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 *Receive_Request_CS(S_j)* is called upon reception of this type of message from S_j.

- *Token* : represents the transmission of the token. The function *Receive_Token(S_j)* is called upon reception of this type of message from S_j.

Algorithm 1 Naimi-Trehel

Every node S_i:

Initialization:

```
  resquesting ← false
  next ← ∅
  if self = Elected_node then
    token ← true
    owner ← ∅
  else
    token ← false
    owner ← Elected_node
```

Request_CS:

```
  requesting ← true
  if owner ≠ ∅ then
      { The site hasn't the token, it should request it }
    Send ⟨Resquest, S_i⟩ to owner
    owner ← ∅
    Wait for receiving message ⟨Token⟩
```

Release_CS:

```
  requesting ← false
  if next ≠ ∅ then
    Send ⟨Token⟩ to next
    token ← false
    next ← ∅
```

Receive_Request_CS(S_j) :

```
                                   { S_j is the requesting node }
  if owner = ∅ then
                                                    { root node }
    if resquesting = true then
                     { The node asked for the Critical Section }
      next ← Sj
    else
                 { First request to the token since the last CS:
               send the token directly to the requesting node }
      token ← false
      Send ⟨Token⟩ to S_j
  else
                           { Non-root node, forward the request }
    Send ⟨Request, S_j⟩ to owner
  owner ← S_j
```

Receive_Token(S_j):

```
                               { Receive the token from node S_j }
  token ← true
```

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* ($owner_B = A$). B 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. The request is forwarded to B which updates its *next* to C ($next_B = C$). 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 $next_A = B$.

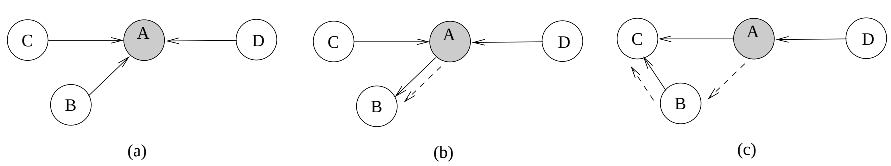

Figure 1: Example of Naimi-Trehel's algorithm execution

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 *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 *next* links is modified for serving local requests before remote ones.

Note: The last node which will enter the critical section within a cluster is named *Local_root*. It is identified by the fact that its *owner* variable is equal to $\emptyset$.

Algorithm 2 Hierarchical Algorithm - Initialization, Request and Release functions

```
Every node S_i:

Initialization:
  requesting ← false
  next ← ∅
  remote_owner ← ∅
  if Elected_node ∈ LocalCluster_i then
    if Elected_node = self then
      token ← true
      owner ← ∅
    else
      token ← false
      owner ← Elected_node
  else
    token ← false
    if self = Proxy_i then
      owner ← Elected_node
    else
      owner ← Proxy_i

Request_CS:
  nb_preempt ← 0
  requesting ← true
  if owner ≠ ∅ then
        { The node hasn't the token, it requests it }
    Send ⟨Request, S_i⟩ to owner
    owner ← ∅
    Wait for receiving message ⟨Token⟩

Release_CS:
  requesting ← false
  if next ≠ ∅ then
    if next ∉ LocalCluster then
        { The token will be sent to a remote node }
      nb_preempt ← 0
      if owner = ∅ then
        owner ← remote_owner
      remote_owner ← ∅
    Send ⟨Token, nb_preempt⟩ to next
    token ← false
    next ← ∅
```

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

- *LocalCluster$_i$*: identifies the cluster to which node S_i belongs. All nodes are aware of it.
- *remote_owner*: the *Local_root* node updates this variable whenever it receives a first CS request from a remote node. Its *owner* variable will only be updated with the *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 = ∅ then
                                        { S_i is the local_root }
  if resquesting = true then
                                        { The node asked for CS }
    if next = ∅ then
      next ← S_j
      if S_j ∈ LocalCluster_i then
        owner ← S_j
      else
        remote_owner ← S_j
    else
      if S_j ∈ LocalCluster
        and nb_preempt < Threshold then
        { Local preemption of the token by the sender }
        nb_preempt ← nb_preempt + 1
        owner ← S_j
        Send ⟨Preempt, nb_preempt, next⟩ to S_j
        next ← S_j
      else
        Send ⟨Request, S_j⟩ to next
        owner ← S_j
  else
    token ← false
    Send ⟨Token, nb_preempt⟩ to S_j
    owner ← S_j
else
  Send ⟨Request, S_j⟩ to owner
  if S_j ∈ LocalCluster_i then
    owner ← S_j
```

Receive_Token(S_j **):**

{ *Receive the token from node* S_j }

```
token ← true
```

Receive_Preempt(*nb_preempt*$_j$**,** *remote_node***):**

```
nb_preempt ← nb_preempt + nb_preempt_j
if next = ∅ then
                                        { S_i is the local_root }
  next ← remote_node
  remote_owner ← remote_node
else
  Send ⟨Preempt, nb_preempt, remote_node⟩ to owner
```

- *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 *Preempt* message (see below).

A third type of message, *Preempt*, has been defined. This message passes around the value of *nb_preempt* of the sender node as well as the identification of the node that should be preempted. The function *Receive_Preempt* is called upon reception of this message.

The type *Token* of message has also been modified for including the value of the local variable *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 *owners* to the multi-cluster topology of the network. Every cluster C_i, except the *Elected_Node*'s cluster, designates a local node to which all the other nodes in the cluster should point to. This node is called $Proxy_i$. It is worth remarking that while we named this node as *Proxy*, 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 *owner* variable of a $Proxy_i$ as well as the nodes that belong to *Elected_Node*'s cluster point to the *Elected_Node*. However, the *owner* variable of the other nodes S_i points to the $Proxy_i$ 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 $Proxy_1$ of C_1. Since A has been elected to have the token, it is the *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 *owner* to D. When the request arrives at A, the latter updates its *next* and *remote_owner* variables to D. E asks then for the token. Node F, the $Proxy_1$, locally redirects the request to D which updates its *next* and *owner* variables to E. F also updates its *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.

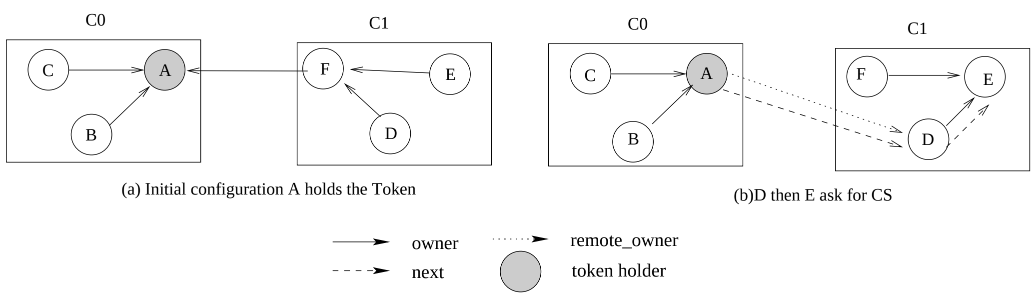

Figure 2: Hierarchical execution scenarios

3.1.2 Body of the Algorithm

Similar to the original Naimi-Trehel algorithm, a request for entering a critical section follows the *owner*'s path until it reaches its *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 *Local_root* node (*owner* $\neq \emptyset$), it forwards the request to the node pointed to by its *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 proxies 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 proxies. 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.

- *L_Queue*: 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 $\langle Type, \ldots \rangle$ **to** $dest$**:**

if $dest \in LocalCluster$ **then**
  Send $\langle Type, \ldots \rangle$ to $dest$
else
  if $self = Proxy_i$ **then**
    Send $\langle Type, \ldots \rangle$ to $Proxy_{dest}$
  else
    Send $\langle Type, \ldots \rangle$ to $Proxy_i$

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 $self = Proxy_i$ **then**
  $remote\_next \leftarrow \emptyset$
  if $Elected\_node \in LocalCluster_i$ **then**
    $local\_owner \leftarrow Elected\_node$
    $remote\_owner \leftarrow \emptyset$
  else
    $local\_owner \leftarrow \emptyset$
    $remote\_owner \leftarrow ProxyOf(Elected\_node)$
  $L\_Queue \leftarrow \emptyset$

Request_CS:

Does not change from one-level hierarchical algorithm

Release_CS:

$requesting \leftarrow false$
if $next \neq \emptyset$ **then**
  Send $\langle Token, nb\_preempt \rangle$ to $next$
  if $owner = \emptyset$ **then**
    $owner \leftarrow Proxy_i$
      { *local_root : next points to a remote node* }
  $token \leftarrow false$
  $next \leftarrow \emptyset$

Based on the identification *dest* of the receiver node, $Proxy_i$ 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, $Proxy_i$ must route the message to the node which is the *proxy* of *dest*'s cluster.

Function $ProxyOf(S_j)$ 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 *remote_owner* variables point to the *proxy* of *Elected_node*'s cluster (if *Elected_node* belongs to a remote cluster). The *local_owner* variable of the *proxy* of *Elected_node*'s cluster points to *Elected_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 *Request_CS* remains as described in 3.1.

Contrary to our previous algorithm, in the function *Release_CS*, the *Local_root* S_i does not need to update its *owner* variable with *remote_owner*'s value when granting the token to a remote node. It just sets its *owner* variable to $Proxy_i$ as it knows that the last future *owner* of the token is a remote node.

The core of *Receive_Request* has been slightly modified. In our previous algorithm, if there is a remote CS request waiting to be served, the *Local_root* of 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 *Local_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 *Stock* to it.

Proxy node:

When acting as a *proxy*, variables *remote_owner* and *remote_next* have the same role as the *owner* and *next* variables respectively, but at the *proxy* level.

Algorithm 5 Two-level 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 $resquesting = true$ **then**

{ *The node asked for CS* }

    if $next = \emptyset$ **then**

      $next \leftarrow S_j$

      if $S_j \in LocalCluster_i$ **then**

        $owner \leftarrow S_j$

    else

      if $nb\_preempt < Threshold$ **then**

        { *Local preemption of the token by the sender* }

        $nb\_preempt \leftarrow nb\_preempt + 1$

        $owner \leftarrow S_j$

        Send $\langle Preempt, next, nb\_preempt \rangle$ to owner

        $next \leftarrow S_j$

      else

        Send $\langle Stock, S_j \rangle$ to $Proxy_i$

        $owner \leftarrow S_j$

  else

    $token \leftarrow false$

    Send $\langle Token, nb\_preempt \rangle$ to S_j

    $owner \leftarrow S_j$

else

  Send $\langle Request, S_j \rangle$ to $owner$

  if $S_j \in LocalCluster_i$ **then**

    $owner \leftarrow S_j$

Receive_Token($nb\_preempt_j$)**:**

Does not change
from one-level hierarchical algorithm

Receive_Preempt($nb_p reempt_j$, $node$)**:**

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 ∈ LocalCluster then
  if local_owner = ∅ then
        { no current request from local cluster }
    if L_Queue = ∅ then
      Send ⟨Request, S_j⟩ to remote_owner
      remote_owner ← ∅
    L_Queue ← L_Queue + S_j
  else
    Send ⟨Request, S_j⟩ to local_owner
  local_owner ← S_j
else
        { remote request }
  if local_owner = ∅ then
        { Token is not requested by the cluster }
    Send ⟨Request, S_j⟩ to remote_owner
    remote_owner ← ProxyOf(S_j)
  else
    if remote_next = ∅ then
      remote_next ← ProxyOf(S_j)
      remote_owner ← ProxyOf(S_j)
      Send ⟨Request, S_j⟩ to local_owner
        { redirect request to another cluster }
    else
      Send ⟨Request, S_j⟩ to remote_owner
      remote_owner ← ProxyOf(S_j)
```

Proxy_Receive_Stock(S_j):

```
L_Queue ← L_Queue + S_j
local_owner ← S_j
```

Proxy_Receive_Token_CS(S_j):

{ *Receive the Token from node S_j* }

```
if S_j ∈ LocalCluster then
  Send ⟨Token, S_j⟩ to remote_next
  remote_next ← ∅
  if L_Queue ≠ ∅ then
    Send ⟨Request, Head(L_Queue)⟩ to remote_owner
  else
    local_owner ← ∅
  remote_owner ← ∅
else
  Send ⟨Token, S_j⟩ to Head(L_Queue)
  L_Queue ← L_Queue − Head(L_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 *Proxy_Receive_Request_CS*, *Proxy_Receive_Token* and *Proxy_Receive_Stock* are called appropriately by the *proxy* node upon reception of these messages.

The *Proxy_Receive_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 ($local\_owner = \emptyset$), $Proxy_i$ must store S_j at the end of *L_Queue* to know to which node it will forward the token when it receives it from a remote node. Furthermore, if no previous remote request is pending, $Proxy_i$ should send the new local request to *remote_owner*. On the other hand, when the token has already been requested by a node of C_i ($local\_owner \neq 0$), the mentioned request is simply 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), $Proxy_i$ 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 *remote_owner* is also set to this same node.

Upon receiving the *token*, function *Proxy_Receive_Token* is called by $Proxy_i$. 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 *L_Queue* is not $\emptyset$, $Proxy_i$ 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 latter is forwarded to the node at the head of *L_Queue* and this node is remove from the queue.

Function *Proxy_Receive_Stock* queues local CS requests which will later be forwarded to a remote node. The requests are put at the tail of *L_Queue*. It is worth remembering that a *proxy* node can only issue one CS request message to a remote node at a time to avoid cycles in *owner*'s path.

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 $Proxy_i$ 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 *proxys* 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 *L_Queue* 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 *L_Queue* 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

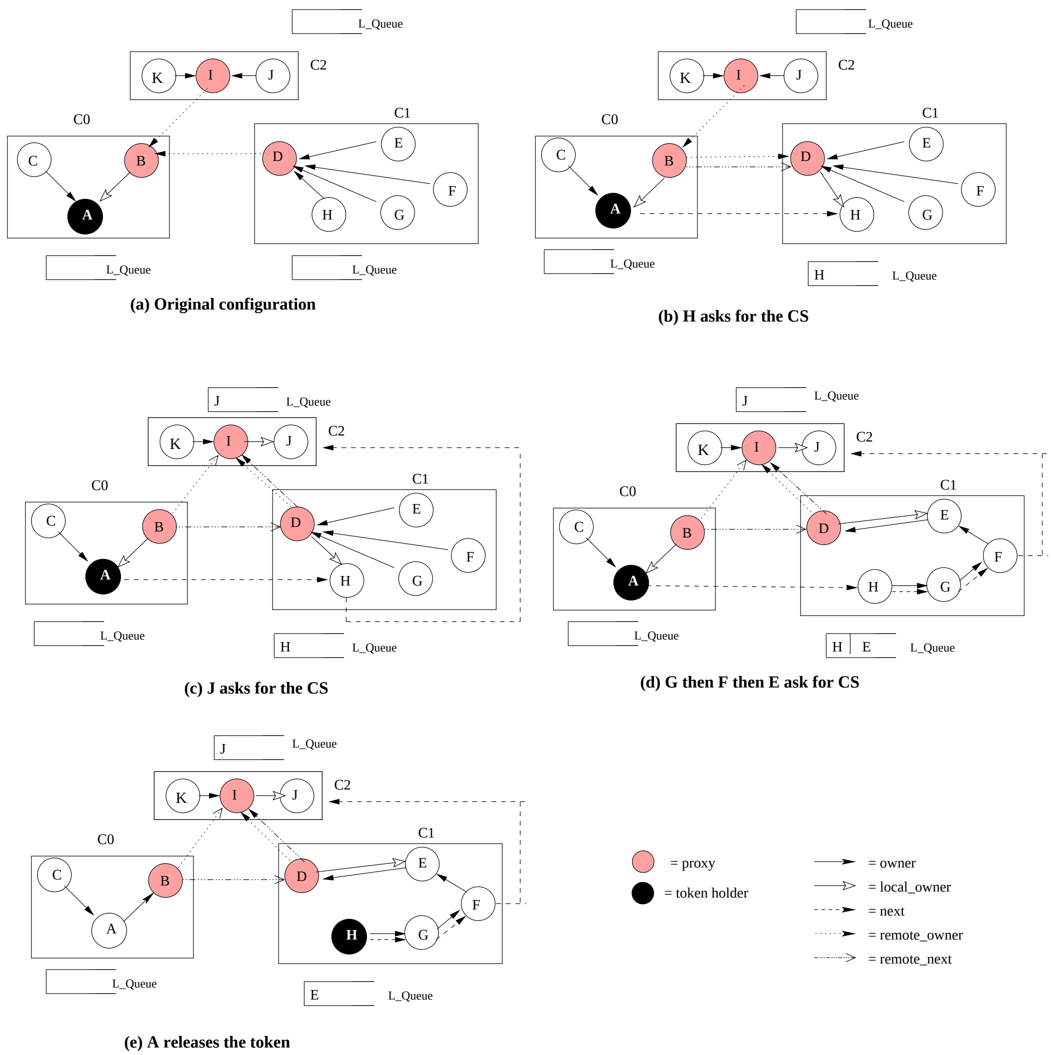

Figure 3: Sample execution of the two-level clustered algorithm

remote_owner to I. It then forwards the request to H. When H (*Local_root* of C_1) receives the request, it updates its *next* variable to the requesting node J.

In Figure 3(d), node G, F, and E request the CS in this order. The requests of G and F will pass before the remote request of J as the number of current preemption of C_1 is under the maximum threshold ($Threshold = 3$). However, the request of E, the fourth successive local request, can not be satisfied before J's request. Thus, when receiving the request, F (the current *Local_root* of $C1$) will send a *Stock* message to its *proxy* D, indicating that the latter must forward the request to the remote node J. F updates its *owner* variable to E. Notice that at this point $next_F = J$. Upon receiving the *Stock* message from F, node D will not forward it since a previous request from a node of its cluster (H's request) has not yet been satisfied. D will then queue E in *L_Queue* and will update is *local_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_owner* ($remote\_owner_B = D$). It will also reset its *remote_next* variable and its *local_owner* variables (as B's $L\_Queue = \emptyset$). When the *Token* message arrives at D, it will be forwarded to H, the node at the head of D's *L_Queue*. H will be then removed from D's *L_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:

- $NaimiTrehel$ algorithm, which implements Naimi-Trehel token-based algorithm presented in section 2.
- $one - level$ algorithm, which implements the one-level clustered algorithm presented in section 3.1.
- $two - level$ 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 *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 (*Elected_node*). In each experiment, every node issues 10 critical section.

Experiments are characterized by:

- α: time taken by a node to execute the critical section,
- β: mean time interval between the release of the CS by a node and the request of it by this same node.
- ϵ: preemption threshold (only for the $hierarchical$ algorithm),
- γ: delay introduced in inter-cluster communication.

For each experiment, the following metrics are considered:

- **number of messages**, divided in two categories: *local messages*, exchanged between two nodes within the same cluster and *global messages*, exchanged between two nodes of different clusters. The ratio of *local message* per *global message* is also calculated.
- **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 (*ST DEV*) of the obtaining time.
- **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 β and α vary. We called that "application behavior influence" as the ratio β/α expresses the frequency with which the critical section is requested. Table 1 summarizes performance measures as function of the ratio β/α. Basically, for all measurements, except the last one, the mean time in the CS, α, is fixed to 0.5, while the time interval between the execution of two successive CS by a node, β, varies. The different configurations are (in s): 10/0.5, 5/0.5, 2/0.5, 0.5/0.5 and 0.5/1.

Type	Ratio	Obtaining time (s)		number of	Nb of messages		
	β/α	average	ST DEV	preemption	local	global	%
Naimi-Trehel	20	15.21	2.46	-	1032	782	1.32
	10	19.34	3.28	-	1073	750	1.43
	4	22.55	4.14	-	1087	770	1.41
	1	24.19	4.63	-	1067	785	1.36
	0.5	45.71	8.51	-	1089	728	1.5
One-level	20	14.29	2.35	1	1826	74	24.68
	10	18.4	8.83	96	1952	62	24.68
	4	21.19	9.03	96	1963	62	31.48
	1	22.64	9.18	98	1965	62	31.66
	0.5	44.45	18.03	99	1960	60	31.69
Two-level	20	14.22	2.37	2	1873	72	26.01
	10	18.43	11.5	146	1997	51	39.16
	4	20.91	11.58	147	2006	50	40.12
	1	22.57	11.67	146	2015	50	40.3
	0.5	44.87	22.88	146	2015	50	40.3

Table 1: Application behavior influence

For all algorithms, when the ratio β/α 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 β/α 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 β/α 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 $10 \rightarrow 1$, 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 β/α 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 ϵ 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 ϵ, 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,

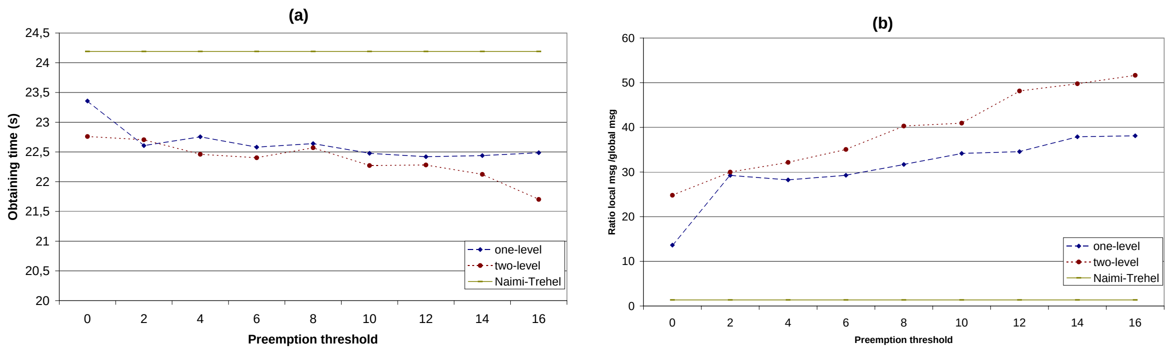

Figure 4: Evolution of preemption

Type	Preemption threshold	Obtaining time (s)		number of preemption	Nb of messages		
		average	ST DEV		local	global	%
one-level	0	23.36	4.363	0	1868	137	13.64
	2	22.61	5.86	27	2106	72	29.25
	4	22.76	7.01	50	1892	67	28.24
	6	22.58	8.14	73	1932	66	29.27
	8	22.64	9.17	98	1965	62	31.69
	10	22.48	9.76	105	1947	57	34.16
	12	22.42	10.61	121	1970	57	34.56
	14	22.44	11.32	140	1970	52	37.88
	16	22.49	11.84	146	1982	52	38.12
Two-level	0	22.76	4.22	0	1909	77	24.79
	2	22.71	6.98	50	1949	65	29.98
	4	22.46	9.1	96	1994	62	32.16
	6	22.4	10.52	124	1998	57	35.05
	8	22.57	11.67	146	2015	50	40.3
	10	22.27	13.1	180	2047	50	40.94
	12	22.28	14.01	190	2022	42	48.14
	14	22.12	14.97	210	2090	42	49.76
	16	21.7	16.15	240	2066	40	51.65

Table 2: Summarize preemption influence

slowing down remote ones. This happens since our algorithms adopts a token transmission strategy that relaxes 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_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 (γ) varies from 0 ms to 200 ms.

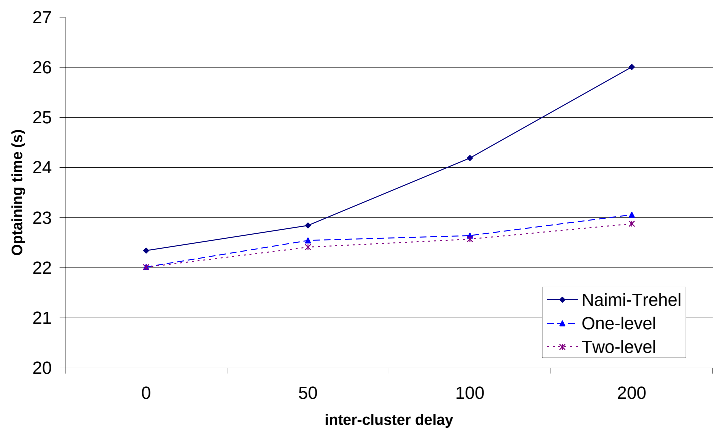

Figure 5: Preemption influence

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_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

[1] G. Antoniu, L. Bouge, 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*, pages 516–521, 2003.

[2] M. Bertier, L. Arantes, and P. Sens. Hierarchical token based mutual exclusion algorithms. In *4th IEEE/ACM International Symposium on Cluster Computing and the Grid*, 10 April 2004.

[3] 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*, pages 289–294, 1990.

[4] I. Chang, M. Singhal, and M. T. Liu. An improved O(log N) mutual exclusion algorithm. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages 295–302, August 1990.

[5] 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*, pages 26–29, June 2001.

[6] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–564, July 1978.

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

[8] 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*, pages 791–795, March 1998.

[9] M. Naimi, M. Trehel, and A. Arnold. A log (N) distributed mutual exclusion algorithm based on path reversal. *Journal of Parallel and Distributed Computing*, 34(1):1–13, 10 April 1996.

[10] 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)*, pages 354–360, Washington, DC, 1991.

[11] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems (TOCS)*, 7(1):61–77, 1989.

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

[13] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, 1997.

[14] M. Singhal. A dynamic information structure for mutual exclusion algorithm for distributed systems. *IEEE Transactions on Parallel Distributed Systems*, 3(1):121–125, 1992.

[15] I. Suzuki and T. Kasami. A distributed mutual exclusion algorithm. *ACM Transactions on Computer Systems (TOCS)*, 3(4):344–349, 1985.