Hierarchical token based mutual exclusion algorithms

Marin Bertier, Luciana Arantes, Pierre Sens
Laboratoire d'Informatique de Paris 6, Université Paris 6, CNRS, INRIA
marin.bertier@lip6.fr, luciana.arantes@lip6.fr, pierre.sens@lip6.fr

Abstract

Mutual exclusion is a basic block of distributed synchronization algorithms. One of the challenge in highly distributed environments (like peer-to-peer or Grid configurations) is to provide scalable synchronizations taking into account the hierarchical network topology. This paper proposes hierarchical mutual exclusion algorithms. These algorithms are extensions of the Naimi-Trehel's token algorithm, reducing the cost of latency and the number of messages exchanges between far hosts. We propose three main extensions : (1) hierarchical proxy-based approach, (2) aggregation of requests, and (3) token preemption by closer hosts.

We compared the performance of these algorithms on an emulated Grid testbed. We study the impact of each of the extensions, showing that the combination of them can greatly improve performances of the original algorithm.

I. Introduction

Basic classical algorithms are commonly used by distributed applications. However, with the emergence of peer-to-peer and Grid computing, these applications spread over a larger number of nodes. Furthermore, in such environments latency gaps between hosts interconnects are very important. Therefore, distributed algorithms should be adapted to take into account those characteristics. A well-known example of such algorithms is mutual exclusion algorithm, which ensures exclusive access to a shared resource.

Many algorithms have been proposed to solve the problem of mutual exclusion in distributed systems. They can basically be divided into two groups: *permission-based* (Lamport [1], Ricart-Agrawala [2], Carvalho-Roucairol [3], Maekawa [4]) and *token-based* (Suzuki-Kazami [5], Raymond [6], Naimi-Trehel [7], Neilsen-Mizuno [8], Chang, Singhal and Liu [9]). The first group of algorithms are based on the principle of consensus between hosts, i.e., a host gets into a critical section only after having received permission from all other hosts. In the secong group of algorithms, a system-wide unique token is shared among hosts and the possession of it gives a host the exclusive right to enter into the critical section. Permission-based algorithms generally suffer from limited scalability. In contrast, token-based algorithms have an average low message cost and usually result in logarithmic message complexity O(log(N)) with regard to the number of hosts.

The majority of O(log(N)) token-based algorithms are tree-based, i.e., a logical tree structure expresses the different token requests and token propagation paths at a given time. Raymond'algorithm [6] organizes the hosts in a static logical tree structure. This tree remains unchanged, but the direction of its edges can change dynamically as the token propagates. Consequently, the directions of the edges always point to the possible token holder. Neilsen and Mizuno [8] extended this algorithm by passing the token directly to the requesting host instead of through intermediate hosts. Naimi-Trehel's algorithm [7] maintains a dynamic logical tree, such that the root of the tree is always the last host that will get the token among the current requesting ones. Chang Singhal and Liu [9] improved this algorithm, aiming at reducing the number of messages to find the last requesting host in the logical tree. Mueller [10] also proposed an extension to Naimi-Trehel's algorithm, introducing the concept of priority in it. A token request is associated with a priority and the algorithm first satisfies the requests with higher priority.

Although all those O(log(N)) token-based algorithms achieve better performance with respect to the average number of messages exchanged per critical section entry when compared to other mutual exclusion algorithms, they do not consider latency differences in hosts interconnects. We propose distributed token-based mutual exclusion algorithms, based on Naimi-Trehel's algorithm, which takes into account network topology, specially the latency gap between local and remote clusters of machines. Our algorithms reduce the numbers of inter-cluster messages and give a higher priority to local mutual exclusion requests. We have chosen to adapt Naimi-Trehel's algorithm because it uses a changeable logical tree structure to control mutual exclusion requests. This dynamic property of the tree is strongly exploited in our solution in order to tolerate higher latencies.

It is worth reminding that some authors [11] [12] have proposed mutual exclusion algorithms where nodes are, for some reason, gathered into groups. They basically propose hybrid approaches where the algorithm for intra-group requests is different from the inter-group one. However, they do not consider difference in network latency between hosts as a factor for grouping hosts. In [13], the authors propose to adapt the mutual exclusion mechanism of a DSM system to the latency hierarchy of an interconnection of clusters. Contrary to our proposal, their solution is based on a centralized token-based mutual exclusion protocol.

In the rest of the paper, we consider a general model where each host has a local memory and can send messages to any other. Communication between hosts is assumed to be perfect. Hosts are divided into clusters.

We distinguish local hosts belonging to the same cluster from remote hosts belonging to remote clusters. Furthermore, the words hosts and nodes are interchangeable.

Section II describes Naimi-Trehel's algorithm. In section III, we present our hierarchical versions of Naimi-Trehel's algorithm, which limit the propagation of requests between clusters. The three extensions to Naimi-Trehel's algorithm that we propose, per cluster proxy, aggregation and token preemption, are also described in this section. Section IV presents comparative performance evaluation of these algorithms, while the last section concludes our work.

II. Naimi-Trehel's algorithm

Naimi-Trehel's algorithm is a token-based algorithm, which maintains a logical dynamic tree structure such that the root of the tree is always the last node that will get the token among the current requesting nodes.

Each node i stores the following variables:

- The $owner$ variable, which represents the probable owner of the token.
- The $next$ variable, which represents the node that will receive the token when the critical section is released by i.
- The boolean $token$ variable, whose value is true if the process owns the token, or false otherwise.
- The boolean $requesting$ variable, whose value is true if the process requests the token, or false otherwise.

The identifier of i is represented by the variable *self*, while $Elected\_node$ identifies a unique node, among all nodes, that initially holds the token.

Figure 1 summarizes Naimi-Trehel's algorithm.

An example of Naimi-Trehel's algorithm execution with 4 nodes is shown in figure 2. Solid lines represent $owner$ links, while dashed ones represent $next$ links. The dark node keeps the token. Initially (a), A is the $Elected\_Node$ which holds the token. The $owner$ of all nodes points to A. In (b), B asks for the token, sending a request to its $owner$ (A), and becomes the new root ($owner_B = \emptyset$). Then, A updates its $next$ and $owner$ to point to B. In (c), C asks A for the token, then the request is forwarded to B which updates its $next$ to C. Both A and B update their $owner$ to C, since the latter is the last requester of the token (C becomes the root of the tree). When A will release the critical section, the token will be sent to B ($next$).

III. Hierarchical algorithms

Since inter-cluster latencies are higher than intra-cluster ones, the three extensions we proposed to Naimi Trehel's algorithm is based on the idea of limiting the propagation of requests between nodes of different clusters. To this end, we apply the following three extensions to Naimi-Trehel's algorithm:

- First, we introduce on each cluster, excepting the one that initially holds the token, a dedicated process, called proxy, which is in charge of storing the last request to remote clusters. Before asking for a token which it believes belong to a node of a remote cluster, a node i first sends a request to its corresponding proxy. If another node j of the same cluster has recently asked for the token and the proxy is aware of it, the proxy redirects the request to j avoiding transmission to the remote cluster. In fact, the proxy operates like a cache of remote requests. This hierarchical algorithm is presented in section III-A.
- The second extension aims at reducing the number of inter-cluster messages by aggregating remote requests. When a request has to be redirected to a probable owner, belonging to a remote cluster, the request is not sent to it but stored in a queue. This queue accumulates therefore requests for remote clusters. It is stored in the last node which will enter the critical section within the cluster. We name this node the *local_root*. We must remind that queuing of requests has been used by other Naimi-Trehel's-based algorithms as in [10]. However, in our case, it is applied only for remote cluster requests.
- Finally, we perform a local preemption of the token giving a higher priority to requests originating from the local cluster in order to exploit cluster locality. We define a threshold that defines the degree of locality and avoids starvation. When the number of local

Every node i:

Initialization
$resquesting \leftarrow false$
$next \leftarrow \emptyset$
$owner \leftarrow Elected\_node$
if $owner = self$ **then**
$token \leftarrow true$
$owner \leftarrow \emptyset$
else
$token \leftarrow false$

Request_CS
$requesting \leftarrow true$
if $owner \neq \emptyset$ **then**
{*The process hasn't the token, request for it:*}
Send $\langle Resquest, S_i \rangle$ to $owner$
$owner \leftarrow \emptyset$
Wait for receiving message $\langle Token \rangle$

Release_CS
$requesting \leftarrow false$
if $next \neq \emptyset$ **then**
Send $\langle Token \rangle$ to $next$
$token \leftarrow false$
$next \leftarrow \emptyset$

Receive_Request_CS(S_j)
{S_j *is the requesting process*}
if $owner = \emptyset$ **then**
{*Terminal node*}
if $resquesting = true$ **then**
{*The node asked for CS*}
$owner \leftarrow S_j$
if $next = \emptyset$ **then**
$next \leftarrow Sj$
else
{First request to the token since the last CS,}
{directly send the token to the requesting process}
$token \leftarrow false$
Send $\langle Token \rangle$ to S_j
else
{Non-ternimal node, following the resquest}
Send $\langle Request, S_j \rangle$ to $owner$
$owner \leftarrow S_j$

Receive_Token
{Receive the token from node k}
$token \leftarrow true$

Fig. 1. Naimi-Trehel's algorithm

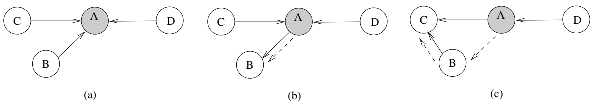


Fig. 2. Sample execution of Naimi-Trehel's algorithm

request is below this threshold, the requesting path is modified in order to serve local requests first. These last two extensions are presented in section III-B.

A. Proxy-based algorithm

We modify Naimi-Trehel's algorithm presented in section II as follows:

The *LocalCluster* variable, added on each node i, identifies the cluster to which node i belongs.

On each cluster Ci, excepting the one that has the $Elected\_node$, a node is elected among Ci's nodes to have a specific initialization role. This node is called the $Proxy_i$.

Initially, the $owner$ variable of $Proxy_i$ points to the $Elected\_Node$ as well as the $owner$ variable of the nodes that belong to $Elected\_node$'s cluster. On the other hand, the $owner$ variable of the other nodes points to the $Proxy_i$ of their respective cluster.

Figure 3 summarizes our hierarchical proxy-based version of Naimi-Trehel's algorithm.

Every proxy $Proxy_i$:

Initialization
$owner \leftarrow Elected\_node$

Every node i:

Initialization
$resquesting \leftarrow false$
$next \leftarrow \emptyset$
if $Elected\_node \in LocalCluster$ **then**
$owner \leftarrow Elected\_node$
if $owner = self$ **then**
$token \leftarrow true$
$owner \leftarrow \emptyset$
else
$token \leftarrow false$
else
$owner \leftarrow Proxy_i$
$token \leftarrow false$

Request_CS
{*Unchanged* }

Release_CS
{*Unchanged* }

Receive_Request_CS(S_j)
{*Unchanged* }

Receive_Token
{*Unchanged* }

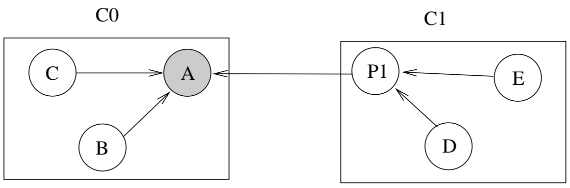
Fig. 3. Proxy-based algorithm

Figure 4(a) presents an example of an initial configuration with two clusters, $C0$ and $C1$, where nodes A, B, and

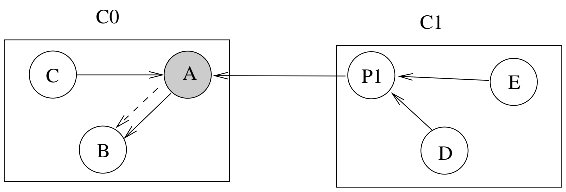
C belong to cluster $C0$, and nodes D, E and $P1$ belong to $C1$. $P1$ is the $Proxy$ of $C1$. Initially, A has been elected to have the token (the $Elected\_Node$). We consider that A is in the critical section (CS).

In 4(b), B asks A for the token. A and B belong to the same cluster. Since A is in the CS, A sets both its $owner$ and $next$ variable to B.

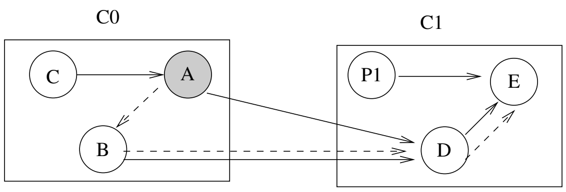
In 4(c), D, which does not belong to the same cluster of the token holder, asks for the token. It sends then a request to its proxy $P1$, which redirects the request to A. $P1$ sets its $owner$ to D. When arriving at A, the request is forwarded to the root (B) which simply updates its $next$ and $owner$ variables to D. E asks then for the token. The proxy $P1$ locally redirects the request to D which updates its $next$ and $owner$ variables to E. At the end of each critical section execution, the token will follow the path pointed by $next$ variables. This scenario shows the advantage of the algorithm since E request was not forwarded to the remote cluster C_0, as it would be the case in the original Naimi-Trehel's algorithm.



(a) Initial configuration A holds the Token



(b) B asks for CS



(c)D and E ask for CS

Fig. 4. Hierarchical proxy-based algorithm execution scenarios

B. Aggregation and preemption algorithms

We have modified the proxy-based algorithm of the previous section III-A to reduce even more the number of inter-cluster messages. This improvement is based on aggregation of messages and preemption of the token by local nodes.

On each node i, we added the following variables:

- The *R_Queue* variable, which is a queue of requests issued from remote clusters.
- The *nb_preempt* variable, which represents the number of local requests that have preempted requests issued from remote clusters.

Figures 5 and 6 describe the algorithm.

Similar to the original Naimi-Trehel's algorithm, a request for entering a critical section (CS) follows the $owner$'s path until it reaches its $local\_root$ (the node of the same cluster whose $owner$ variable is set to $\emptyset$, i.e., the last node of the cluster to have requested the critical section).

When a node receives a request, if it is not a $local\_root$ node ($owner \neq \emptyset$), it forwards the request and updates its $owner$ (only if the request is issued from the local cluster in order to avoid redirection to remote clusters). If the receiver is a $local\_root$ node ($owner = \emptyset$) which waits for the token ($requesting = true$), we distinguish two cases: (1) The received request is the first one since the node waits for the token (i.e. $next = \emptyset$). Then, the $next$ is set to the requester because after the node obtains and releases the critical section, it will have to send the token to the requester. The $owner$ is also updated *only if* the request came from the local cluster. (2) The $next$ is already set. Since the receiver is a $local\_root$, the $next$ inevitably points to a remote node. In this case, if the requester is local, and the number of preemptions is below the threshold, we perform a local preemption of the token by setting $next$ to the requester and memorizing the old $next$ in the beginning of R_Queue. Each time a node becomes the new $local\_root$, the R_Queue is sent to it. The R_Queue is also included in the token message.

Figures 7 and 8 show some samples of the algorithm's execution. We consider the same configuration presented in figure 4(b) where B asks A for the token. A third cluster $C2$, with nodes F, G and $P2$ ($Proxi_2$) is included in the figure. $Threshold$ of preemption is equal to 2.

In figure 7(a), the node D asked for the token. It sends a request to its proxy $P1$, which redirects it to A, setting its own $owner$ to A. On A the request is propagated to the $local\_root$ (B), which simply updates its $next$ (the $owner$ of A and B are not updated since the requester is a remote node). Then, F belonging to cluster C_2 asks A for the token. The request is forwarded to the $local\_root$ (B).

Every proxy $Proxy_i$:

Initialization
$owner \leftarrow Elected\_node$

Every node i:

Initialization
$requesting \leftarrow false$
$next \leftarrow \emptyset$
$R\_Queue \leftarrow \emptyset$
if $Elected\_node \in LocalCluster$ **then**
$owner \leftarrow Elected\_node$
if $owner = self$ **then**
$token \leftarrow true$
$owner \leftarrow \emptyset$
else
$token \leftarrow false$
else
$owner \leftarrow Proxy_i$
$token \leftarrow false$

Request_CS
$requesting \leftarrow true$
if $owner \neq \emptyset$ **then**
{*The process hasn't the token, request for it:*}
Send $\langle Request, S_i \rangle$ to $owner$
$owner \leftarrow \emptyset$
$R\_Queue \leftarrow \emptyset$
Wait for receiving message $\langle Token \rangle$

Release_CS
$requesting \leftarrow false$
if $next \neq \emptyset$ **then**
if $next \notin LocalCluster$ **then**
{*The token will be sent to a remote*}
$nb\_preempt \leftarrow 0$
if $R\_Queue \neq \emptyset$ **then**
$owner \leftarrow Queue(R\_Queue)$
else
$owner \leftarrow next$
Send $\langle Token, R\_Queue, nb\_preempt \rangle$ to $next$
$token \leftarrow false$
$next \leftarrow \emptyset$

Fig. 5. Aggregation and preemption algorithm: request and release critical section

Since B's $next$ is already set ($next = D$), the requester (F) is inserted in the R_Queue.

In figure 7 (b), node E of $C1$ asks for the token. The proxy $P1$ *locally* redirects the request to D (the *local root* of C_1) which updates its $next$ and *owner* to E. At the same time, C reclaims the token. The request is redirected to B. Since B's $next$ points to a node of a remote cluster and the number of local preemption is below the threshold, the $next$ and *owner* paths are changed: the $next$ and *owner* of B are updated to designated the local node C. The old value of B's $next$ (D) is added at the beginning of R_Queue (it is not shown in the figure). C is the new

Every node i:

Receive_Request_CS(S_j)
{*S_j is the requesting process*}
if $owner = \emptyset$ **then**
{*Terminal node*}
if $resquesting = true$ **then**
{*The node asked for CS*}
if $next = \emptyset$ **then**
$next \leftarrow Sj$
if $S_j \in LocalCluster$ **then**
$owner \leftarrow S_j$
else
if $S_j \in LocalCluster$ **and**
$nb\_preempt < Threshold$ **then**
{Local preemption of the token by the sender}
$nb\_preempt \leftarrow nb\_preempt + 1$
$R\_Queue \leftarrow next + R\_Queue$
$next \leftarrow S_j$
$owner \leftarrow S_j$
Send $\langle Queue, R\_Queue, nb\_preempt \rangle$ to owner
$R\_Queue \leftarrow \emptyset$
else
{Add the sender to the end of R_Queue}
$R\_Queue \leftarrow R\_Queue + S_j$
else
{First request to the token since the last CS,}
{directly send the token to the requesting process}
$token \leftarrow false$
Send $\langle Token, R\_Queue, nb\_preempt \rangle$ to S_j
else
{Non-ternimal node, following the resquest}
Send $\langle Request, S_j \rangle$ to $owner$
if $S_j \in LocalCluster$ **then**
$owner \leftarrow S_j$

Receive_Token($R\_Queue_k, nb\_preempt_k$)
{Receive the token from node k}
$token \leftarrow true$
$R\_Queue \leftarrow R\_Queue_k + R\_Queue$
if $k \in LocalCluster$ **then**
$nb\_preempt \leftarrow nb\_preempt_k$
if $next = \emptyset$ **then**
$next \leftarrow Head(Q)$
$R\_Queue \leftarrow R\_Queue - Head(Q)$

Receive_Queue(Q, nb)
$nb\_preempt \leftarrow nb$
if $next = \emptyset$ **then**
$next \leftarrow Head(Q)$
$R\_Queue \leftarrow R\_Queue - Head(Q)$

Fig. 6. Aggregation and preemption algorithm: message handlers

$local\_root$, receiving then the R_Queue. When C receives the R_Queue, it removes the head value of it ($= D$) and sets its $next$ to this value.

In figure 8(a), A releases the token. It sends it to its $next$ node (B). B releases then the CS and sends the token to C. A and B set their $next$ to $\emptyset$. In (b), C ends the execution of the CS and sends the token with the R_Queue to the remote node D. It also sets it *owner* to F. At the end of its section, D will send the token to E according to its $next$. In (c), when D ends executing the CS, it sends the token to E and the R_Queue. When receiving the message, E updates its next variable to F.

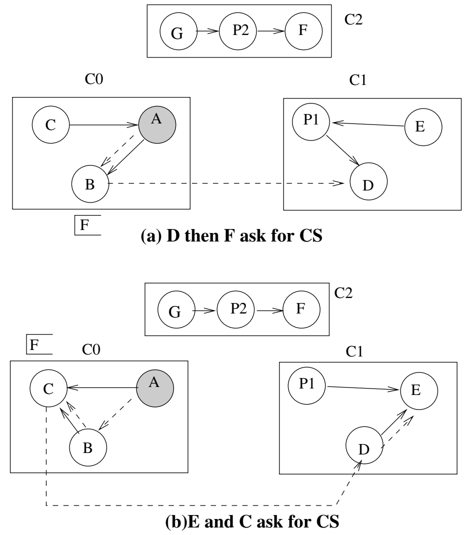


Fig. 7. Aggregation and preemption algorithm execution: Inter-clusters requests

IV. Performance evaluation

This section presents the performance evaluation of several experiments which compare the efficiency of mutual exclusion algorithms. We compare the six following algorithms:

- $Centralized$ token-based algorithm. In this classical algorithm, a unique host, the $lock\_manager$, manages all token requests and granting messages. When a host wants to enter into the CS, it sends a request to the $lock\_manager$; when a host ends the CS it gives back the token to the $lock\_manager$, which forwards it to the next requesting host.
- $Broadcast$ algorithm, which implements the permission-based Ricart-Agrawala algorithm [2]. A node wishing to enter into the CS sends a request to every other node and waits for their permission. When a node receives a request, it sends its permission to the requesting node if either it is not requesting itself the token or another node's request precedes its own.
- $NaimiTrehel$ algorithm, which implements the Naimi-Trehel's token-based algorithm presented in section II.
- $Proxy$ algorithm, which implements the algorithm presented in section III-A.
- $PreemptAggregation$ algorithm, which implements the algorithm presented in section III-B, which provides token preemption and message aggregation approaches.
- $Preempt$ algorithm, which just disables the aggregation mechanism of $PreemptAggregation$ algorithm.

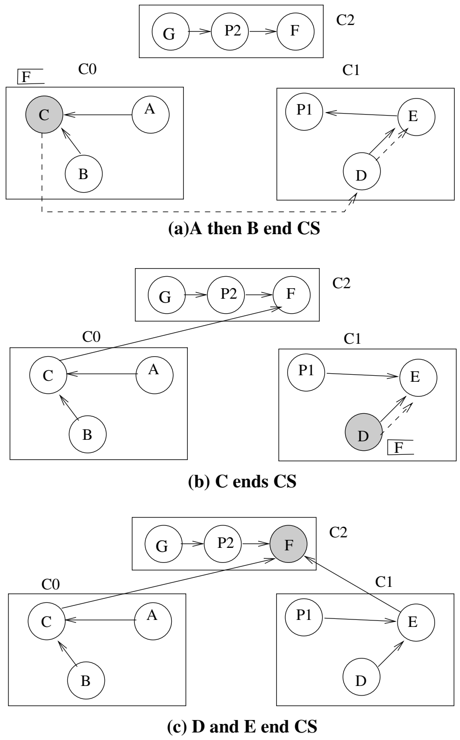


Fig. 8. Aggregation and preemption algorithm execution: Token transmission

To emulate a Grid environment with multilevel network latencies, we have used a specific distributed test platform, that allows injection of network delays. We establish a virtual router by using DUMMYNET [14] and IPNAT. We use IPNAT, an IP masquering application, to divide our network into virtual LANs. DUMMYNET is a flexible tool originally designed for testing network protocols. It simulates bandwidth limitations, delays, packet losses. In practice, it intercepts packets, selected by address and port of destination and source, and passes them through one or more objects called queues and pipes which simulate the network effects. In our experiment, each message exchanged between two different LANs passes through this specific host.

A. Evaluation experiment configuration

The experiment described in this section was performed on a non dedicated cluster of nine PCs. We consider a heterogeneous network composed of two Pentium III 600 MHz and six Pentium IV 2 GHz linked by a 100 Mbits/s Ethernet. The algorithms were implemented in Java (Sun's JDK 1.4) on top of a Linux 2.4 kernel.

PCs are spread in 3 clusters of 3 hosts. The topology is preliminary known by every system member as well as the initial owner of the token.

For these experiments, we introduce a delay of 300 ms for inter-cluster communication. Every involved site produces 20 mutual exclusion requests.

These requests are characterized by :

- α is the time taken by a node to execute the critical section,
- β specifies the mean time between releasing the critical section and requesting it again,
- ϵ is the preemption threshold only for $Preempt$ and $PreemptAggregation$ algorithms.

The performance measures include :

- the **number of exchanged messages**, divided in two categories : messages exchanged between two hosts in the same cluster (local messages) and messages between two hosts of different clusters (global messages).
- **obtaining time** is the time for an host between the moment when it requests the critical section and the moment when it get into it.

B. Results and Discussion

The aim of these experiments is to observe evolution of the behavior of each algorithm when the relation between α and β varies and ϵ increases. Figures 9 and 10 show the obtaining time as a function of α, β and ϵ. Figure 9 compares our $PreemptAggregation$ algorithm with classical algorithms : $NaimiTrehel$, $Centralized$ and $Broadcast$ algorithms. While figure 10 compares the same algorithm $PreemptAggregation$ with our other algorithms : $Proxy$ and $Preempt$. The figure 11 compares the number of messages exchanged between hosts during an experiment where $\alpha = \beta = 500\ ms$ and table I summarizes the same experiment. To represent in the same figure the number of messages exchanged in the broadcast experiment, this number of messages is represent by a scale four times greater than the scale of the other algorithms.

For all algorithms when the ratio α/β decreases the obtaining time decreases too, simply because statistically when an host requests the token, there are less requesting hosts and they stay less time in the CS. The first remark about these experiments is that for each algorithm which uses the preemption, the obtaining time decreases when the preemption threshold increases. We can easily explain this result by the fact that local communication is cheaper than global communication. As shown in figure 11, the number of local messages increases when the number of preemption increases while the number of global message decreases. This also explains why all the algorithms presented in figure 10 are more efficient than the classical algorithms presented in figure 9.

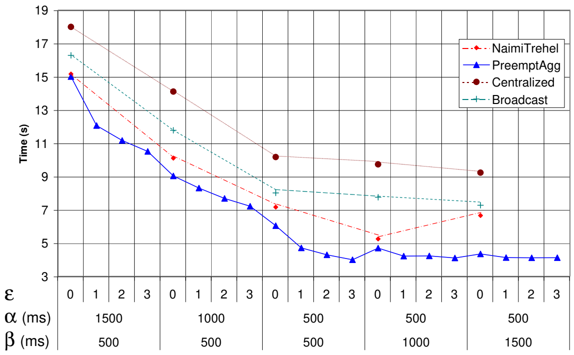


Fig. 9. Comparison algorithm : obtaining time

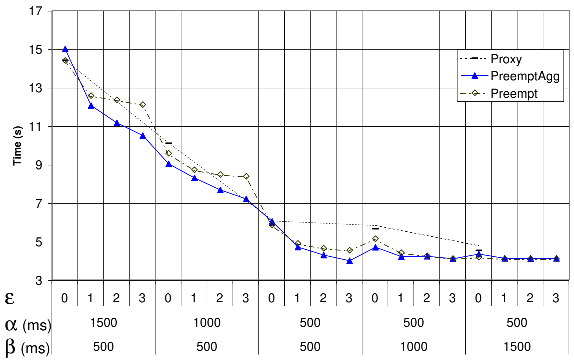


Fig. 10. Comparison algorithm : obtaining time

The $PreemptAggregation$ algorithm is the most efficient algorithm presented here. The $Aggregation$ mechanism reduces the number of messages exchanged in the $Preempt$ algorithm. In the $Proxy$ algorithm, the fact that initial inter-Lan requests are intercepted by the local cluster leader allows to decrease the number of global messages.

One avantage of the $PreemptAggregation$ algorithm is that global requests are not transmited to the host which has requested the critical section. Therefore, as long as the token has not arrived in a cluster, no hosts in the cluster know that a remote host has requested the token. Consequently, the number of $preemption$ is not limited

Type	ϵ	Obtaning time (s)		Average	Total	Nb of messages		
		average	standard deviation	path length	path length	local	global	%
Naimi-Trehel		7.157	1.351	2.37	419	148	393	0.39
Proxy		5.858	1.016	2.35	414	237	299	0.84
PreemptAgg	0	6.037	1.186	4.14	731	261	185	1.53
	1	4.704	1.757	4.56	806	359	140	2.56
	2	4.287	2.269	4.53	804	395	120	3.34
	3	3.992	2.573	4.68	840	395	120	3.34
Preempt	0	5.858	0.999	2.75	486	402	196	2.11
	1	4.897	2.053	2.69	473	466	124	3.76
	2	4.640	2.517	2.66	470	477	109	4.37
	3	4.540	2.884	2.62	463	482	79	6.12
Centralized		10.161	1.981			119	353	0.34
Broadcast		8.007	1.010			720	2159	0.33

TABLE I. Summary of experiement with α and β equals to $500\ ms$

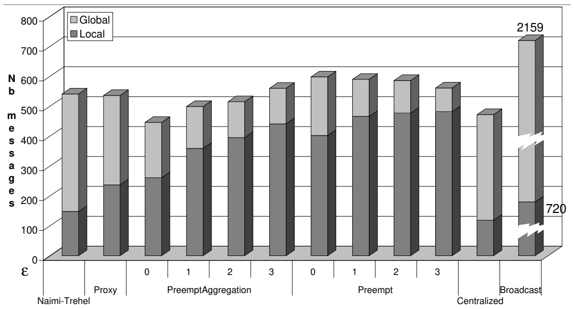


Fig. 11. Comparison algorithm : Number of messages exchanged

by ϵ but by $n + \epsilon$ where n is the number of hosts in the cluster. The locality mechanism is more exploited but the absence of starvation is preserved.

V. Conclusion

We have presented in this paper a new approach to optimize mutual exclusion algorithms in a GRID environement. The main idea is to adapt the algorithm according to network topology in order to confine most communications to intra-cluster. This improvement allows to optimize the obtaining time of the token at the expense of fairness.

This behavior is particulary shown with the $PreemptAggregation$ algorithm presented in section III-B. As we have seen in the performance evaluations, this algorithm allows to decrease significally the obtaining time with respect to the other algorithms presented but specially with $NaimiTrehel$ or $Centralized$ algorithms.

References

[1] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–564, July 1978.

[2] G. Ricart and A. Agrawala, "An optimal algorithm for mutual exclusion in computer networks," *CACM: Communications of the ACM*, vol. 24, 1981.

[3] O. S. F. Carvalho and G. Roucairol, "On mutual exclusion in computer networks," *Communications of the ACM*, vol. 26, no. 2, pp. 146–147, 1983.

[4] M. Maekawa, "A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems," *ACM Transactions on Computer Systems*, vol. 3, no. 2, pp. 145–159, May 1985.

[5] I. Suzuki and T. Kasami, "A distributed mutual exclusion algorithm," *ACM Transactions on Computer Systems (TOCS)*, vol. 3, no. 4, pp. 344–349, 1985.

[6] K. Raymond, "A tree-based algorithm for distributed mutual exclusion," *ACM Transactions on Computer Systems (TOCS)*, vol. 7, no. 1, pp. 61–77, 1989.

[7] M. Naimi, M. Trehel, and A. Arnold, "A log (N) distributed mutual exclusion algorithm based on path reversal," *Journal of Parallel and Distributed Computing*, vol. 34, no. 1, pp. 1–13, 10 Apr. 1996.

[8] M. L. Neilsen and M. Mizuno, "A dag-based algorithm for distributed mutual exclusion," in *Proceedings of the 11th International Conference on Distributed Computing Systems (ICDCS)*, Washington, DC, 1991, pp. 354–360, IEEE Computer Society.

[9] I. Chang, M. Singhal, and M. T. Liu, "An improved log (N) mutual exclusion algorithm for distributed systems," in *Proceedings of the 1990 International Conference on Parallel Processing*, Aug. 1990, pp. 295–302.

[10] F. Mueller, "Prioritized token-based mutual exclusion for distributed systems," in *Proceedings of 12th Intern. Parallel Proc. Symposium & 9th Symp. on Parallel and Distr. Processing*, Mar. 1998, pp. 791–795.

[11] I. Chang, M. Singhal, and M. T. Liu, "A hybrid approach to mutual exclusion for distributed system," in *Proceedings of the 14th IEEE Annual International Computer Software and Applications Conference*, 1990, pp. 289–294.

[12] A. Housni and M. Trehel, "Distributed mutual exclusion by groups based on token and permission," in *Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications*, June 2001, pp. 26–29.

[13] L. Bouge G. Antoniu and S. Lacour, "Making a DSM consistency protocol hierarchy-aware: an efficient synchronization scheme," in *Proceedings of the Workshop on Distributed Shared Memory on Clusters*, 2003, pp. 516–521.

[14] L. Rizzo, "Dummynet: a simple approach to the evaluation of network protocols," *ACM Computer Communication Review*, vol. 27, no. 1, pp. 31–41, 1997.