RENPAR'16 / CFSE'4 / SympAAA'2005
Le Croisic, France, 6 au 8 Avril 2005

Algorithme d'exclusion mutuelle pour les GRID : une approche hiérarchique

Marin BERTIER[1] Luciana ARANTES[2] Pierre SENS[2]
marin.bertier@lri.fr luciana.arantes@lip6.fr pierre.sens@lip6.fr
[1] Laboratoire de Recherche en Informatique, Université de Paris-Sud, 91405 Orsay cedex
[2] Laboratoire d'Informatique de Paris 6, Université Paris 6, 75005 Paris

Résumé
Actuellement, la majorité des algorithmes d'exclusion mutuelle ne sont pas adaptés aux architectures de type Grille de Calcul (GRID). Nous proposons, dans cet article, deux algorithmes d'exclusion mutuelle distribuée optimisé pour ce contexte. Ces algorithmes sont des extensions de l'algorithme à jeton de Naïmi-Trehel, qui prennent en compte la dissymétrie entre les communications inter et intra-cluster. Le premier de nos algorithmes permet d'exploiter la localité dans les cluster en donnant une plus grande priorité au demande issue d'un même cluster. Le second ajoute une couche de routeur à notre premier algorithme, se qui permet de minimiser les communications entre les clusters. Cet article est conclu par une évaluation de performances permettant de confirmer les avantages de notre approche.

Mots-clés : exclusion mutuelle, GRID, algorithme à jeton

Plusieurs applications réparties bénéficient des architectures en Grille, qui permettent de partager une grande variété de ressources distribuées géographiquement. Cependant, de telles applications nécessitent que leurs processus puissent avoir un accès exclusif à des ressources partagées. Par conséquent, l'exclusion mutuelle est un problème cruciale dans un tel contexte.
En d'autres termes, L'exclusion mutuelle assure à un processus qu'il est le seul processus du système à avoir accès à la ressource pendant un certain temps. Nous appelons le temps où le processus utilise la ressource la *section critique*. L'exclusion mutuelle revient donc à rendre séquentielle des accès concurrents à une ressource partagée.
Il existe plusieurs algorithmes qui permettent de résoudre en réparti le problème de l'exclusion mutuelle. Ils peuvent être divisés en deux groupes : ceux *à base de permission* [5, 11, 13, 6] et ceux *à base de jeton* [14, 10, 8, 9, 3]). Le premier groupe d'algorithmes est basé sur le principe qu'un nœud entre en section critique seulement s'il a reçu la permission de tous les autres processus (ou d'une majorité d'entre eux [6]). Dans le deuxième groupe, un unique jeton est partagé par tous les sites et sa possession par un site donne le droit d'accéder à la section critique. Les algorithmes à jeton sont généralement peu coûteux en messages : au mieux la complexité en nombre de messages est $O(\log(n))$ où n est le nombre de sites du système.
Les algorithmes que nous proposons sont basés sur l'algorithme de Naïmi-Tréhel [8] qui est un algorithme en $O(\log(n))$ basé sur un arbre dynamique. L'arbre logique est entretenu afin que sa racine soit toujours le dernier site qui a demandé le jeton. Chaque site pointe vers le dernier site lui ayant demandé le jeton.
Cet article est organisé de la façon suivante : La deuxième section présente différents travaux sur l'exclusion mutuelle. La section suivante présente l'algorithme de Naïmi-Trehel. La section 3 est dédiée à la présentation de nos algorithmes hiérarchiques. La section 4 introduit des expérimentations permettant de comparer les différents algorithmes introduit dans cet article. Finalement, la dernière section conclue ce travail.

1. Différentes approches sur l'exclusion mutuelle

En complément de l'algorithme de Naimi et Trehel, d'autres auteurs ont proposés des algorithmes en O(log (N)) utilisant une architecture en arbre. L'algorithme de Raymond [10] organise les sites dans une structure d'arbre statique dont la racine est le possesseur du jeton. L'arbre reste inchangé mais l'orientation des arcs changent dynamiquement en fonction de la propagation du jeton. Neilsen et Mizuno [9] ont réalisé une extension de cet algorithme qui permet de transmettre le jeton directement au site demandant le jeton au lieu de traverser les différents sites intermédiaires. Chang, Singhal et Liu [3] améliorent l'algorithme de Naïmi-Tréhel afin de réduire le nombre de messages nécessaires pour trouver la racine de l'arbre.

Mueller [7] a proposé une autre extension pour l'algorithme de Naïmi-Tréhel, en introduisant le concept de priorité. Une demande de jeton est associée à une priorité et l'algorithme satisfait en premier la requête avec la priorité la plus grande. Il n'existe pas dans cette approche de limite au nombre de requêtes propriétaires à satisfaire, il y a donc des cas de famine : certains sites peuvent ne jamais obtenir le jeton.

Housni et al. [4] et Chang et al. [2] introduisent dans leur algorithme le concept de groupe. Les deux articles proposent des approches hybrides où l'algorithme pour les requêtes à l'intérieur du groupe est différent de celui pour les requêtes entre les groupes. Dans Housni et al. [4] les sites qui ont la même priorité sont regroupés et l'algorithme à jeton de Raymond est utilisé à l'intérieur du groupe alors qu'entre les groupes, l'algorithme à diffusion de Ricart-Agrawala est utilisé. Dans l'approche de Chang et al., les deux algorithmes sont à base de diffusion : Singhal localement et Maekawa entre les groupes. Dans cette approche, la constitution des groupes n'est pas justifiée par des considérations topologiques.

Dans [1], les auteurs proposent d'adapter le mécanisme d'exclusion mutuelle d'un système MPR (Mémoire Partagée Répartie) sur réseau local à l'architecture en Grille. Mais cette solution est basée sur un protocole centralisé d'exclusion mutuelle.

2. Algorithme de Naïmi-Tréhel

Nous avons basé nos solutions sur l'algorithme de Naïmi-Tréhel. C'est un algorithme à jeton, qui maintient une structure logique en arbre. Cet arbre est construit dynamiquement afin que sa racine soit toujours le site qui a demandé le jeton en dernier. La deuxième structure de l'algorithme est une file distribuée des sites qui attendent d'entrer en section critique. L'algorithme de Naïmi-Tréhel est décrit dans la figure 2 :

Chaque site S_i possède les variables locales suivantes :

- $self$ contient l'identifiant du site,
- $owner$ est l'identité du site qui a demandé le jeton en dernier,
- $next$ est l'identité du site a qui le jeton sera transmis à la fin de la section critique de S_i,
- $requesting$ est une variable booléenne, dont la valeur est vraie si le site a demandé le jeton et faux sinon,
- $token$ est aussi un variable booléenne, dont la valeur est vraie si le site possède le jeton.

La constante $Elected\_node$ désigne le nœud du système qui possède initialement le jeton.

Il existe deux types de messages dans cet algorithme :

- *Request* : envoyé par un site qui ne possède pas le jeton et qui désire entrer en section critique. La fonction *Receive_Request_CS(S_j)* est appelée à la réception de ce type de message en provenance de S_j.
- *Token* : représente le jeton. La fonction *Receive_Token(S_j)* est alors appelée à la réception de ce message en provenance du site S_j.

Les variables $owner$ forment un arbre dont, initialement, la racine de l'arbre est le possesseur du jeton : le site $Elected\_node$. Quand les requêtes sont envoyées, elles sont transmises à travers la chaîne des $owner$ jusqu'à la racine courante. Chaque site sur le chemin de propagation change la valeur de sa variable $owner$ pour l'identité du demandeur. L'arbre est ainsi changé dynamiquement par chaque requête. A la réception d'une demande, la racine affecte à ses variables $owner$ et $next$ la valeur de l'identifiant du demandeur, sauf si la racine n'a plus besoin du jeton ($requesting = faux$). Dans ce cas elle lui transmet le jeton directement. A la fin d'une section critique le jeton suit le lien $next$.

Un exemple de l'exécution de l'algorithme de Naïmi-Tréhel avec quatre sites est présenté dans la figure 2. Les flèches pleines représentent les liens $owner$ et celles en pointillés représentent les liens $next$. Le site qui est ombré est celui qui possède le jeton.

```
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_Token(S_j):
                        { Receive the token from node S_j }
  token ← true

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
```

FIG. 1 – *Naimi-Trehel*

Initialement (a), le site A est le $\mathtt{Elected\_node}$, il possède donc le jeton. La variable $owner$ de tous les sites pointent vers A. Dans (b), le site B demande le jeton en envoyant une requête à son $owner$ ($owner_B = A$). B devient la nouvelle racine ($owner_B = \emptyset$). Alors, A change son $next$ et son $owner$ pour pointer vers B. Dans (c), C demande le jeton à A. La requête est rédigée par A vers B qui met à jour son $next$ à C ($next_B = C$). A et B font pointer leur $owner$ vers C puisque c'est le dernier demandeur du jeton (C est la racine de l'arbre logique). Quand A relâche sa section critique, le jeton est envoyé à B ($next_A = B$). Puis lorsque B terminera sa section critique, il sera envoyé à C ($next_B = C$).

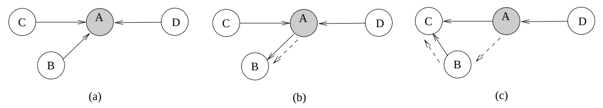

FIG. 2 – *Exemple d'exécution de l'algorithme de Naïmi-Tréhel*

3. Approche hiérarchique

Le but de notre approche est de concevoir un algorithme d'exclusion mutuelle adapté aux Grilles. Les algorithmes basés sur un arbre logique étant peu coûteux en messages (une complexité en $O(\log(n))$, ils passent relativement bien à l'échelle. Ils semblent donc adaptés à l'architecture en Grille. Mais ces algorithmes ne prennent pas en compte la dissymétrie entre les communications inter et intra-clusters. Comme les communications entre les clusters sont de moindre qualité comparées à celles à l'intérieur d'un cluster, notre adaptation de l'algorithme de Naïmi-Tréhel consiste à limiter la propagation des requêtes entre des sites de différents clusters.

Nous proposons dans un premier temps un algorithme à un niveau qui permet d'exploiter la localisation des sites en donnant un degré de priorité plus grand aux requêtes émanant du même cluster par rapport à celles distantes. Nous avons aussi modifié la phase d'initialisation de l'algorithme de Naïmi-Tréhel afin de regrouper les informations sur les requêtes venant des sites d'un même cluster.

Nous introduisons ensuite une approche à deux niveaux, qui considère que tous les messages entre des sites de clusters différents sont envoyés par l'intermédiaire de $proxy$. Cette approche est inspirée de la topologie en grille et permet une meilleure utilisation des mécanismes de priorité mais surtout de factoriser les communications inter-clusters.

3.1. Algorithme à un niveau

Le but de cet algorithme est de limiter le transfert du jeton d'un cluster vers un autre. Quand c'est possible, les requêtes du même cluster sont satisfaites avant celles issues de clusters distants. Mais pour éviter les cas de famine, nous avons introduit une borne pour limiter le nombre maximum de requêtes locales qui peuvent être satisfaites prioritairement. L'algorithme garde donc la trace du nombre de requêtes locales successives satisfaites prioritairement. Cette valeur, appelée *nb_preempt*, est communiquée d'un site à l'autre dans les messages. Tant que le nombre de préemptions est inférieur à la borne, la queue distribuée des requêtes à servir peut être modifiée pour servir d'abord les requêtes locales.

Le dernier site d'un cluster qui veut entrer en section critique est nommé la *racine locale*. Il est identifiable par le fait que sa variable $owner$ est égale à $\emptyset$. Dans la suite nous appelons site local un site appartenant à un même cluster que le site courant et site distant un site appartenant à un cluster différent.

```
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 should request 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 ← ∅

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
  else
    Send
    ⟨Preempt, nb_preempt, remote_node⟩
    to owner

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 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 owner
          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 then
      owner ← S_j
```

FIG. 3 – *Algorithme à un niveau hiérarchique - Fonctions : Initialization, Request, Release et Receive*

Nous considérons les mêmes variables que décrites dans la section 2 et nous utilisons en plus :

- $LocalCluster$: identifiant du cluster auquel appartient un site. Cette valeur est connue par tous les sites.
- $remote\_owner$: identifie le premier site distant qui recevra le jeton lorsque les requêtes locales seront servies. Le site *racine locale* ne met pas à jour sa variable $owner$ s'il reçoit une requête distante mais il stocke l'identité du site distant dans cette variable. La variable $owner$ est mise à jour avec la valeur de la variable $remote\_owner$ quand le jeton est envoyé au site distant.
- $nb\_preempt$: compteur qui désigne le nombre de requêtes locales qui sont passées prioritairement devant une requête distante. Elle ne peut excéder la valeur de la borne : $Threshold$.

Nous introduisons un troisième type de messages : $Preempt$. Ce message permet d'insérer l'identifiant du site local dans la file répartie avant un site distant. Il transporte donc l'identité du site distant préempté et la valeur de $nb\_preempt$.

Le message *Token* a été modifié afin de permettre le transport de la valeur de $nb\_preempt$ du site qui relâche le jeton. Notre algorithme est résumé dans la figure 3.1.

Phase d'initialisation.

La phase d'initialisation de l'algorithme organise l'arbre des $owners$ en fonction de la topologie du système. Pour chaque cluster sauf celui de l'$Elected\_Node$, un nœud est nommé comme $proxy$ et tous les autres membres du groupe local pointent leur $owner$ sur lui. La variable $owner$ d'un $proxy$ ainsi que les sites appartenant au cluster de l'$Elected\_Node$ pointent vers l'$Elected\_Node$.

La figure 4 montre l'avantage d'une telle initialisation. La figure 4(a) présente deux clusters, C_0 et C_1, où les nœuds A, B et C appartiennent au cluster C_0 et les nœuds D, E et F au cluster C_1. F est élu comme $proxy$ du cluster C_1. Comme A a été nommé $Elected\_Node$, il possède initialement le jeton et il est

en section critique. Dans la figure 4(b), D, qui n'appartient pas au même cluster que le possesseur du jeton, demande le jeton en envoyant une requête à F. F redirige la demande à son $owner$ A et change la valeur de sa variable $owner$ pour D. Quand la requête arrive sur le site A, A change sa variable $next$ et $remote\_owner$ avec la valeur D.
E demande à son tour le jeton. Le site F, le $proxy$ de C_1, redirige localement cette requête vers D. D met à jour ses variables $next$ et $owner$ à E. F change aussi son $owner$ pour E. Ce scénario montre que le changement de l'initialisation permet que la requête de E ne soit pas envoyée au cluster distant C_0 contrairement à l'algorithme de Naïmi-Tréhel original.

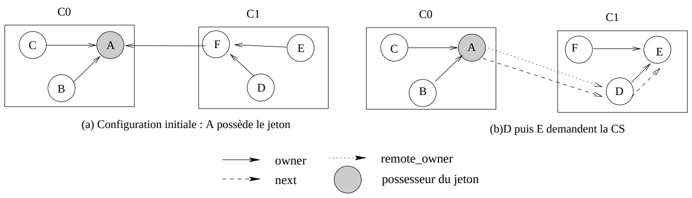

FIG. 4 – *Exemple de scénario de l'algorithme à un niveau*

Cœur de l'algorithme

De façon similaire à l'algorithme de Naïmi-Tréhel, une requête d'entrer en section critique suit le chemin des $owner$ jusqu'à ce qu'elle soit reçue par la racine locale de son cluster (le dernier site du cluster à avoir demandé la section critique).
Quand un site S_i reçoit une requête de S_j, si le récepteur S_i n'est pas la racine locale ($owner \neq \emptyset$), il transmet la requête au site pointé par sa variable $owner$. Si S_j et S_i sont dans le même cluster, S_j est stockée dans la variable $owner$ de S_i. Si S_i est la racine locale ($owner = \emptyset$) et qu'il attend d'entrer en section critique ($requesting = true$), nous distinguons deux cas :

- sur S_i la variable $next = \emptyset$, c'est-à-dire que la requête de S_j est la première que S_i reçoit depuis qu'il a demandé la section critique. Dans ce cas, le $next$ de S_i est changé pour S_j. En outre, si le demandeur est dans le même cluster, S_i change sa variable $owner$ pour S_j. Dans le cas contraire il met S_j dans sa variable $remote\_owner$.
- sur S_i la variable $next$ possède déjà une valeur. Comme le récepteur est la racine locale alors inévitablement le $next$ de S_i est un site distant. Dans ce cas, si S_j est dans le même cluster et si le nombre de préemptions est inférieur à la borne prédéfinie alors une préemption est réalisée : la requête de S_j sera satisfaite avant la requête du site distant. S_i incrémente donc son $nb\_preempt$ et envoie ses variables $nb\_preempt$ et $next$ à S_j dans un message $Preempt$. Le $next$ de S_i pointe vers S_j. S_j devient la nouvelle racine locale du cluster. Lorsque S_j reçoit le message $Preempt$, S_j affecte à son $next$ l'ancienne valeur du $next$ de S_i et met à jour sa variable $nb\_preempt$.

3.2. Algorithme à deux niveaux

Nous proposons un deuxième algorithme inspiré du premier auquel nous ajoutons un deuxième niveau constitué par les $proxys$ de chaque cluster.
Dans chaque cluster C_i, un site, nommé $proxy$, est désigné pour avoir un deuxième rôle : il centralise toutes les requêtes issues de son cluster C_i et reçoit toutes les requêtes issues des clusters distants. Par conséquent, tous les messages qui sont envoyés du site S_j au site S_i de deux clusters différents sont envoyés au $proxy$ de S_j, qui le transfère à celui de S_i. Ce dernier envoie finalement le message à son destinataire S_i. De ce fait les deux $proxys$ peuvent collecter des informations sur le transfert du jeton et les requêtes en cours au niveau cluster. A partir de ces information, il peut prendre des décisions et factoriser ainsi les communications globales.
Nous gardons les variables introduites dans la section 3.1 pour l'algorithme à un niveau, sauf que la variable $remote\_owner$ n'existe que dans les sites qui sont $proxy$. Chaque $proxy$ utilise les variables suivantes :

- $local\_owner$: variable qui mémorise le dernier site de C_i qui a demandé le jeton.
- $remote\_next$: identité du site distant qui recevra le jeton quand le jeton sera relâché par le cluster.

– L_Queue : file qui rassemble les requêtes en attente issues du cluster local. Cette file est nécessaire car le proxy ne peut envoyer qu'une requête à un site distant à un moment donné. Les requêtes en attente ne seront transmises que lorsque la précédente aura été satisfaite. Si le proxy envoyait une nouvelle requête à un site distant, cela induirait un cycle dans la chaîne des demandes (next) et dans l'arbre des last. En conséquence, les requêtes suivantes ne pourraient pas aboutir.

Un nouveau type de messages, Stock, est défini pour ajouter un site dans la file L_Queue. Il est envoyé au proxy local quand une requête doit être envoyée à un site distant. Quand un message est reçu, le proxy le traite en appelant la fonction Receive_Stock(S_j).
La fonction *Send*(S_i), qui permet d'envoyer un message au processus S_i, a été redéfinie. Si le destinataire est un site d'un cluster distant alors la fonction envoie son message au proxy local. Si l'émetteur est le proxy et le destinataire est distant alors *Send* envoie le message au proxy du destinataire. En revanche, si le destinataire est local alors le message est envoyé directement. Le code de cette fonction est :

```
Send ⟨Type, ...⟩ to dest:
  if dest ∈ LocalCluster then
    Send ⟨Type, ...⟩ to dest
  else
    if self = proxy then
      Send ⟨Type, ...⟩ to ProxyOf(dest)
    else
      Send ⟨Type, ...⟩ to proxy
```

La fonction *ProxyOf*(S_i) retourne l'identité du proxy auquel le site S_i appartient. La figure 3.2 présente les fonctions de bases de notre algorithme et la figure 3.2 les fonctions exécutées par les proxys.

```
Every node S_i :

Initialization:
  Does not change from one-level hierarchical algorithm
  if self = Proxy_i then
    remote_next ← ∅
    if Elected_node ∈ LocalCluster then
      local_owner ← Elected_node
      remote_owner ← ∅
    else
      local_owner ← ∅
      remote_owner ← ProxyOf(Elected_node)
    L_Queue ← ∅

Request_CS:
  Does not change from one-level hierarchical algorithm

Release_CS:
  requesting ← false
  if next ≠ ∅ then
    Send ⟨Token, nb_preempt⟩ to next
    if owner = ∅ then
      owner ← Proxy_i
                        { local_root : next points to a remote node }
    token ← false
    next ← ∅

Receive_Token(nb_preempt_j):
  Does not change
  from one-level hierarchical algorithm

Receive_Preempt(nb_preempt_j, node):
  Does not change
  from one-level hierarchical algorithm

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 then
          owner ← S_j
      else
        if nb_preempt < Threshold then
                        { Local preemption of the token by the sender }
          nb_preempt ← nb_preempt + 1
          owner ← S_j
          Send ⟨Preempt, next, nb_preempt⟩ to owner
          next ← S_j
        else
          Send ⟨Stock, S_j⟩ to Proxy_i
          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 then
      owner ← S_j
```

FIG. 5 – *Algorithme à deux-niveaux hiérarchiques   Fonctions : Initialization, Request et Release*

3.3. Phase d'initialisation

La phase d'initialisation est similaire à celle de l'algorithme précédent, sauf pour les proxys : leur remote_owner pointe vers le proxy de l'Elected_Node, si ce dernier est distant. La variable local_owner du proxy du site possesseur initial du jeton pointe vers l'Elected_Node.

3.4. Corps de l'algorithme

Nous distinguons dans cette présentation le comportement des sites en fonction du fait qu'il soit ou non proxy.

Site non proxy :

```
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)
```

FIG. 6 – *Algorithme à deux-niveaux hiérarchiques (cont.)* *Fonctions du niveau* proxy

La fonction Request_CS est la même que celle décrite dans la section 3.1.
Contrairement à l'algorithme précédent, dans la fonction Release_CS, la racine locale (local_root) S_i n'a pas besoin de changer sa valeur de owner avec la variable remote_owner quand il envoie le jeton à un site distant. Il met juste sa valeur de owner à proxy quand il sait que le futur possesseur du jeton est distant.
Le cœur de la fonction Receive_Request est peu modifié. Dans notre précédent algorithme, s'il y a une requête distante qui attend d'être satisfaite, la racine locale de C_i transmet la nouvelle requête locale au site distant si une nouvelle préemption ne peut avoir lieu. Dans ce nouvel algorithme, la racine locale transmet directement la requête à son proxy avec un message de type Stock.
Site proxy **:**
Dans le rôle de proxy les variables remote_owner et remote_next ont la même fonction que les variables owner et next mais au niveau proxy.
Un proxy reçoit des messages Request et Token à la fois des sites locaux et des sites distants, ainsi que les messages Stock des sites de son cluster. Les fonctions Proxy_Receive_Request_CS, Proxy_Receive_Token et Proxy_Receive_Stock sont respectivement appelées à la réception de ces messages.
La fonction Proxy_Receive_Request_CS distingue son comportement en fonction de la localisation de l'émetteur S_j du message :

– La requête vient d'un site local. Si le jeton n'est pas déjà demandé par un site du cluster (local_owner $\neq \emptyset$), le proxy enregistre S_j à la fin de la file L_Queue pour savoir à quel site il devra transférer le jeton quand il le recevra d'un autre cluster. Puis il transfère la requête à son remote_owner. Dans le cas où le jeton est déjà demandé par un site du cluster C_i (local_owner $\neq \emptyset$), alors la requête est simplement redirigée vers le local_owner.
– La requête provient d'un site distant. Dans le cas où local_owner $\neq \emptyset$ (aucun site du cluster local ne demande actuellement le jeton) ou si remote_next $\neq \emptyset$ (une autre requête distante existe) le proxy redirige la requête directement au remote_owner sans avoir besoin de la transférer à l'intérieur du cluster. Dans le cas où remote_next $= \emptyset$, la requête est rédigée au local_owner. La variable remote_next est mise à jour avec l'identité de l'émetteur. Dans les deux cas, la valeur de remote_owner prend l'identité de l'émetteur.

Le comportement de la fonction Proxy_Receive_Token dépend aussi de la provenance du message. Si l'émetteur est local alors le jeton est envoyé au site remote_next. De plus si la file L_Queue n'est pas vide, le proxy génère une nouvelle requête qui correspond à celle du site en tête de la file. Si le jeton est envoyé par un site distant alors il est transféré au site en tête de la file L_Queue. Ce dernier est alors

retiré de la file.
La fonction Proxy_Receive_Stock est toujours déclenchée par une requête locale qui doit être transférée plus tard à un site distant. La requête est donc placée à la fin de la file L_Queue. La requête ne peut être immédiatement communiquée car une autre requête du même cluster est en cours et cela engendrerait un cycle dans la chaîne des next.

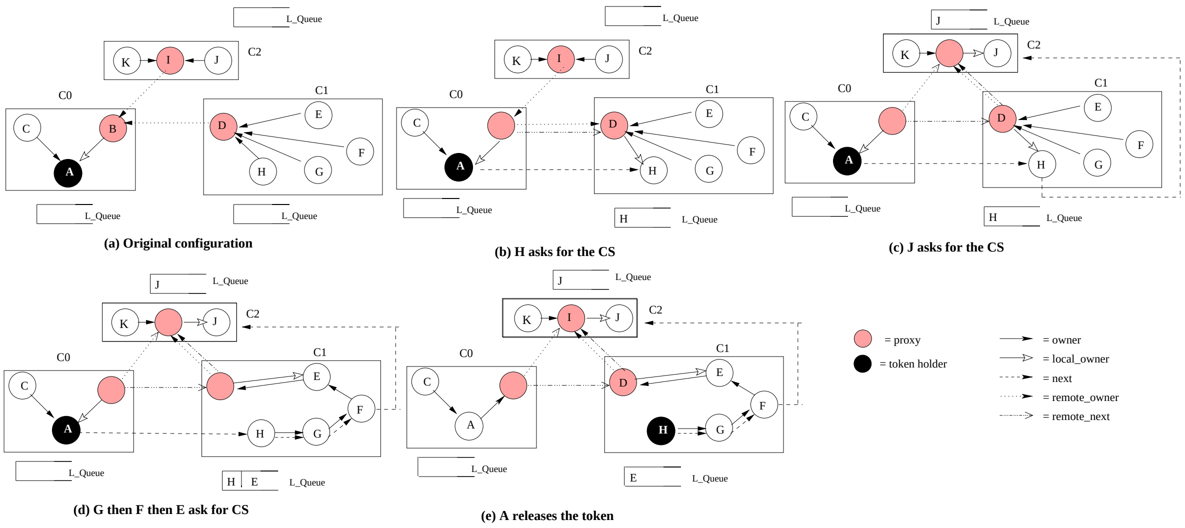

FIG. 7 – *Exemple de scénario de l'algorithme à deux niveaux*

La figure 7 montre un exemple d'exécution de l'algorithme à deux niveaux. Nous considérons que la borne sur le nombre de préemptions (Threshold) est de deux. La figure 7(a) présente trois clusters C_0, C_1 et C_2 où les sites A, B et C appartiennent au cluster C_0, les sites D, E, F, G et H au cluster C_1 et K, I et J au cluster C_2. Les sites B, D et I sont les proxys respectivement des clusters C_0, C_1 et C_2. Le jeton est initialement détenu par le site A qui est donc l'Elected_Node. La variable remote_owner des sites D et I pointe sur B (le proxy du cluster contenant l'Elected_Node). Le local_owner du proxy B pointe sur l'Elected_Node. Toutes les files des requêtes locales L_Queue sont vides.
Dans la figure 7(b), le site H du cluster C_1 demande le jeton et donc envoie une requête à son owner ($owner_H = D$ son proxy). Comme la file L_Queue de D est vide, D transfère la requête à B ($remote\_owner_D = B$). D ajoute aussi la requête de H dans sa file L_Queue et change la valeur de local_owner pour H. Comme le jeton a déjà été demandé par C_0 ($local\_owner_B = A$ et $remote\_owner = \emptyset$), B envoie la requête à A et change ses variables remote_owner et remote_next pour D (le proxy de H). Quand le site A reçoit la requête, il met son next à H (l'émetteur de la requête).
La figure 7(c) montre la demande d'entrée en section critique du site J du cluster C_2. J envoie sa requête à son owner ($owner_J = I$). Comme dans la figure 7(b), le site I transmet la requête à B ($remote\_owner_I = B$), l'ajoute à sa file L_Queue et fixe son local_owner à J. A la réception de la requête, B transmet directement à son remote_next puisqu'il est différent de $\emptyset$ ($remote\_next_B = D$). B met à jour sa variable remote_owner à I. Quand cette requête arrive à D (le proxy du cluster de H), D enregistre I dans ses variables remote_owner et remote_next. Il envoie alors la requête à H. A sa réception H fixe next à J.
Dans la figure 7(d), les sites G, F et E demandent l'accès à la section critique dans cet ordre. Les requêtes de G et F seront satisfaites avant la requête distante de J puisque le nombre de préemptions (nb_preempt) dans le cluster C_1 est inférieur à la borne de préemption ($Threshold = 2$). Cependant la requête de E ne peut être satisfaite avant la requête de J. Donc à la réception de cette requête, F (le local_root courant de C_1) envoie un message Stock à son proxy D et change sa variable owner pour E. Notons qu'à ce moment $next_F = J$. En recevant le message Stock de F, le site D ne transmet pas la requête à un site distant car la précédente requête de son cluster (celle du site H) n'a pas été encore satisfaite. D sauve E dans sa file et met à jour local_owner à E.
Quand le site A relâche la section critique, figure 7(e), il envoie le jeton à H ($next_A = H$) via les proxys respectifs. Il change sa variable owner pour B, le proxy de son cluster puisqu'il sait que le possesseur du

jeton est un site distant. Quand le $proxy$ B reçoit le message $Token$, il le transmet à son $remote\_owner$ D. Il met son $remote\_owner$ et son $local\_owner$ à $\emptyset$ car la file de B est vide. Quand le jeton arrive à D, il le transmet au site H, le site en tête de sa file et le supprime de sa file. A chaque fin de section critique, le jeton suit le chemin designé par les $next$ (G, F et J).

4. Expérimentations

Cette section décrit un ensemble d'expérimentations que nous avons réalisées afin de comparer l'efficacité de nos algorithmes d'exclusion mutuelle avec l'algorithme original de Naïmi-Tréhel. Les trois algorithmes que nous considérons sont :

– *Naïmi-Tréhel*, qui implémente l'algorithme proposé par Naïmi et Tréhel et présenté dans la section 2.
– *un niveau* est l'algorithme que nous avons présenté dans la section 3.1
– *deux niveaux* est l'algorithme basé sur les $proxys$ présenté dans la section 3.2.

4.1. Environnement

Les expérimentations présentées dans cette section sont réalisées sur un cluster dédié de 16 Pentiums IV cadencés à 2.66 GHz reliés entre eux par un réseau Ethernet à 1 Gbits/s. Les algorithmes sont implémentés en Java (JVM 1.4 de SUN) au dessus d'un noyau Linux 2.4..

Pour émuler un environnement de Grilles avec des différentes latences entre les clusters, nous utilisons une plate-forme distribuée de test. Nous établissement un réseau virtuel en utilisant DUMMYNET [12] et IPNAT. Ce dernier est un outil d'IP Masquerade qui permet de diviser le réseau en réseaux locaux virtuels (clusters). DUMMYNET est un outils flexible conçu initialement pour tester les protocoles réseaux. Il permet de simuler une bande passante limitée, d'introduire des délais de propagation ainsi que des pertes de messages. Tous les messages échangés entre deux sites de clusters différents passent par le serveur DUMMYNET.

Sur chaque ordinateur, nous lançons trois sites virtuels, permettant d'émuler une plate-forme de 48 sites, répartis en 3 clusters de 16 sites. Chaque site virtuel d'une même machine appartient à des clusters différents, de cette façon les communications entre les sites virtuels d'un même site ne sont pas locales mais passent par le routeur DUMMYNET. Dans le reste de cette section, nous appelons *site* un site virtuel et non un ordinateur.

La topologie de la plate-forme est préalablement connue de tous les sites ainsi que l'identité du possesseur initial du jeton ($Elected\_Node$). Dans chaque expérimentation, tous les sites demandent à accéder à 10 sections critiques successives.

Les expérimentations sont caractérisées par :

– α : le temps pris par un site pour exécuter une section critique,
– β : c'est le délai inter-section critique qui est le temps entre la fin d'une section critique et le moment où le même site en demande une nouvelle,
– ϵ : la borne de préemption (uniquement pour les algorithmes *un niveau* et *deux niveaux*),
– γ: délai introduit grâce à DUMMYNET dans les communications entre les clusters.

Chacune des expérimentations est évaluée à l'aide des métriques suivantes :

– **nombre de messages**, divisés en deux catégories : *messages locaux*, échangés par deux sites appartenant au même cluster et *messages globaux*, échangés par deux sites de clusters différents. Le ratio du nombre de messages locaux sur le nombre de messages globaux est aussi calculé.
– **temps d'obtention** : temps entre le moment où un site demande la section critique et le moment où il y entre. Nous calculons la moyenne et l'écart type du temps d'obtention.
– **nombre de préemptions** : nombre de préemptions effectives pendant l'expérimentation.

4.2. Influence du comportement de l'application

Le but de ces expérimentations est d'observer le comportement de chaque algorithme quand β et α varient. Nous appelons "comportement de l'application" le ratio β/α, qui exprime la fréquence avec laquelle la section critique est demandée. Dans ces expérimentations, le délai entre clusters γ est fixé à 100 ms. La figure 8 résume les mesures de performance en fonction du ratio β/α. Pour toutes les mesures sauf la dernière, le temps en section critique, α, est fixé à 0.5 s, alors que l'intervalle entre deux demandes successives par un site, β, varie. Les différentes configurations sont : 10/0.5, 5/0.5, 2/0.5, 0.5/0.5 et 0.5/1, où α et β sont exprimés en secondes.

Algorithmes	Ratio	Temps d'obtention (sec.)		nombre de	Nb de messages		
	β/α	moyenne	écart-type	préemptions	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
un niveau	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
deux niveaux	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

FIG. 8 – *Influence du comportement de l'application*

Pour tous les algorithmes, quand le ratio β/α diminue, le temps d'obtention augmente. Ceci est facilement explicable, étant donné que la probabilité que d'autres sites aient aussi demandé d'accéder à la section critique augmente aussi. Dans le cas de nos algorithmes, quand le ratio β/α est égal à 20, le mécanisme de préemption est rarement utilisé (une ou deux préemptions par expérimentation), puisqu'il n'y a pas beaucoup de demandes simultanées dans un cluster. Quand le ratio β/α est inférieur ou égal à 10, le mécanisme devient effectif. Dans le même temps, nous observons que l'écart type augmente et que les messages se concentrent dans les clusters. Pour un ratio variant de 10 à 1, le comportement des algorithmes ne changent pas beaucoup car ces ratios sont relativement petits : quand un site demande à entrer en section critique, presque tous les autres sites du système le demandent aussi. Dans ces cas, seul le temps de transmission du jeton peut être réduit, donc la variation du temps d'obtention n'est pas très significatif. Quand le ratio β/α est plus petit que 1, le temps d'obtention et l'écart type doublent (ou presque) car les sites restent deux fois plus longtemps en section critique.

4.3. Influence de la préemption

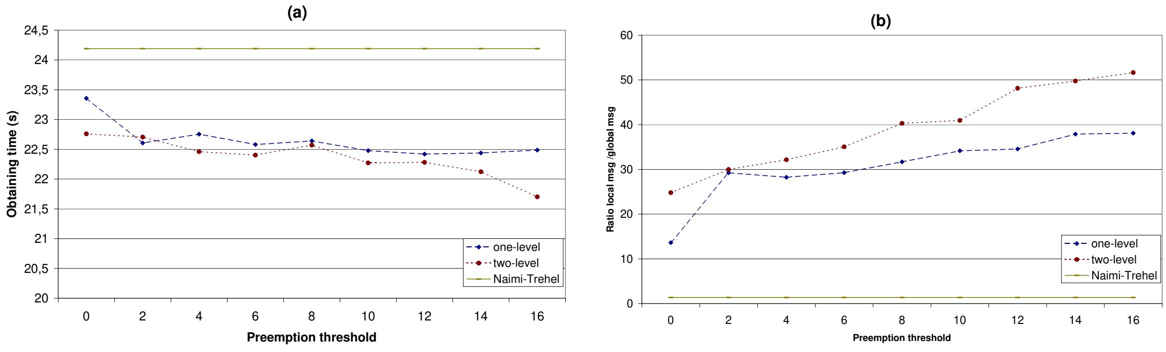

FIG. 9 – *Influence de la préemption*

Ces expérimentations permettent d'évaluer l'influence du nombre maximum de préemptions (la borne ϵ) sur le comportement de nos algorithmes. Nous avons fixé : $\alpha = \beta = 500$ ms et $\gamma = 100$ ms. La figure 9.(a) présente le *temps d'obtention* moyen et la figure 9.(b) le ratio *messages locaux / messages globaux* quand la borne sur le nombre de préemption augmente. La figure 10 résume ces expérimentations.

Le premier commentaire est que même quand la borne de préemption ϵ est égale à 0, les temps d'obtention de nos algorithmes sont plus courts que celui de Naïmi-Tréhel. Cet écart est dû à l'introduction du

Type	Borne de préemptions	Temps d'obtention (s)		nombre de préemption	Nb de messages		
		moyenne	écart-type		local	global	%
un niveau	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
deux niveaux	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

FIG. 10 – *Résumé sur l'influence de la préemption*

concept de racine locale, que nous avons introduit dans nos algorithmes. Il permet d'informer un site qu'un autre site du même cluster a déjà demandé la section critique, évitant donc une requête distante. Les transmissions du jeton en local sont déjà ainsi favorisées.

Naturellement, quand ϵ augmente, le nombre de préemptions augmente aussi. Le *temps d'obtention*, lui, diminue et *l'écart type* augmente significativement. Le nombre de transmissions du jeton entre les clusters diminue, permettant un gain en termes de temps d'obtention. L'écart type augmente puisque la préemption accélère les requêtes locales au détriment des distantes. Nos algorithmes relâchent l'équité entre les temps d'obtention des sites mais conservent une équité au niveau des clusters.

Nous pouvons aussi observer que le mécanisme de préemption est plus efficace avec l'algorithme à *deux niveaux* qu'avec celui à *un niveau*. La raison est sans doute que seule la première requête est transmise à l'intérieur du cluster dans l'algorithme à deux niveaux. Les requêtes distantes suivantes sont directement transmises au nœud désigné par le $\mathtt{remote\_owner}$ du $\mathtt{proxy}$. En conséquence, une requête distante est moins redirigée et arrive donc plus rapidement à la racine locale du cluster final. Cela augmente donc la probabilité que cette requête soit préemptée par une requête locale.

La dernière remarque est la très forte concentration des messages dans les clusters que permettent nos algorithmes. Avec l'algorithme de Naïmi-Tréhel, il y a $1,36$ messages locaux pour un message global, ce ratio passe à $38,12$ pour l'algorithme à *un niveau* et à $51,65$ pour celui à *deux niveaux*.

4.4. Influence de la plate-forme

La figure 11 illustre l'évolution du *temps d'obtention* moyen quand le délai introduit dans les communications entre les clusters augmente. Pour ces expérimentations, nous avons $\alpha = \beta = 500\,\mathrm{ms}$ et $\epsilon = 8$. Le délai inter-cluster (γ) varie de $0\,\mathrm{ms}$ à $200\,\mathrm{ms}$. $0\,\mathrm{ms}$ signifie qu'aucun ralentissement supplémentaire n'est introduit par le routeur.

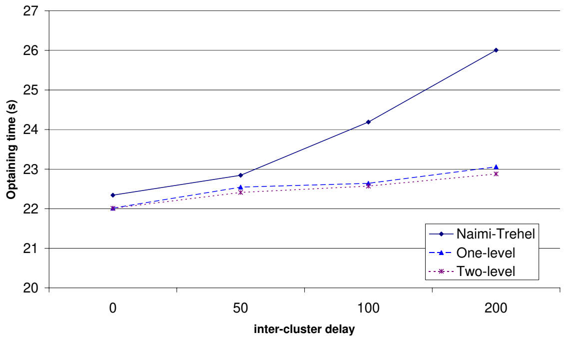

FIG. 11 – *Évolution de la préemption*

Les résultats montrent que nos algorithmes passent mieux à l'échelle que l'algorithme de Naïmi-Tréhel et sont moins sensibles à l'éloignement des sites. Nous observons que l'écart entre le temps d'obtention de nos algorithmes et celui de Naïmi-Tréhel augmente avec le délai inter-cluster. Ceci s'explique car nos algorithmes concentrent les communications dans les clusters, le délai entre les clusters a donc moins d'influence.

5. Conclusion

Nous avons présenté deux nouveaux algorithmes d'exclusion mutuelle optimisés pour les environnements de type Grilles. L'idée principale de cette approche est d'adapter l'algorithme de Naïmi-Tréhel à la topologie du réseau en minimisant le nombre de messages inter-clusters. Les deux algorithmes proposés utilisent le même mécanisme de racine locale et de préemption locale mais le deuxième utilise en plus un mécanisme de `proxy`.

Les résultats des expérimentations présentés dans la section 4 permettent de conclure que nos algorithmes réduisent le *temps d'obtention* par rapport à l'algorithme de Naïmi-Tréhel, mais aussi concentrent les communications au niveau local. De nos deux algorithmes, celui utilisant les `proxys` concentre encore plus les communications locales. L'utilisation de nos algorithmes introduit une perte d'équité dans le traitement des différentes requêtes. Cette perte est due à la priorité donnée aux requêtes locales par rapport aux requêtes distantes. Cependant, nos algorithmes conservent une équité dans le traitement des différents clusters et évitent les famines grâce à la définition d'un seuil maximal de préemption.

Bibliographie

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, May 2003.
2. 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.
3. 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.
4. 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.
5. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–564, July 1978.
6. M. Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, May 1985.
7. 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.
8. 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.
9. 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.
10. K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems (TOCS)*, 7(1):61–77, 1989.
11. G. Ricart and A. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM (CACM)*, 24, 1981.
12. L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, 1997.
13. M. Singhal. A dynamic information structure for mutual exclusion algorithm for distributed systems. *IEEE Transactions on Parallel Distributed Systems*, 3(1):121–125, 1992.

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