for a process to be in critical section is to receive the message *OS(X)* from its leader. According to *lemma 3.1* whichever the leader, the opened session is the same; consequently processes which are concurrently in critical section are for the same session.

4.2 Liveness

Here the purpose is to proof that if a process, say *p* wants to access a session then, it finally executes its critical section.

Lemma 4.2: A request sent by a leader, say C_{pi} eventually reaches the root or a leader which has received the message *OS(X)*.

Proof: Once a leader receives the message *ASK (S)* from its descendants, it has to do the test *"if Open^Current S=S"* that is to say, if the current opened session is *S*. In case of unfavorable opinion, the leader has to send in turn an *ASK (S)* message to its parent; in the case the opinion is still unfavorable a leader has to react as previously until the request reaches the root. By this way the request of a leader will be satisfied in case of favorable opinion or in the case its request reaches the root. Consequently, a request sent by a leader, say C_{pi} eventually reaches the root or a leader which has received the message *OS(S)*.

Lemma 4.3: If there is a conflicting request in the system, then the current opened session terminates in a finite time.

Proof: If there is a conflicting request in the system, then according to *lemma4.2* it will finally reach the root or a leader which have received the message *OS(S)* in a finite time.

Again, in of reception of a request for a session different from the current session the root stops sending *OS(S)* messages to its cluster members and to any other process which requested the current session. It after sends the closing session messages to its descendants; these messages will reach the descendants at a finite time as it was the case during the forwarding of the request to the root, according to *lemma4.2*. By the same way, the descendants in turn will send the closing session messages towards the root in a finite time. Consequently, if there is a conflicting in the system, then the current session terminates in a finite time.

Theorem4.2: If a process *p* requests a session *X*, then *p* finally enters in critical section.

Proof: Assume that *p* can suffer from starvation. *P* can suffer from starvation if its leader did not receive the message *OS (S)* after it received the request.
This can happen in the following cases:

- **i)** The request of the leader did not reach the root or any other process which received the message *OS (S)*. This is not possible according to the *lemma4.2*.
- **ii)** The request of the leader reached the root but the requested session has not been chosen as the next session to be opened.

The case **(ii)** is possible in the following cases:

- **a)** The current session did not terminate in a finite time. This is not possible according to *lemma4.3*.
- **b)** The other sessions different from the current session are selected one after another.

Since the links are FIFO, the root will not select the sessions different from *X* during an infinite long time after the request for *X* has been added to the root's queue. Thus the situations **(a)** and **(b)** are not possible and consequently **(ii)** is not possible. Since **(i)** and **(ii)** are not possible, then our initial assumption is false and consequently the theorem holds.

4.3 Concurrent entry

The aim here is to show that if processes want to access a session and no other processes want to access a different session, then these processes can access the session concurrently. The following theorem demonstrates it.

Theorem 4.3: If processes request a session and no other one request a different session, then all of them execute their critical section concurrently.

Proof: Assume that a process *p* requests a session *X* while no other process requests a session different from *X*. Two cases must be considered at this level:

- **a)** Session *X* is not opened when *p* requests it. According to *theorem 4.2*, once the request of *p* reaches the root, it starts the opening of session *X*; then all the processes that have requested *X* can concurrently enter the critical section once they received the *OS (X)* message.
- **b)** Session *X* is opened when *p* requests it. In this case either *p* has received the OS (X) message, or has not received. In the first case, *p* enters immediately in critical section. In the second case, *p* can concurrently enter in critical section with other processes once it receives OS (X). Consequently the *theorem 4.3* holds.

The previous correctness proof also holds for the second algorithm.

7. CONCLUSION AND FUTURE WORK

We have presented two group mutual exclusion solutions for cluster tree networks. The first solution TBGMEACα is very efficient in terms of concurrency since the degree of concurrency is unbounded; each entry in the critical section generates between 0 and *O(p)* messages. The second solution ***TBGMEAC$_\beta$*** achieves the average number of messages per entry in a critical section in *O(log* p*)*. It also provides an unbounded degree of concurrency. Both ***TBGMEAC$_\alpha$*** and ***TBGMEAC$_\beta$*** use a fixed root.

A comparison of our solutions with the one of Swaroop [23] is given in the table 1.

Table1. Results comparison

Algorithms and authors	Swaroop **HGME** algorithm	Our ***TBGMEAC$_\alpha$*** algorithm	Our ***TBGMEAC$_\beta$*** algorithm
Year	2009	2016	2016
Type	Token based	Permission based	Permission based
Message complexity	*O(p)*	*O(p)*	*O(log* p*)*
Degree of concurrency	Bounded	Unbounded	Unbounded

In our two solutions only the root has the task of opening and closing of sessions even if a process in the root cluster is not interested in the requested session. As future work, we want to rebuild the solution so that it can take into account the dynamic root cluster. Also, we considered that the links are all infallible; it will be very interesting to reconsider that nodes can crash in order to build a fault tolerant solution.

REFERENCES

[1] A. Swaroop and A. K. Singh, (2013) “A Hybrid Algorithm to Solve Group Mutual Exclusion Problem in Message passing Distributed Systems”, International Journal of Computer Applications (0975 – 8887) Volume 67– No.9 pp 51-59.

[2] Aoxueluo, W. Wu, J. Cao, and M. Raynal, (2013) “A generalized mutual exclusion problem and its algorithm”, In Parallel Processing (ICPP), 42nd International Conference, pp 300–309.

[3] J. Lejeune, L. Arantes, J. Sopena and P. Sens, (2012) “Service level agreement for distributed mutual exclusion in cloud computing”, In 12th IEEE/ACM International Conference on Cluster, cloud and Grid Computing (CCGRID’12), pp 180-187.

[4] J. Lejeune, (2014), Algorithme distribué d’exclusion mutuelle: vers une gestion efficace des ressources, PhD thesis, Université Pierre et Marie Curie.

[5] M. A. Rahman and M. Mostofa, (2010), “Permission Based Hierarchical Algorithm for Mutual Exclusion”, Journal of Computers (JCP), Academy Publisher, Vol. 5, No. 12, pp 1789-1799.

[6] M. A. Rahman and M. Mostofa, (2012), “A Permission based Multilevel Parallel Solution for Distributed Mutual Exclusion”, Journal of Computers (JCP), Academy Publisher, Vol. 7, No. 8, pp 1831-1846.

[7] M. Naimi, M. Tréhel, and A. Arnold, (1996) “Distributed mutual exclusion algorithm based on path reversal”, Journal of Parallel and Distributed Computing, Vol. 34, pp. 1-13.

[8] G. Richa, G. Shikha, and D. Pooja, (2011) “NTBCBT: a distributed mutual exclusion algorithm”, International Journal of Peer to Peer Networks (IJP2P), Vol. 2, No. 3, pp. 25-31.

[9] V. C. Kamla, Souleymanou and I. Damakoa, (2014) “PBCBT: An improvement of NTBCBT algorithm”, International Journal of Peer to Peer Networks (IJP2P), Vol. 5, No 1, pp 1-8 .

[10] Y. Joung, (1998) “Asynchronous group mutual exclusion algorithm (extended abstract)”. Proc.in the 17t h annual ACM symposium on principles of distributed computing.

[11] Y. J. Joung, (2000), “Asynchronous group mutual exclusion”, Distributed Computing, Vol. 13, pp. 189-206.

[12] J. Beauquier, S. Cantarell, A. K. Datta, and F. Petit, (2003), “Group mutual exclusion in tree networks”, Journal of Information Science and Engineering, Vol. 19, pp. 415-432.

[13] O. Thiare, (2007), “Exclusion Mutuelle de Groupe dans les Systèmes distribués”, Thèse, Laboratoire Informatique de Cergy-Pontoise (LICP) EA 2175.

[14] V. Madenur and N. Mittal, (2008), "A Delay-Optimal Group Mutual Exclusion Algorithm for a Tree Network", Journal of Information Science and Engineering, Vol. 24, pp. 573-583.
[15] S. Dehghan and A.M. Rahmani, (2010), "A New Extended Group Mutual Exclusion Algorithm with Low Message Complexity in Distributed Systems", International Journal of Computer, Electrical, Automation, Control and Information Engineering Vol: 4, No: 4, pp 748-753.
[16] A. Swaroop and A. k. singh, (2010), "An Improved Quorum-Based Algorithm for Extended GME Problem in Distributed Systems", Journal of Information Science and Engineering Vol 26, pp 57-69.
[17] V. Hadzilacos, (2001) "A note on group mutual exclusion", in Proceedings of the 20th ACM Symposium on Principles of Distributed Computing, pp. 100-106.
[18] K. P. Wu and Y. J. Joung, (2000) "Asynchronous group mutual exclusion in ring networks", in IEE Proceedings of Computers and Digital Techniques, Vol. 147, pp. 1-8.
[19] S. Cantarell, A. K. Datta, F. Petit, and V. Villain, (2001) "Token based group mutual exclusion for asynchronous rings", in Proceedings of the IEEE International Conference on Distributed Computing Systems, pp. 691-694,.
[20] Y. J. Joung, (2002) "The congenial talking philosophers problem in computer networks" Distributed Computing, pp. 155-175.
[21] N. Mittal and P. K. Mohan, (2005) "An efficient distributed group mutual exclusion algorithm for non-uniform group access", in Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems, pp. 367-372.
[22] N. Mittal and Mohan (2007) "A priority-based distributed group mutual exclusion when group access is non-uniform", Journal of parallel and distributed computing, vol 67, no 7, pp 796-815.
[23] A. Swaroop, (2009) "Efficient Group Mutual Exclusion Protocols for Message Passing Distributed computing Systems", PhD thesis, National Institute of technology, Kurukshetra, Haryana, India, PIN-136119.