Distributed Mutual Exclusion Algorithms for Grid Applications: a Hierarchical Approach

Marin Bertier, Luciana Arantes and Pierre Sens.
LIP6 - Université de Paris 6 - INRIA Rocquencourt
4, Place Jussieu 75252 Paris Cedex 05, France.
Phone: (33).1.44.27.34.23 - Fax: (33).1.44.27.74.95
email: [marin.bertier, luciana.arantes, pierre.sens]@lip6.fr

Abstract

The majority of current distributed mutual exclusion algorithms are not suited for parallel or distributed applications on a Grid as they do not consider the heterogeneity of latency on Grids. We propose two distributed mutual exclusion algorithms, based on Naimi-Trehel’s token-based algorithm, which take into account latency gaps, especially those between local and remote clusters of machines. Our first algorithm exploits cluster locality by giving higher priority to critical section requests issued from nodes of the same cluster when compared to those from remote nodes. Our second algorithm adds a router layer to the first algorithm, bringing it closer to Grid network topology. Viewing each cluster as a single node, the Naimi-Trehel algorithm is applied to this router layer. Redirection of inter-cluster messages to cluster’s nodes is then minimized.

Keywords: distributed mutual exclusion algorithms, token-based algorithm, Grid, latency heterogeneity, cluster locality.

1 Introduction

Distributed and parallel applications benefit from Grid infrastructure, which enables the sharing of a wide variety of geographically distributed resources acting as a single powerful computer. However, many of these applications may require that their processes obtain exclusive access to one or more of these shared resources. Therefore, the principle of mutual exclusion is important in Grid computing.

Several distributed algorithms have been proposed to solve the problem of mutual exclusion in distributed systems, serializing concurrent accesses to a shared resource. They can basically be divided into two groups: permission-based (e.g., Lamport [6], Ricart-Agrawala [12], Singhal [14], Maekawa [7]) and token-based (Suzuki-Kazami [15], Raymond [11], Naimi-Trehel [9], Neilsen-Mizuno [10], Chang, Singhal and Liu [4]). The first group of algorithms are based on the principle that a node gets into critical section only after having received permission from all other nodes (or the majority of them [7]). In the second group of algorithms, a system-wide unique token is shared among all nodes, and the possession of it gives a node the exclusive right to enter into critical section. The latter usually have an average lower message cost and many of them result in logarithmic message complexity O(log N) with regard to the number of nodes. The majority of O(log N) token-based algorithms are tree-based i.e., a logical tree structure expresses the different paths of token requests and its propagation at a given time.

Since in a Grid environment the number of nodes can be very large, scalability of a distributed mutual exclusion algorithm is an important feature. Considering that tree-based token mutual exclusion algorithms scale quite well, they would seem to be adequate for Grid applications. However, these algorithms do not
take into account the communication latency heterogeneity of a Grid environment. For instance, latency between machines in different clusters can be much higher than the latency between nodes within a single cluster. Consequently, the performance of mutual exclusion algorithms can be critical for Grid applications.

We propose in this article two distributed token-based mutual exclusion algorithms which take into account the hierarchical network topology of Grids. Our work particularly considers the communication latency gap between local and remote clusters of machines. Our algorithms reduce the number of inter-cluster messages, giving higher priority to local mutual exclusion requests. Both of them are based on Naimi-Trehel’s $O(\log N)$ token-based algorithm [9]. This algorithm maintains a dynamic logical tree, such that the root of the tree is always the last site that will get the token among the current requesting ones. Our choice of Naimi-Trehel’s algorithm can be justified by its dynamic property, which is strongly exploited in our solution for tolerating higher latency.

The first algorithm adapts Naimi-Trehel’s algorithm by prioritizing critical section (CS) requests issued from nodes of the same cluster over those from remote nodes. The second algorithm considers that messages exchanged by nodes of different clusters always pass along routers (proxys). A proxy layer is added to the first algorithm, bringing it closer to Grid network topology. Naimi-Trehel’s algorithm is then applied to this proxy layer. Using this two-layer Naimi-Trehel approach, some inter-cluster messages can be managed at the proxies’ level, without being necessary to be redirected to the other nodes inside clusters.

In the rest of this paper, we consider a general distributed model where no common shared memory is available. Nodes have local memory, communicating by message passing. There is one process per node and only a single shared resource. We also assume a fully connected network where message delivery is guaranteed and message transfer delays are finite. Our Grid infrastructure is composed of nodes grouped into clusters. We distinguish local nodes belonging to the same cluster from remote nodes belonging to remote clusters. The words node and site are interchangeable as well as router and proxy.

The organization of this paper is as follows. Section 2 presents Naimi-Trehel’s algorithm. Our hierarchical versions of Naimi-Trehel’s algorithm limiting the propagation of requests between clusters are described in section 3. Some related work is given in section 4. Comparative performance evaluation of the algorithms are discussed in section 5, while the last section concludes our work.

2 Naimi-Trehel algorithm

Naimi-Trehel’s algorithm [9] is a token-based algorithm, where nodes are logically arranged, by their requests, as a rooted tree. In other words, it maintains a logical dynamic tree structure such that the root of the tree is always the last node that will get the token among current requesting nodes. A second structure of the algorithm is a distributed queue which keeps nodes’ Critical Section (CS) pending requests.

Naimi-Trehel’s algorithm is described in Algorithm 1.

Each site $S_i$ has the following local variables:

- `self`: keeps the identification $S_i$ of the node
- `owner`: stores the probable owner of the token.
- `next`: indicates the node that will receive the token when the critical section is released by $S_i$.
- `token`: boolean variable, whose value is true if the node owns the token, and false otherwise.
- `requesting`: boolean variable, whose value is true if the node has requested the token, and false otherwise.

The constant $Elected\ node$ identifies the node of the system that initially holds the token.

Messages are sent through the function $Send(\ Type,...)$ where $\ Type$ specifies the type of message. The other parameters of the function vary based on the type of message. Two types of messages have been defined:
- **Request**: sent by a node which does not have the token, but wants to enter into the critical section. The identification $S_j$ of the requesting node is included in the message. The function $\text{Receive}_\text{Request}_\text{CS}(S_j)$ is called upon reception of this type of message from $S_j$.

- **Token**: represents the transmission of the token. The function $\text{Receive}_\text{Token}(S_j)$ is called upon reception of this type of message from $S_j$.

---

**Algorithm 1 Naimi-Trehel**

*Every node $S_i$:*

**Initialization:**

- $\text{requesting} \leftarrow \text{false}$
- $\text{next} \leftarrow \emptyset$
- if $\text{self} = \text{Elected}_\text{Node}$ then
  - $\text{token} \leftarrow \text{true}$
  - $\text{owner} \leftarrow \emptyset$
- else
  - $\text{token} \leftarrow \text{false}$
  - $\text{owner} \leftarrow \text{Elected}_\text{Node}$

**Request_Co**:

- $\text{requesting} \leftarrow \text{true}$
- if $\text{owner} \neq \emptyset$ then
  - { *The site hasn’t the token, it should request it* }
  - Send $(\text{Request}, S_i)$ to owner
  - $\text{owner} \leftarrow \emptyset$
  - Wait for receiving message $(\text{Token})$

**Release_Co**:

- $\text{requesting} \leftarrow \text{false}$
- if $\text{next} \neq \emptyset$ then
  - Send $(\text{Token})$ to $\text{next} $
  - $\text{token} \leftarrow \text{false}$
  - $\text{next} \leftarrow \emptyset$

Initially, the root is the token holder. When requests are issued, they are guided through a chain of *owners* to the current root. Each node on the propagation path sets its probable owner to the requester, i.e. the tree is modified dynamically. At the end of the critical section, the token follows the *next* links.

An example of Naimi-Trehel algorithm execution with 4 nodes is shown in Figure 1. Solid lines represent *owner* links, while dashed ones represent *next* links. The shaded node holds the token. Initially (a), node $A$ is the *Elected.Node* which holds the token. The *owner* of all nodes points to $A$. In (b), node $B$ asks for the token by sending a request to its *owner* ($\text{owner}_B = A$). $B$ becomes the new root ($\text{owner}_B = \emptyset$). Then, $A$ updates its *next* and *owner* to point to $B$. In (c), $C$ asks $A$ for the token. The request is forwarded to $B$ which updates its *next* to $C$ ($\text{next}_B = C$). Both $A$ and $B$ update their *owner* to $C$, since the latter is the last requester of the token ($C$ becomes the new root of the tree). When $A$ releases the critical section, the token will be sent to $B$ since $\text{next}_A = B$.

![Figure 1: Example of Naimi-Trehel’s algorithm execution](image-url)
3 Hierarchical algorithms

Since inter-cluster latency is higher than intra-cluster latency, the adaptation of Naimi Trehel algorithm to a cluster-based Grid platform focuses on limiting the propagation of requests between nodes of different clusters. We first propose a one-level clustered algorithm which exploits cluster locality by giving higher priority to intra-cluster requests as compared to remote ones. We have also modified the initializing phase of the original Naimi Trehel algorithm in order to gather more information about CS requests issued from nodes of the same cluster. We propose a two-level clustered algorithm, assuming that all messages between nodes of different clusters are sent through routers (proxies). This approach is closer to the topology of Grid platforms and has the advantage of allowing for some of the messages to be managed at the routers’ level without being forwarded to the other nodes of the clusters.

3.1 One-level clustered algorithm

Whenever possible, intra-cluster CS requests are satisfied before remote ones. To avoid starvation, a threshold value limits the maximum number of CS that can be successively executed by nodes of the same cluster. The algorithm then keeps track of the number of current successive CS executions within a single cluster. This value, called the \textit{number of preemption}, is passed in messages. Thus, whenever the number of successive local requests is below that threshold, the distributed queue of CS requests pointed to by \textit{next} links is modified for serving local requests before remote ones.

\textbf{Note:} The last node which will enter the critical section within a cluster is named \textit{Local\_root}. It is identified by the fact that its \textit{owner} variable is equal to $\emptyset$.

\begin{algorithm}[H]
\caption{Hierarchical Algorithm - Initialization, Request and Release functions}
Every node $S_i$:

\textbf{Initialization:}
\begin{verbatim}
requesting ← false
next ← \emptyset
remote\_owner ← \emptyset
if Elected\_node ∈ LocalCluster, then
  if Elected\_node = self then
    token ← true
    owner ← \emptyset
  else
    token ← false
    owner ← Elected\_node
else
  token ← false
  if self = Proxy, then
    owner ← Elected\_node
  else
    owner ← Proxy
\end{verbatim}

\textbf{Request\_CS:}
\begin{verbatim}
if owner \neq \emptyset then
  \{ The node hasn't the token, it requests it \}
  Send (Request, $S_i$) to owner
  owner ← \emptyset
  Wait for receiving message (Token)
\end{verbatim}

\textbf{Release\_CS:}
\begin{verbatim}
if next \neq \emptyset then
  if next \notin LocalCluster then
    \{ The token will be sent to a remote node \}
    \textit{nb\_preempt} ← 0
  if owner = \emptyset then
    owner ← remote\_owner
    remote\_owner ← \emptyset
  Send (Token, nb\_preempt) to next
  token ← false
  next ← \emptyset
\end{verbatim}

We consider the same local variables described in section 2 for Naimi-Trehel algorithm, adding the following new ones:

- \textit{LocalCluster\_i}: identifies the cluster to which node $S_i$ belongs. All nodes are aware of it.
- \textit{remote\_owner}: the \textit{Local\_root} node updates this variable whenever it receives a first CS request from a remote node. Its \textit{owner} variable will only be updated with the \textit{remote\_owner}’s value when the token is sent to a remote node or the number of preemption is greater than the threshold value.
Algorithm 3 Hierarchical Algorithm (cont.) - Receive function

Every node $S_i$:

Receive Request CS($S_j$):

- $S_j$ is the requesting process
- if owner = $\emptyset$ then
- $S_i$ is the local root
- if requesting = true then
- The node asked for CS
- if next = $\emptyset$ then
- $S_j$
- if $S_j \in \text{LocalCluster}$, then
- owner $\leftarrow S_j$
- else
- remote owner $\leftarrow S_j$
- else
- if $S_j \in \text{LocalCluster}$
- and nb_preempt < Threshold then
- Local preemption of the token by the sender
- nb_preempt $\leftarrow$ nb_preempt + 1
- owner $\leftarrow S_j$
- Send (Preempt, nb_preempt, next) to $S_j$
- next $\leftarrow S_j$
- else
- Send (Request, $S_j$) to next
- owner $\leftarrow S_j$
- else
- token $\leftarrow$ false
- Send (Token, nb_preempt) to $S_j$
- owner $\leftarrow S_j$
- else
- Send (Request, $S_j$) to owner
- if $S_j \in \text{LocalCluster}$, then
- owner $\leftarrow S_j$

Receive Token ($S_j$):

- Receive the token from node $S_j$
- token $\leftarrow$ true

Receive Preempt (nb_preempt, remote_node):

- nb_preempt $\leftarrow$ nb_preempt + nb_preempt
- if next = $\emptyset$ then
- $S_i$ is the local root
- next $\leftarrow$ remote node
- remote owner $\leftarrow$ remote node
- else
- Send (Preempt, nb_preempt, remote_node) to owner
- \textit{nb\_preempt}: counter that keeps track of the number of successive local CS requests. It is increased by 1 when site \( S_i \) receives a request from a local node and it is updated upon receiving a \textit{Preempt} message (see below).

A third type of message, \textit{Preempt}, has been defined. This message passes around the value of \( \text{nb\_preempt} \) of the sender node as well as the identification of the node that should be preempted. The function \textit{Receive\_Preempt} is called upon reception of this message.

The type \textit{Token} of message has also been modified for including the value of the local variable \( \text{nb\_preempt} \) of the site which grants the token.

Algorithms 2 and 3 summarize our one-level clustered version of Naimi-Trehel algorithm.

### 3.1.1 Initialization Phase

The initialization phase of the algorithm maps the path of \textit{owners} to the multi-cluster topology of the network. Every cluster \( C_i \), except the \textit{Elected\_Node}'s cluster, designates a local node to which all the other nodes in the cluster should point to. This node is called \textit{Proxy} \( i \). It is worth remarking that while we named this node as \textit{Proxy} \( i \), this first algorithm is not a proxy-based (router-based) one. We simply use this terminology to simplify the description of both algorithms that we present in this article.

Initially, the \textit{owner} variable of a \textit{Proxy} \( i \) as well as the nodes that belong to \textit{Elected\_Node}'s cluster point to the \textit{Elected\_Node}. However, the \textit{owner} variable of the other nodes \( S_i \) points to the \textit{Proxy} \( i \) of their respective cluster \( C_i \).

Figure 2 shows an example of such initialization. Figure 2(a) presents two clusters, \( C_0 \) and \( C_1 \), where nodes \( A, B, \) and \( C \) belong to cluster \( C_0 \), and nodes \( D, E, \) and \( F \) belong to \( C_1 \). \( F \) is the \textit{Proxy} \( 1 \) of \( C_1 \). Since \( A \) has been elected to have the token, it is the \textit{Elected\_Node}. \( A \) is in the critical section (CS). In Figure 2(b), \( D \), which does not belong to the same cluster as the token holder, asks for the token, sending a request to \( F \), which redirects the request to \( A \). \( F \) then sets its \textit{owner} to \( D \). When the request arrives at \( A \), the latter updates its \textit{next} and \textit{remote\_owner} variables to \( D \). \( E \) asks then for the token. Node \( F \), the \textit{Proxy} \( 1 \), locally redirects the request to \( D \) which updates its \textit{next} and \textit{owner} variables to \( E \). \( F \) also updates its \textit{owner} link to \( E \). This scenario shows the advantage of our modification in the initialization phase: \( E \)'s request was not forwarded to the remote cluster \( C_0 \), as it would be in the case of the original Naimi-Trehel's algorithm.

![Figure 2: Hierarchical execution scenarios](image)

### 3.1.2 Body of the Algorithm

Similar to the original Naimi-Trehel algorithm, a request for entering a critical section follows the \textit{owner}'s path until it reaches its \textit{Local\_root} (the last node of the cluster to have requested the CS).

When node \( S_i \) receives a request from \( S_j \), if the former is not a \textit{Local\_root} node (\textit{owner} \( \neq \emptyset \)), it forwards the request to the node pointed to by its \textit{owner}. If \( S_j \) and \( S_i \) belong to the same cluster, \( S_j \) is
stored in $S_i$’s owner variable. However, if $S_i$ is a Local root (owner = $\emptyset$) which is waiting for the token (requesting = true), we distinguish two cases:

- $S_i$’s next = $\emptyset$. This means that $S_j$’s request is the first one since node $S_i$ asked for the token. In this case, the next of $S_i$ is set to the requester $S_j$. Furthermore, if the request came from a node of the same cluster, $S_i$’s owner is set to $S_j$; otherwise its remote owner variable is set to $S_j$.

- $S_i$’s next is already set. Since the receiver is a Local root, next inevitably points to a remote node. In this case, if the requester $S_j$ is a node of $S_i$’s cluster, and the number of preemptions is below the pre-defined threshold, a local preemption is performed i.e., the request from $S_j$ will be satisfied before the remote one pointed by $S_i$’s next. In this case, $S_i$’s current number of preemption is incremented. This preemption value and $S_i$’s next value are sent to the requesting node $S_j$ and $S_i$’s next is updated to $S_j$. Notice that $S_j$ becomes the new Local root. Upon receiving the Prempt message, $S_j$ updates its next to the old next $S_i$’s value.

### 3.2 Two-level clustered algorithm

We propose a second algorithm based on the first one. We add a layer of per cluster proxys (routers). Naimi-Trehel’s algorithm is then applied to this layer too.

In each cluster $C_i$, a node, named Proxy$_i$, is designated to have a second role, as a proxy responsible for centralizing all requests issued from its cluster $C_i$. Thus, every message that is sent from node $S_j$ to node $S_i$, belonging to different clusters, is routed to $S_j$’s proxy node which forwards the message to $S_i$’s proxy node. The latter then sends the message to the receiving node $S_i$. The sender’s and receiver’s proxys can then gather information about token transfers and requests at cluster level, taking decisions based on such information. We can say that each proxy acts as a single node at proxys’ layer.

The same local variables described in section 3.1 for the one-level clustered algorithm were kept. However, the remote owner variable is used only by those nodes that serve as proxys. Each Proxy$_i$ node of $C_i$ needs the new following variables:

- **local owner**: stores the probable token’s owner belonging to $C_i$ of which Proxy$_i$ is aware.

- **remote next**: stored the ID of the node in a remote cluster that will receive the token when it is released by the local cluster.

- **LQueue**: queue that gathers pending requests to remote nodes issued from the nodes of $C_i$. This queue is necessary as a proxy node only sends one CS request to a remote node at a time. Proxy$_i$ forwards a message from the queue each time the that the outstanding request is satisfied.

A new type of message, Stock, has been defined. Nodes send this type of message to their local Proxy node whenever a local request must be sent to a node of a remote cluster. The identification $S_j$ of the requesting node is added to the message. Upon receiving this message, the Proxy node treats it by calling the function Receive Stock($S_j$).

The basic Send function has been modified. If the receiver node belongs to a remote cluster, the function routes the message to the local proxy node of the sender’s cluster. If the sender is a proxy itself and the receiver is a remote node, it forwards the message to the proxy of the remote receiver’s cluster. Its code is as follows (Send called by node $i$) :

```
Send \((Type, \ldots)\) to dest:

if \(dest \in \text{LocalCluster}\) then
Send \((Type, \ldots)\) to dest
else
if \(self = \text{Proxy}_i\) then
Send \((Type, \ldots)\) to \(\text{Proxy}_{dest}\)
else
Send \((Type, \ldots)\) to \(\text{Proxy}_i\)

**Algorithm 4** Two-level hierarchical algorithm - Initialization, Request and Release functions

Every node \(S_i\):

- **Initialization:**
  Does not change from one-level hierarchical algorithm
  
  if \(self = \text{Proxy}_i\) then
  remote\_next \(\leftarrow \emptyset\)
  if \(\text{Elected}\_\text{node} \in \text{LocalCluster}_i\) then
  local\_owner \(\leftarrow \text{Elected}\_\text{node}\)
  remote\_owner \(\leftarrow \emptyset\)
  else
  local\_owner \(\leftarrow \emptyset\)
  remote\_owner \(\leftarrow \text{Proxy}O\_f(\text{Elected}\_\text{node})\)
  \(L\_\text{Queue} \leftarrow \emptyset\)

Request\_CS:

- Does not change from one-level hierarchical algorithm

Release\_CS:

- \(requesting \leftarrow \text{false}\)
  
  if \(next \neq \emptyset\) then
  Send \((\text{Token}, nb\_\text{preempt})\) to \(next\)
  if \(owner = \emptyset\) then
  owner \(\leftarrow \text{Proxy}_i\)
  \{ local\_root : next points to a remote node \}
  token \(\leftarrow \text{false}\)
  next \(\leftarrow \emptyset\)

Based on the identification \(dest\) of the receiver node, \(\text{Proxy}_i\) knows if it should act as a proxy node or as a Naimi-Trehel one i.e., if it is not the \(dest\) node itself, \(\text{Proxy}_i\) must route the message to the node which is the proxy of \(dest\)’s cluster.

Function \(\text{Proxy}O\_f(S_j)\) returns the proxy node of \(S_j\)’s cluster.

Algorithms 4, 5 and 6 summarize our two-level clustered version of Naimi-Trehel’s algorithm.

### 3.2.1 Initialization Phase

The initialization phase is similar to the previous algorithm except for proxy nodes. Their \(\text{remote}\_\text{owner}\) variables point to the proxy of \(\text{Elected}\_\text{node}\)’s cluster (if \(\text{Elected}\_\text{node}\) belongs to a remote cluster). The \(\text{local}\_\text{owner}\) variable of the proxy of \(\text{Elected}\_\text{node}\)’s cluster points to \(\text{Elected}\_\text{node}\).

### 3.2.2 Body of the Algorithm

We distinguish two cases : the node is acting as a non-proxy node or it is acting as a proxy node.

**Non-proxy node:**

The function \(\text{Request}_\text{CS}\) remains as described in 3.1.

Contrary to our previous algorithm, in the function \(\text{Release}_\text{CS}\), the \(\text{Local}\_\text{root} \ S_i\) does not need to update its \(\text{owner}\) variable with \(\text{remote}\_\text{owner}\)’s value when granting the token to a remote node. It just sets its \(\text{owner}\) variable to \(\text{Proxy}_i\) as it knows that the last future \(\text{owner}\) of the token is a remote node.

The core of \(\text{Receive}_\text{Request}\) has been slightly modified. In our previous algorithm, if there is a remote \(CS\) request waiting to be served, the \(\text{Local}\_\text{root} \ C_i\) forwards a new local \(CS\) request to this remote node whenever the number of preemption is greater than a threshold value. In the current two-level algorithm, the \(\text{Local}\_\text{root}\) does not forward this message, but informs the local proxy of its cluster of this new local request by sending a message of type \(\text{Stock}\) to it.

**Proxy node:**

When acting as a proxy, variables \(\text{remote}\_\text{owner}\) and \(\text{remote}\_\text{next}\) have the same role as the \(\text{owner}\) and \(\text{next}\) variables respectively, but at the proxy level.
Algorithm 5 Two-level Hierarchical Algorithm (cont.) - Receive function

Every node $S_i$:

Receive$_{\text{Request-CS}}(S_j)$:

1. if $owner = \emptyset$ then
2. if $requesting = true$ then
   - if $next = \emptyset$ then
     - if $S_j \in \text{LocalCluster}_i$, then
       - $next \leftarrow S_j$
     - if $nb_{\text{preempt}} < \text{Threshold}$ then
       - $nb_{\text{preempt}} \leftarrow nb_{\text{preempt}} + 1$
       - $owner \leftarrow S_j$
       - Send $(Preempt, next, nb_{\text{preempt}})$ to $owner$
     - else
       - Send $(Stock, S_j)$ to Proxy$_i$
       - $owner \leftarrow S_j$
     - else
       - Send $(Request, S_j)$ to $owner$
     - if $S_j \in \text{LocalCluster}_i$, then
       - $owner \leftarrow S_j$
   - else
     - Send $(Preempt, next, nb_{\text{preempt}})$ to $owner$
   - else
     - Send $(Stock, S_j)$ to Proxy$_i$
     - $owner \leftarrow S_j$
   - else
     - Send $(Request, S_j)$ to $owner$
     - if $S_j \in \text{LocalCluster}_i$, then
       - $owner \leftarrow S_j$

Receive$_{\text{Token}}(nb_{\text{preempt}})$:

- Does not change
- from one-level hierarchical algorithm

Receive$_{\text{Preempt}}(nb_{\text{preempt}}, node)$:

- Does not change
- from one-level hierarchical algorithm
Algorithm 6 Two-level Hierarchical Algorithm (cont.) - Proxys’ level functions

Every Proxy $i$:

Proxy\_Receive\_Request\_CS($S_j$):

{ $S_j$ is the requesting process }

if $S_j \in \text{LocalCluster}$ then
  if $\text{local\_owner} = \emptyset$ then
    { no current request from local cluster }
    if $L\text{Queue} = \emptyset$ then
      Send $\langle \text{Request}, S_j \rangle$ to $\text{remote\_owner}$
      $\text{remote\_owner} \leftarrow \emptyset$
      $L\text{Queue} \leftarrow L\text{Queue} + S_j$
    else
      Send $\langle \text{Request}, S_j \rangle$ to $\text{local\_owner}$
      $\text{local\_owner} \leftarrow S_j$
    else
      { remote request }
      if $\text{local\_owner} = \emptyset$ then
        { Token is not requested by the cluster }
        Send $\langle \text{Request}, S_j \rangle$ to $\text{remote\_owner}$
        $\text{remote\_owner} \leftarrow \text{ProxyOf}(S_j)$
      else
        if $\text{remote\_next} = \emptyset$ then
          $\text{remote\_next} \leftarrow \text{ProxyOf}(S_j)$
          $\text{remote\_owner} \leftarrow \text{ProxyOf}(S_j)$
          Send $\langle \text{Request}, S_j \rangle$ to $\text{local\_owner}$
          { redirect request to another cluster }
        else
          Send $\langle \text{Request}, S_j \rangle$ to $\text{remote\_owner}$
          $\text{remote\_owner} \leftarrow \text{ProxyOf}(S_j)$

Proxy\_Receive\_Stock($S_j$):

$L\text{Queue} \leftarrow L\text{Queue} + S_j$
$\text{local\_owner} \leftarrow S_j$

Proxy\_Receive\_Token\_CS($S_j$):

{ Receive the Token from node $S_j$ }

if $S_j \in \text{LocalCluster}$ then
  Send $\langle \text{Token}, S_j \rangle$ to $\text{remote\_next}$
  $\text{remote\_next} \leftarrow \emptyset$
if $L\text{Queue} \neq \emptyset$ then
  Send $\langle \text{Request}, \text{Head}(L\text{Queue}) \rangle$ to $\text{remote\_owner}$
else
  $\text{local\_owner} \leftarrow \emptyset$
  $\text{remote\_owner} \leftarrow \emptyset$
else
  Send $\langle \text{Token}, S_j \rangle$ to $\text{Head}(L\text{Queue})$
  $L\text{Queue} \leftarrow L\text{Queue} - \text{Head}(L\text{Queue})$
A proxy node receives CS Request messages as well as Token messages from both local and remote nodes. It also receives Stock messages from local nodes of its cluster. Functions ProxyReceive_Request_CS, ProxyReceive-Token and ProxyReceive-Stock are called appropriately by the proxy node upon reception of these messages.

The ProxyReceive_Request_CS function distinguishes which treatment to give to a Request message based on the sender’s location:

- The request came from a local node. If the token has not been requested by a node of \( C_i \) (local_owner = \( \emptyset \)), Proxy issues a new CS request, which corresponds to the request made by the node which is at the head of the queue. On the other hand, if it is a remote node that granted the token, the request is redirected to local_owner.

- The request came from a remote node. In the case that local_owner = \( \emptyset \) (no node of the local cluster has currently requested the CS) or remote_next \( \neq \emptyset \) (another remote request already exists), Proxy forwards the receiving request directly to the remote_owner node without needing to redirect it to any of its cluster’s nodes. However, if remote_next = \( \emptyset \), the request is redirected to local_owner.

Variable remote_next is then updated to the proxy of the requesting node \( S_j \). In both cases, the variable local_owner is also set to this same node.

Upon receiving the token, function ProxyReceive-Token is called by Proxy. It verifies from which node it received the token. If the granting node is a local node, the token is sent to remote_node. Furthermore, if \( LQueue \) is not \( \emptyset \), Proxy issues a new CS request, which corresponds to the request made by the node which is at the head of the queue. On the other hand, when the token has already been requested by a node of \( C_i \) (local_owner \( \neq \emptyset \)), the mentioned request is simply redirected to local_owner.

Figure 3 shows an example of the two-level clustered algorithm execution. We suppose that the maximum number of preemption is three.

Figure 3(a) presents tree clusters, \( C_0, C_1, \) and \( C_2 \), where nodes \( A, B, \) and \( C \) belong to cluster \( C_0 \), nodes \( D, E, F, G, \) and \( H \) belong to \( C_1 \) and \( K, I, \) and \( J \) belong to \( C_2 \). Nodes \( B, D \) and \( I \) are the Proxy of the \( C_0, C_1 \) and \( C_2 \) clusters respectively. The token is held by node \( A \) of \( C_0 \). Thus, the remote_owner variables of proxies \( D \) and \( I \) point to \( B \), i.e. the proxy of the Elected Node’s cluster. The local_owner of proxy \( B \) points to the Elected Node.

In Figure 3(b), node \( H \) of \( C_1 \) asks for the CS, sending a request to its owner, which is its proxy (owner_H = \( D \)). As the \( LQueue \) of \( D \) is empty, \( D \) forwards the request to \( B \) (remote_owner_D = \( B \)). \( D \) also queues the request in order to know to which node the token should be granted when \( B \) receives it. The local_owner variable of \( D \) is then updated to \( H \). As the token has previously been requested by \( C_0 \) (local_owner_B = \( A \) and remote_owner = \( \emptyset \)), upon receiving the request, \( B \) sends the request to \( A \), updating its remote_owner and remote_next to \( D \) (the proxy of \( H \)). When node \( A \) (Local_root of \( C_0 \)) receives the request, it sets its next to the requesting process \( H \).

Figure 3(c) shows the requesting of the CS by node \( J \) of \( C_2 \). It sends the request to its owner which is its proxy (owner_j = \( I \)). As in Figure 3(b), node \( I \) forwards the request to \( B \) (remote_owner_I = \( B \)), queues \( J \) in its \( LQueue \) and sets its local_owner to \( J \). Upon receiving the request, \( B \) directly forwards it to its remote_next, since it is \( \neq \emptyset \) (remote_next_B = \( D \)). \( B \) also sets its remote_owner to \( I \). When this request messages is received by \( D \) (proxy of \( H \)’s cluster), the latter sets both its remote_next and
Figure 3: Sample execution of the two-level clustered algorithm
remote_\text{owner} \text{ to } \text{I}. \text{ It then forwards the request to } H. \text{ When } H (\text{Local_root} \text{ of } C_1) \text{ receives the request, it updates its next variable to the requesting node } J.

In Figure 3(d), node \text{G}, \text{F}, \text{and } \text{E} \text{ request the CS in this order. The requests of } \text{G} \text{ and } \text{F} \text{ will pass before the remote request of } \text{J} \text{ as the number of current preemption of } C_1 \text{ is under the maximum threshold (Threshold } = 3). \text{ However, the request of } \text{E}, \text{ the fourth successive local request, can not be satisfied before } J's \text{ request. Thus, when receiving the request, } F (\text{the current Local_root} \text{ of } C_1) \text{ will send a Stock message to its proxy } D, \text{ indicating that the latter must forward the request to the remote node } J. \text{ F updates its owner variable to E. Notice that at this point } next_F = J. \text{ Upon receiving the Stock message from } F, \text{ node } D \text{ will not forward it since a previous request from a node of its cluster (H’s request) has not yet been satisfied. } D \text{ will then queue E in } L\text{Queue and will update is local_\text{owner} to E.}

When node A releases the token, Figure 3(e), it will send it to H (next_A = H) through the respective proxy nodes. It will also update its owner to B, the proxy node of its own cluster, as it knows that the owner of the token is now a remote node. When proxy B receives the message with the token, it will forward it to its remote_\text{owner} (remote_\text{owner}_B = D). \text{ It will also reset its remote_next variable and its local_\text{owner} variables (as B’s } L\text{Queue} = \emptyset). \text{ When the Token message arrives at } D, \text{ it will be forwarded to } H, \text{ the node at the head of D’s } L\text{Queue}. \text{ H will be then removed from } D’s \text{ } L\text{Queue.}

4 Related Work

Besides Naimi and Trehel, other authors proposed O(log N ) token-based algorithms exploiting tree structures. Raymond’s algorithm [11] organizes nodes 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 edges always point to the possible token holder. Neilsen and Mizuno [10] extended this algorithm by passing the token directly to the requesting node instead of through intermediate nodes. Chang Singhal and Liu [4] improved Naimi-Trehel’s algorithm, aiming at reducing the number of messages to find the last requesting node in the logical tree.

Mueller [8] has 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. We can say that we adopt a similar strategy when satisfying intra-cluster requests before inter-cluster ones. However, in our algorithms, the number of assignment is limited by a pre-defined threshold value.

Housni et al. [5] and Chang et al. [3]’s mutual exclusion algorithms gather nodes into groups. Both articles basically propose hybrid approaches where the algorithm for intra-group requests is different from the inter-group one. In Housni et al. [5], sites with the same priority are gathered at the same group. Raymond’s tree-based token algorithm [11] is used inside a group, while Ricart-Agrawala [12] diffusion-based algorithm is used between groups. Chang et al.’s [3] hybrid algorithm applies diffusion-based algorithms at both levels: Singhal’s algorithm [14] locally, and Maekawa’s algorithm [7] between groups. The former uses a dynamic information structure while the latter is based on a voting approach. Our work is related to these articles in the gathering of machines into groups (clusters in our case) influences the conception of the algorithm. However, the authors do not consider differences in communication latency as the main reason for grouping machines.

In [1], 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.

We have presented in a previous work [2] a hierarchical token-based algorithm for multi-cluster platforms which is also based on Naimi-Trehel algorithms. However, this algorithm is not a router-based one and a global queue is used for aggregating remote requests. This approach reduces the number of inter-cluster
messages but introduces some lack of fairness to the algorithm.

5 Performance evaluation

This section describes a set of performance evaluation experiments aimed at comparing the efficiency of our mutual exclusion algorithms with the original Naimi-Trehel algorithm. The three algorithms considered are:

- \textit{NaimiTrehel} algorithm, which implements Naimi-Trehel token-based algorithm presented in section 2.
- \textit{one-level} algorithm, which implements the one-level clustered algorithm presented in section 3.1.
- \textit{two-level} algorithm, which implements the two-level clustered algorithm presented in section 3.2

5.1 Experimental testbed and configuration

The experiments described in this section were performed on a dedicated cluster of sixteen Pentium IV 2.66 GHz computers linked by a 1 Gbits/s Ethernet switch. The algorithms were implemented in Java (Sun’s JDK 1.4) on top of the Linux 2.4 kernel.

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 [13] and IPNAT. The latter is an IP masquerading application that divides the network into virtual LANs (clusters). DUMMYNET is a flexible tool originally designed for testing network protocols. It simulates bandwidth limitations, delays, and packet losses. Based on addresses and ports of both destination and source nodes, DUMMYNET intercepts packets, passing them through one or more queues and pipes, which simulate different message transmission configurations. In our experiment, every message exchanged between two different clusters passes through a dedicated machine which runs DUMMYNET.

Each machine runs three “virtual nodes”, emulating a platform with 48 nodes grouped in 3 clusters of 16 nodes. However, each “virtual node” of a machine belongs to a different cluster. Therefore, communication between each other is performed through network, always passing by the dedicated machine that runs DUMMYNET. In the rest of this section, we called \textit{node} a “virtual node” and not a physical machine.

The topology of the platform is known at the outset by every node as well as the initial holder of the token (\textit{Elected node}). In each experiment, every node issues 10 critical section.

Experiments are characterized by:

- \(\alpha\): time taken by a node to execute the critical section,
- \(\beta\): mean time interval between the release of the CS by a node and the request of it by this same node.
- \(\epsilon\): preemption threshold (only for the \textit{hierarchical} algorithm),
- \(\gamma\): delay introduced in inter-cluster communication.

For each experiment, the following metrics are considered:

- \textbf{number of messages}, divided in two categories: \textit{local messages}, exchanged between two nodes within the same cluster and \textit{global messages}, exchanged between two nodes of different clusters. The ratio of \textit{local message} per \textit{global message} is also calculated.
- \textbf{obtaining time}: time between the moment a node requests the critical section and the moment it gets it. We measure the average as well as the standard deviation (\textit{ST DEV}) of the obtaining time.
- \textbf{number of preemptions}: number of preemptions during an evaluation test.

5.2 Results and Discussion

The influence of the application behavior, number of preemption and latency between clusters have been studied in our performance measures.
5.2.1 Application behavior influence

The aim of the current experiments is to observe the behavior of each algorithm when $\beta$ and $\alpha$ vary. We called that “application behavior influence” as the ratio $\beta/\alpha$ expresses the frequency with which the critical section is requested. Table 1 summarizes performance measures as function of the ratio $\beta/\alpha$. Basically, for all measurements, except the last one, the mean time in the CS, $\alpha$, is fixed to 0.5, while the time interval between the execution of two successive CS by a node, $\beta$, varies. The different configurations are (in s): 10/0.5, 5/0.5, 2/0.5, 0.5/0.5 and 0.5/1.

<table>
<thead>
<tr>
<th>Type</th>
<th>Ratio $\beta/\alpha$</th>
<th>Obtaining time (s)</th>
<th>number of preemption</th>
<th>Nb of messages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>average</td>
<td>ST DEV</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Naimi-Trehel</td>
<td>20</td>
<td>15.21</td>
<td>2.46</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>19.34</td>
<td>3.28</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>22.55</td>
<td>4.14</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>24.19</td>
<td>4.63</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0.5</td>
<td>45.71</td>
<td>8.51</td>
<td>-</td>
</tr>
<tr>
<td>One-level</td>
<td>20</td>
<td>14.29</td>
<td>2.35</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>18.4</td>
<td>8.83</td>
<td>96</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>21.19</td>
<td>9.03</td>
<td>96</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>22.64</td>
<td>9.18</td>
<td>98</td>
</tr>
<tr>
<td></td>
<td>0.5</td>
<td>44.45</td>
<td>18.03</td>
<td>99</td>
</tr>
<tr>
<td>Two-level</td>
<td>20</td>
<td>14.22</td>
<td>2.37</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>18.4</td>
<td>11.5</td>
<td>146</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>20.91</td>
<td>11.58</td>
<td>147</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>22.57</td>
<td>11.67</td>
<td>146</td>
</tr>
<tr>
<td></td>
<td>0.5</td>
<td>44.87</td>
<td>22.88</td>
<td>146</td>
</tr>
</tbody>
</table>

Table 1: Application behavior influence

For all algorithms, when the ratio $\beta/\alpha$ decreases, the obtaining time increases. This can easily be explained as the probability that other nodes have also requested the CS increases as well. In the case of our algorithms, when the ratio $\beta/\alpha$ is equal to 20, the preemption mechanism is rarely exploited (just one or two preemptions). This happens because there are not many simultaneous requests within a cluster. However, when the ratio $\beta/\alpha$ is equal or smaller than 10, the preemption mechanism becomes effective. At the same time, we observe that the standard deviation increases and messages are more concentrated inside clusters. For ratios equal 10 — 1, the behavior of the algorithms does not change very much because these ratios are relative low: when a node requests the token, almost all others have requested it too. As in these cases only token transmission time can be reduced; the variation of the obtaining time is not very significant. When the ratio $\beta/\alpha$ is smaller that 1, the obtaining time and the standard deviation almost double since all nodes stay twice longer in the critical section than in non critical section.

5.2.2 Preemption influence

The set of the current experiments allows us to evaluate the influence of the preemption threshold on our algorithms’ behavior. They are characterized by: $\alpha = \beta = 500$ ms, and $\gamma = 100$ ms. Figure 4.(a) presents the obtaining time and the Figure 4.(b) the ratio local messages / global messages when the preemption threshold increases. Table 2 also summarizes these experiments.

A first comment about these results is that even when the preemption threshold $\epsilon$ is equal to 0, the obtaining time of our algorithms is reduced when compared to Naimi-Trehel algorithm. In fact, the Local_root approach that we introduced in our algorithms informs a node that another node of the same cluster has already requested the CS, thus avoiding remote requests. Local token transmission is then prioritized.

Naturally, when increasing $\epsilon$, the number of preemption increases too. However, the obtaining time decreases while its standard deviation increases significantly. The former goes down because there are fewer remote token transmission while the latter goes up because preemption speeds up local requests,
Table 2: Summarize preemption influence
slowing down remote ones. This happens since our algorithms adopts a token transmission strategy that relazes fairness for token obtaining time, but preserves fairness based on the time that a single cluster keeps the token.

We also observe that the preemption mechanism is more effective for the two-level algorithm than for the one-level algorithm. The reason for this can be justified as follows. In the two-level algorithm only the first remote request is delivered directly. Subsequent requests are handled at the proxy level. Therefore, the remote request is less redirected, arriving faster at a Local\textsubscript{root} node of the final cluster. This increases the probability for this message to be preempted by a local request of this cluster.

A last worth remark is that the impressive concentration of messages per cluster of our algorithms. With Naimi-Trehel’s algorithm, there are 1.36 local per global message. On the other hand, for $\epsilon = 16$, there are 38.12 and 51.65 local per global message for the one-level algorithm and two-level one respectively.

5.2.3 Platform influence

Figure 5 illustrates the evolution of the obtaining time when inter-cluster delay increases. For these experiments, we consider $\alpha = \beta = 500\ ms$ and $\epsilon = 8$. Inter-cluster delay ($\gamma$) varies from 0 ms to 200 ms.

![Figure 5: Preemption influence](image)

Results show that our algorithms scale better than Naimi-Trehel’s. We observe that the gap between our algorithms’s obtaining time and Naimi-Trehel algorithm’s obtaining time increases when inter-cluster delay increases too. We can explain this behavior since our algorithms concentrate communication inside clusters. Thus, distance between clusters has less influence in our algorithms than in Naimi-Trehel’s.

6 Conclusion

We have presented in this paper a new approach to optimize mutual exclusion algorithms for Grid environment. The main idea of our work is to adapt Naimi-Trehel algorithm to the network topology, minimizing inter-cluster messages. Two different hierarchical algorithms were proposed. Both exploit the same mechanisms such as per cluster Local\textsubscript{root} node and preemption of local requests, but the second algorithm has an extra layer of proxy nodes.

Performance evaluation results, discussed in section 5, conclude that our algorithms minimize the time that a node waits for the token compared to Naimi-Trehel algorithm. Between our two algorithms, the two-level clustered one shows to be more efficient. Furthermore, the latter could be easily generalized to a n-level hierarchical organization. However, we must point out that our algorithms suffer from some
unfairness due to the higher priority given to local CS requests. In fact, our algorithms consider token possession time by a single cluster as a factor for fairness instead of the time to obtain a token.

References


