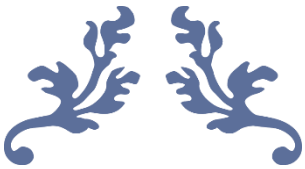


Algorithme de Raymond et Algorithme de Naimi-Tréhel

Parallélisme et Algorithmique Répartie

DATA Engineer

Réalisé par :

JEHBALI Youssef

Encadre par :

Pr. NAJA Najib

Année universitaire : 2021-2022

Résumé

L'algorithme d'exclusion mutuelle à priorité distribuée permet de définir l'ordre d'accès aux parties critiques des ressources partagées protégées. Ces algorithmes sont utiles lorsqu'ils sont appliqués en temps réel ou pour assurer différents niveaux de qualité de service avec une priorité moindre. Pour pallier ce problème, certains algorithmes proposent d'augmenter progressivement la priorité des requêtes en attente, mais cela peut conduire à une violation de l'ordre de priorité. Par conséquent, pour minimiser ces violations sans provoquer de famine et de surcharge de messages, nous proposons de le modifier pour ralentir la fréquence d'incrémentation de priorité. Notre évaluation de la performance confirme l'efficacité de notre approche. Ce rapport détaille les trois algorithmes d'exclusion mutuelle classique. Ces algorithmes sont l'algorithme de Raymond [Ray89b] et l'algorithme de Naimi-Tréhel [NT87a,NT87b]. Ces algorithmes sont des algorithmes à jeton circulant dans un arbre statique pour Raymond et une forêt d'arbres dynamiques pour Naimi-Tréhel. Le choix de ces algorithmes s'explique par leurs bonnes performances en termes de complexité en messages qui est en moyenne logarithmique.

Mots-clés : Algorithmique distribuée, Exclusion mutuelle distribuée, Priorités

Abstract

The distributed priority mutual exclusion algorithm allows to define the order of access to the critical parts of the protected shared resources. These algorithms are useful when applied in real time or to ensure different levels of quality of service with lower priority. To overcome this problem, some algorithms propose to gradually increase the priority of pending requests, but this can lead to a violation of the order of precedence. Therefore, to minimize these violations without causing starvation and message overload, we propose to modify it to slow down the frequency of priority increment. Our performance evaluation confirms the effectiveness of our approach.This report details the three classic mutual exclusion algorithms. These algorithms are the Raymond algorithm [Ray89b] and the Naimi-Tréhel algorithm [NT87a, NT87b]. These algorithms are token algorithms circulating in a static tree for Raymond and a forest of dynamic trees for Naimi-Tréhel. The choice of these algorithms is explained by their good performance in terms of message complexity which is on average logarithmic.

Sommaire

I. Introduction

L'exclusion mutuelle oriental l'un incontinent paradigmes fondamentaux incontinent systèmes distribués à assurer incontinent ouverture cohérents aux ressources partagées Elle confiant qu'au encore seul mécanisme peut accomplir une fraction pour règlement manipulant une subsistance partagée appelée intersection dramatique (propriété pour sûreté) et pourquoi toute réclamation d'accès de la intersection dramatique sera satisfaite sur seul moment terminé (propriété pour vivacité) Moult algorithmes d'exclusion mutuelle existent sur la poésie Ils peuvent demeurer divisés de accouplement catégories les algorithmes de permissions et les algorithmes de paiement lequel nous passionné (**Raymond,Naimi-Trehel**) Sur la première classe seul mécanisme peut rentrer de intersection dramatique avant détenir quitus la autorisation pour l'ensemble incontinent autres mécanisme (ou d'un sous-ensemble) Sur la instant classe seul exceptionnel paiement oriental communion avec les mécanisme Le faisandé pour disposer ce paiement distribution le privilège unique d'entrer de intersection dramatique Sur la généralité incontinent algorithmes les requêtes sont satisfaites suivant une fédéralisme du "premier parvenu premier-né servi" facilité de une clepsydre rationnel dans les requêtes soit vraiment facilité de l'horloge biophysique due possesseur du paiement (ou due avenir détenteur) Toutefois cette approchant n'est rien adaptée quand l'on compte administrer incontinent mécanisme sans incontinent priorités différentes contrairement sur les applications temps-réel soit sur les systèmes basés dans incontinent niveaux pour perfection pour disposition À obvier ce souci incontinent auteurs ont proposé incontinent algorithmes distribués d'exclusion mutuelle (généralement une mouture modifiée pour ceux mentionnés ci-avant) soit chacun réclamation oriental associée de seul échelon pour primauté Toutefois l'ordre pour primauté peut occasionner incontinent famines i e incontinent moment infinis a qu'un mécanisme obtienne la intersection dramatique violant pareillement la habitation pour crudité La disette apparaît lorsqu'un mécanisme pour haute primauté peut contraindre de constance les autres mécanisme pour encore basses priorités d'exécuter la intersection dramatique Pareillement à prévenir cela les requêtes de basses priorités peuvent demeurer dynamiquement augmentés à arriver incidemment la primauté maximale Pourtant cette tactique donné seul désavantage De conséquence l'ordre incontinent priorités incontinent requêtes n'est encore honorable i e incontinent inversions pour primauté peuvent réapparaître si une réclamation sans une primauté originale guitare oriental satisfaite préalablement une dissemblable réclamation pendante pour encore haute primauté Nous proposons pareillement sur cet paragraphe seul nouvel calcul d'exclusion mutuelle distribué de primauté lequel minimise les violations pour primauté dépourvu incorporer pour disette Ces travaux se basent dans l'algorithme pour **Kanrar-Chaki** lequel utilise seul arbuste rationnel stationnaire à refaire rouler seul paiement Les mécanismes proposés permettent pour freiner la multiplicité incontinent incrémentations pour primauté incontinent requêtes pendantes et dans conséquence pour diminuer le effectif pour violations pour primauté Ces améliorations se font dépourvu messages supplémentaires et ne dégradent rien les moment pour explication.

II. Algorithme de Raymond

Avant d'attaquer cet algorithme il nous faut savoir ce qu'est une **Approche basée sur une structure logique statique**

La donnée approchant énergumène seul aspects formel les mécanismes du principe sont structurés sur structure d'une conformation rationnel stationnaire (une mathématique pour canalisation pour communication) dans dont toute réclamation est propagée de biais les mécanisme demandeurs et ce jusqu'à arriver le mécanisme propriétaire du paiement.

Nous pouvons distinguer dans la littérature trois types de topologies statiques :

- ***Anneau :*** le jeton circule le long d'un anneau unidirectionnel.
- ***Arbre** (qui nous intéresse) :* Le nœud racine de l'arbre est le site qui détient le jeton. Les liens sont orientés vers la racine de manière que lorsqu'un site demande le jeton, cette demande soit propagée jusqu'à la racine. Ainsi, un lien entre deux nœuds indiquera toujours la direction de la racine, i.e., du site possédant le jeton. La complexité moyenne de ces algorithmes est de O(logN). Dans cette catégorie nous pouvons citer les algorithmes de ***Raymond***.
- ***Graphe :*** Tous les nœuds sont placés dans une topologie arbitraire

1. Définition & Principe

Dans cet algorithme, les processus sont structurés sous forme d'un arbre : un processus communique uniquement avec ses voisins.
Chaque processus *Pi* maintient une variable *présenti* indiquant l'identité d'un processus voisin dans le chemin qui mène au processus détenteur du jeton (*présenti = Pi* implique que le processus *Pi* possède le jeton). En outre, le processus *Pi* gère une file d'attente notée *Qi* qui consiste à mémoriser les processus demandeurs de la section critique.

L'algorithme de Raymond se comporte ainsi : lorsqu'un processus *Pi* désire entrer en secton critique, il met sa demande dans la file selon la politique FIFO et envoie ensuite un message ***Requête*** au processus *présenti*.

Lorsqu'un processus *Pj* reçoit un message ***Requête*** provenant d'un de ses voisins, il rajoute la requête dans sa file *Qj* et achemine le même message Requête (celui du processus *Pi*) au processus *présentj* . A son tour, son voisin procède de la même manière et ce jusqu'à ce que la requête de *Pi* atteigne le processus qui détient le jeton. Une fois que ce dernier l'a bien reçu, il renvoie un message ***Jeton*** le long du même chemin, emprunté par la requête de *Pi* , mais dans le sens inverse.

Lorsqu'un processus *Pk* reçoit un message ***Jeton***, il envoie ce dernier au premier processus de la file *Qk* ; ce processus peut être le processus *Pk* lui-même ou un autre processus demandeur *Pj*. le processus Pk supprime ensuite de la file *Qk* la requête en question. En effet, si la première requête de la file *Qk* n'est pas celle du processus *Pk* et que la file *Qk* n'est pas vide, Pk lui envoie à nouveau un message ***Requête*** au processus *Pj* afin que le jeton lui soit retourné.

En gros ;

Organiser les processus du système en une structure d'arbre *reconfigurable*.

Si aucun processus ne veut rentrer en sc , le jeton reste sur le dernier processus qui a quitté la section critique au lieu de circuler

Le processus qui contient le jeton est la racine de l'arbre, de ce fait, les requêtes (demande de jeton) sont propagées vers la racine

Chaque processus détient une variable *possi* qui lui indique par le biais de quel processus il peut accéder au jeton.

Chaque processus gère une file d'attente contenant les demandes formulées

2. Hypothèses

- Chaque processus a un identifiant unique
- Les canaux de communication sont fiables et synchrones
- Les processus sont corrects
- Aucun mécanisme de datation n'est nécessaire

3. Algorithme

```
1  Local variables :
2  begin
3      father : site ∈ Π or nil;
4      Q : FIFO queue of sites ;
5      state ∈ {tranquil, requesting, inCS}
6  end
7  Request_CS()
8  begin
9      if father ≠ nil then
10         add self in Q;
11         if state = tranquil then
12             state ← requesting;
13             send Request to father;
14         wait(father = nil);
15     state ← inCS;
16     /* CRITICAL SECTION                    */
17 end
18 Release_CS()
19 begin
20     state ← tranquil;
21     if Q ≠ ∅ then
22         father ← dequeue(Q);
23         send Token to father;
24         if Q ≠ ∅ then
25             state ← requesting;
26             send Request to father;
27 end
```

```
28 Initialization
29 begin
30     Q ← ∅;
31     state ← tranquil;
32     father ← according to the initial topology;
33 end
34 Receive_Request() from s_j
35 begin
36     if father = nil and state = tranquil then
37         father ← s_j;
38         send Token to father;
39     else if father ≠ s_j then
40         add s_j in Q;
41         if state = tranquil then
42             state ← requesting;
43             send Request to father;
44 end
45 Receive_Token() from s_j
46 begin
47     father ← dequeue(Q);
48     if father = self then
49         father ← nil;
50         notify(father = nil);
51     else
52         send Token to father;
53         if Q ≠ ∅ then
54             state ← requesting;
55             send Request to father;
56         else
57             state ← tranquil;
58 end
```

4. Complexité

- **Pire des cas** :(Configuration la plus défavorable)

Arbre dégénéré ou filiforme

D= n-1

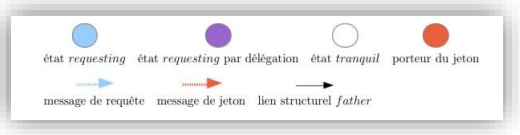
Nombre de messages **2*(n-1)**

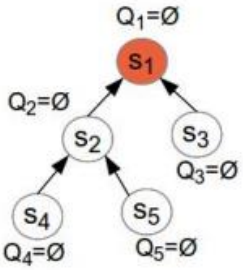
- **Cas quelconque** : obtenu par simulation

0 (log (n)) échange de messages

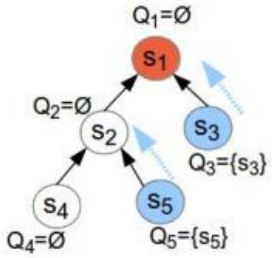
- Sa complexité a une relation logarithmique moyenne avec le nombre de nœuds N lorsque la charge est faible, et devient constante lorsque la charge augmente. Cette économie de message est possible car si la file d'attente locale d'un site contient une requête, le site ne retransmet pas le message de requête à son site parent. Un autre avantage de l'algorithme de Raymond est que des graphes de communication incomplets peuvent être pris en compte. Par conséquent, la topologie logique peut correspondre entièrement ou partiellement à la topologie physique du réseau sous-jacent.

5. Exemple

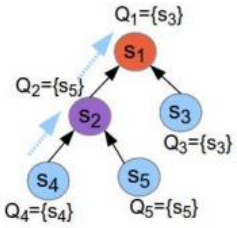




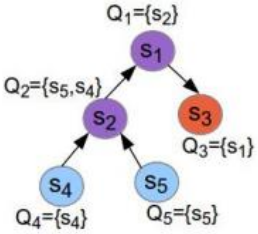
État initial : s1 possède le jeton et est en section critique.



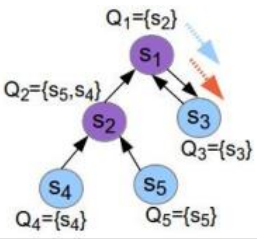
État 1 : s5 et s3 entrent en état *requesting*. Des messages de requête sont envoyés aux pères respectifs



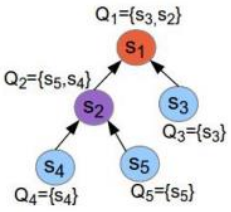
État 2 : s2 et s1 reçoivent les requêtes : ils les ajoutent dans leur file locale. s2 devient requesting pour la requête de s5 et envoie une requête à s1. De plus, s4 entre en état requesting et envoie une requête à s2.



État 5 : s3 reçoit le jeton et entre en section critique puisque s3 est le premier élément de sa file locale. s3 reçoit ensuite le message de requête et ajoute s1 dans Q3.



État 4 : s1 sort de la section critique, dépile le premier élément de sa file locale et envoie le jeton à s3. Comme Q1 n'est pas vide, s1 envoie également, un message de requête pour la requête de s



État 3 : s2 reçoit la *requête* de s4 : aucun message n'est retransmis car ceci a déjà été fait pour s5

III. Algorithme de Naimi-Tréhel

Avant d'attaquer cet algorithme il nous faut savoir ce qu'est une **Approche basée sur une structure logique dynamique**

Une approche est basée, quant à elle, sur une structure logique dynamique reconfigurable en fonction des demandes d'entrées en section critique. Contrairement à l'approche précédente, cette approche utilise une structure logique dynamique appelée arborescence. La topologie initiale sous-jacente au système de N processus est un réseau complet.

Chaque nœud maintient deux listes chaînées distribuées : next pour sauvegarder l'ordre des requêtes pendantes et f ather (ou last) qui indique le chemin vers le dernier demandeur. Ainsi le dernier demandeur est la racine d'un arbre de la forêt. Ces algorithmes ont une complexité moyenne en nombre de messages de O(logN).

Dans la suite, nous décrirons ***l'algorithme de Naimi et Tréhel***. Il utilise les deux messages suivants : *Requête* et *Jeton*.

1. Définition & Principe

Dans notre algorithme chaque processus *Pi* possède deux variables : ***lasti*** et ***nexti***.

- La première variable indique le processus auquel *Pi* doit envoyer un message ***Requête***. Initialement, cette variable est à Nil uniquement pour le processus racine, les autres processus ont leur variable qui pointe vers la racine.

- La seconde indique les processus auxquels *Pi* transmettra un message Jeton à sa sortie de la section critique. L'ensemble des variables locales ***nexti*** de tous les processus constitue une file d'attente distribuée dite file des processus demandeurs du jeton.

  En outre, tout processus *Pi* qui fait une demande d'accès à la section critique devient racine de l'arborescence.

  Un processus demandeur *Pi* ne peut accéder en section critique que s'il a obtenu le jeton. En effet, le message ***Requête*** du processus *Pi* est acheminé séquentiellement le long de l'arborescence jusqu'à ce qu'elle arrive à la racine *Pr*. Tout processus intermédiaire *Pj* (qui fait suivre la requête de *Pi* vers son dernier) met à jour sa variable ***lasti*** à *Pi* .

  Le processus racine *Pr* peut être soit le processus qui détient le jeton : il transmet le jeton directement au processus *Pi* ; ou c'est le dernier processus qui recevra le jeton dans un futur proche. Dans ce cas, il met à jour sa variable ***nextr*** à *Pi* .

a. *Version avec file distribuée [NT87a, NTA96]*

En recevant un message de requête, si la dernière variable n'est pas égale à zéro, transmettre la requête au site pointé par la dernière variable. A l'inverse, si last est égal à nil, le site récepteur est alors la racine : s'il possède le jeton mais n'est pas dans la section critique, le jeton est passé directement à l'expéditeur de la requête, sinon son prochain(***next***) pointeur est mis à jour à ce dernier. Après avoir reçu le jeton, le site passe directement à la section critique

b. *Algorithme de Naimi-Tréhel avec files locales [NT87b]*

La prochaine file d'attente distribuée est remplacée par la FIFO locale. Le site restera root tant qu'il est à l'état inCS ou à l'état demandeur. Ainsi, lorsqu'un site reçoit un message de requête et qu'il se trouve dans l'un de ces deux états, la requête est ajoutée à la FIFO locale. Lorsque le jeton est libéré, la file d'attente locale transmet le jeton au support suivant.

Dès réception, le nouveau porteur fusionne sa file d'attente locale avec la file d'attente locale du jeton. Pour respecter la propriété de vivacité, les requêtes dans la file d'attente des jetons ont priorité sur les requêtes dans la nouvelle file d'attente du support. Ainsi, le prochain ensemble de files d'attente locales forme une file d'attente distribuée virtuelle, et une file d'attente distribuée équivalente peut être trouvée dans la première version de l'algorithme.

2. Algorithme

a. *Algorithme de Naimi-Tréhel avec file distribuée [NT87a]*

```
1  Local variables :
2  begin
3  |   state ∈ {tranquil, requesting, inCS}
   |   next : site ∈ Π or nil;
4  |   last : site ∈ Π or nil;
5  end

6  Initialization
7  begin
8  |   state ← tranquil;
9  |   next ← nil;
10 |   if self = elected_node then
11 |   |   last ← nil;
12 |   else
13 |   |   last ← elected_node;
14 end

15 Request_CS()
16 begin
17 |   state ← requesting;
18 |   if last ≠ nil then
19 |   |   send Request(self) to last;
20 |   |   last ← nil;
21 |   |   wait(state = inCS);
22 |   state ← inCS;
23 |   /* CRITICAL SECTION                    */
24 end

25 Release_CS()
26 begin
27 |   state ← tranquil;
28 |   if next ≠ nil then
29 |   |   send Token to next,
30 |   |   next ← nil;
31 end

32 Receive_Request(requester : site) from s_j
33 begin
34 |   if last = nil then
35 |   |   if state ≠ tranquil then
36 |   |   |   next ← requester;
37 |   |   else
38 |   |   |   send Token to requester;
39 |   else
40 |   |   send Request(requester) to last;
41 |   last ← requester;
42 end

43 Receive_Token() from s_j
44 begin
45 |   state ← inCS;
46 |   notify(state = inCS);
47 end
```

b. Algorithme de Naimi-Tréhel avec files locales [NT87b]

```
1  Local variables :
2  begin
3      state ∈ {tranquil, requesting, inCS}
       next : FIFO queue of sites;
4      last : site ∈ Π or nil;
5  end
6  Initialization
7  begin
8      state ← tranquil;
9      next ← ∅;
10     if self = elected_node then
11         last ← nil;
12     else
13         last ← elected_node;
14 end
15 Request_CS()
16 begin
17     state ← requesting;
18     if last ≠ nil then
19         send Request(self) to last;
20         last ← nil;
21         wait(state = inCS);
22     state ← inCS;
23     /* CRITICAL SECTION                */
24 end
25 Release_CS()
26 begin
27     state ← tranquil;
28     if next ≠ ∅ then
29         last ← getLast(next);
30         site next_holder ← dequeue(next);
31         send Token(next) to next_holder;
32         next ← ∅;
33 end
34 Receive_Request(requester : site ) from s_j
35 begin
36     if last = nil then
37         if state ≠ tranquil then
38             add requester in next;
39         else
40             send Token(∅) to requester;
41             last ← requester;
42     else
43         send Request(requester) to last;
44         last ← requester;
45 end
46 Receive_Token(remote_queue : Queue ) from s_j
47 begin
48     state ← inCS;
49     next ← remote_queue + next;
50     notify(state − inCS);
51 end
```

- L'arbre des ***LAST*** : un arbre logique dynamique qui sert à acheminer les demandes d'entrée en section critique.
- La chaîne des ***NEXT*** : une file d'attente distribuée contenant l'ensemble des sites attendant l'accès à la section critique. Le site possédant le jeton constitue le "début" de cette file d'attente et en est, en absence de requête, son unique membre

3. Complexité (Avantage)

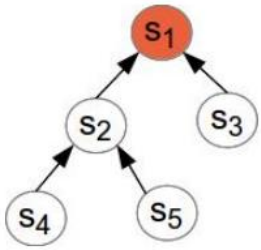
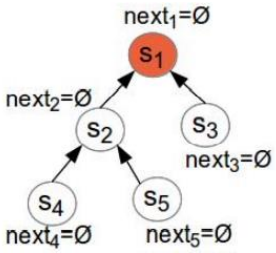
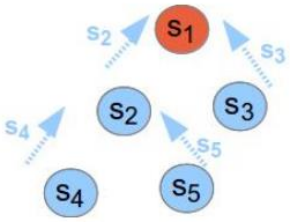
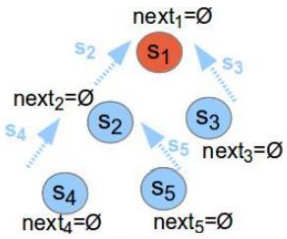
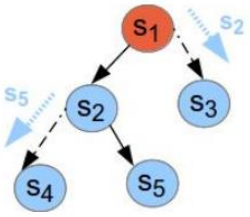
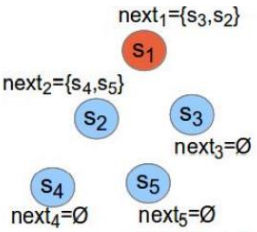
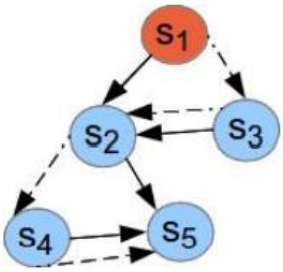
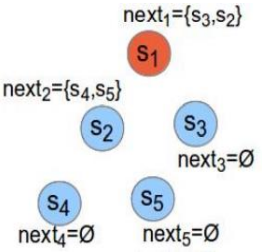
La complexité en messages de cet algorithme est en moyenne de l'ordre de log(N).

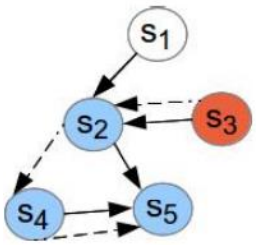
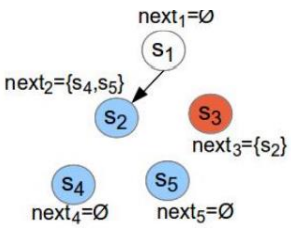
Contrairement à l'algorithme de Raymond, l'envoi du jeton donnera directement au porteur suivant l'accès à la section critique, réduisant le trafic réseau, Augmenter ainsi l'utilisation des ressources clés. De plus, les sites qui ont rarement besoin de sections critiques ne sont pas tenus de transmettre des messages de requête. Dans les deux versions, la complexité moyenne des messages est logarithme par rapport à N, mais la version avec file d'attente locale a une complexité constante à mesure que la charge augmente.

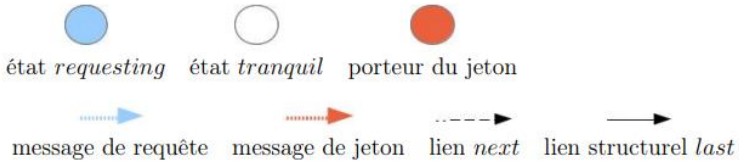
~~**NB :**~~

*Pour le problème de l'allocation dynamique de ressource, A. Bouabdallah et C. Laforest dans [BL00], ont proposé un algorithme distribué basé sur le jeton. Cet algorithme utilise entre 0 et n+3*k messages (ou k est le nombre total de ressources et n le nombre total de processus). En moyenne, O(log n) messages sont utilisés si k est constant.*

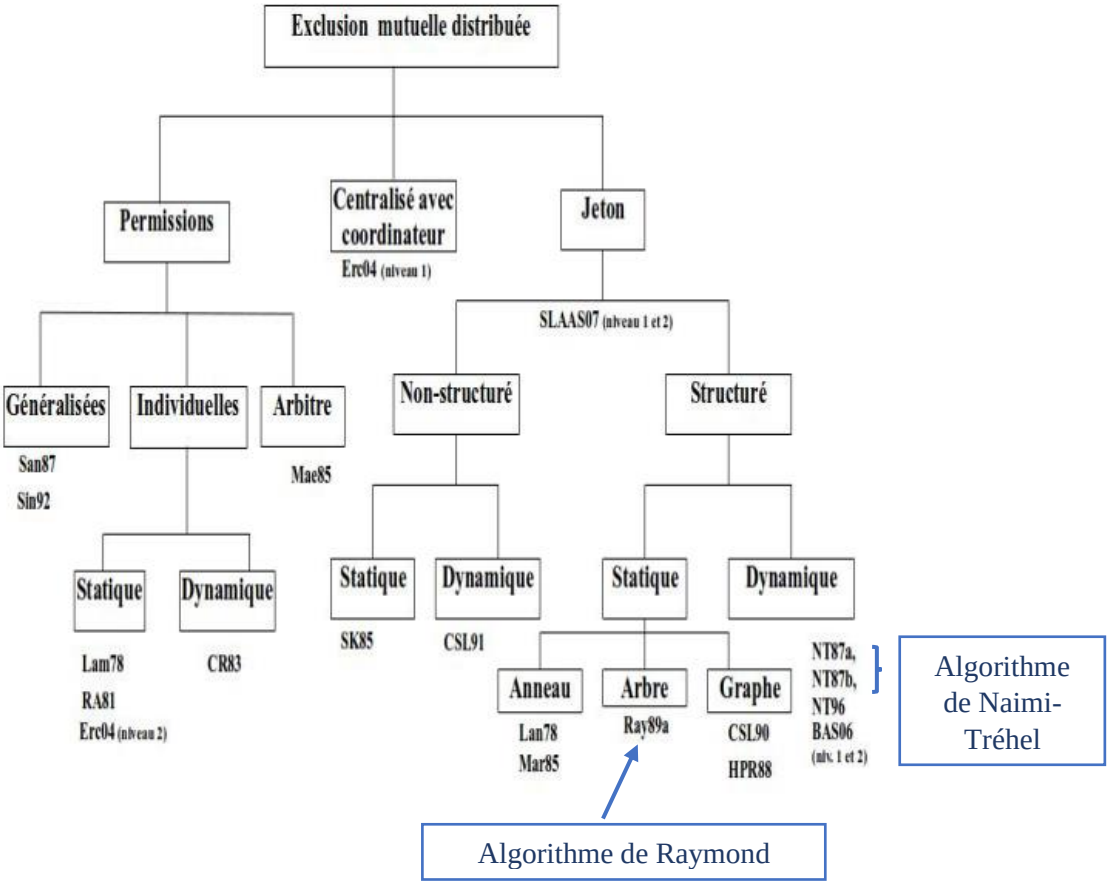
4. Exemple

Etape	File distribuée	Files locales
État initial : s1 possède le jeton et est en section critique.		
Étape 1 : s2, s3, s4, et s5 demandent la section critique et envoient un message request à leur lien last. Tous les sites sont désormais racines.		
Étape 2 : s1 reçoit la requête de s3 puis la requête de s2. s2 reçoit la requête de s4 puis la requête de s5.		
Étape 3 : Pour la version avec file distribuée, s3 (respectivement s4) reçoit la requête de s2 (resp. s5).		

Étape 4 : s1 sort de section critique et envoie le jeton à s3. s3 reçoit le jeton et entre en section critique.		



~~Schéma explicatif :~~



IV. Conclusion

Dans ce travail, nous proposons des mécanismes basés sur l'algorithme de **Raymond [Ray89b]** d'une part et l'algorithme de **Naimi-Trehel [NT87a, NT87b].** d'autre part, qui utilisent la priorité pour assurer la vivacité. Ces mécanismes peuvent réduire considérablement le nombre de violations de l'algorithme d'origine. En retardant les incréments de priorité, ils offrent un bon compromis entre famine et inversion de priorité. Les évaluations de performances confirment que la prise en compte de la localité des requêtes réduit considérablement le nombre de messages. Ils montrent également que le gain obtenu est plus important à fortes charges par rapport aux algorithmes de la littérature. Par conséquent, notre mécanisme est facilement adaptable aux applications avec des pics de charge. Pour une analyse plus approfondie, il serait intéressant d'étudier ces mécanismes car la charge change dynamiquement au cours de l'exécution de l'expérience.

V. Bibliographie

- Naimi (M.) et Trehel (M.). – An improvement of the log(n) distributed algorithm for mutual exclusion. In : ICDCS, pp. 371–377.
- Raymond (K.). – A tree-based algorithm for distributed mutual exclusion. ACM Trans. Comput. Syst., vol. 7, n1, 1989, pp. 61–77.
- Chang (Y.-I.). – Design of mutual exclusion algorithms for real-time distributed systems. J. Inf. Sci. Eng., vol. 11, n4, 1994, pp. 527–548.
- https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwiQz6aG97b2AhVBXRoKHUX6CxIQFnoECAQQAQ&url=http%3A%2F%2Fdocnum.univ-lorraine.fr%2Fpublic%2FDDOC_T_2013_0042_HERNANE.pdf&usg=AOvVaw3-LK-JBdtkXiesfrvgqjgq
- https://tel.archives-ouvertes.fr/tel-01077962/document : Algorithmique distribuée d'exclusion mutuelle : vers une gestion efficace des ressources
- https://www.yumpu.com/fr/document/read/38294144/algorithme-de-naimi-trehel
- https://www.geeksforgeeks.org/raymonds-tree-based-algorithm/
- https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwiQz6aG97b2AhVBXRoKHUX6CxIQFnoECBkQAQ&url=https%3A%2F%2Fpages.lip6.fr%2FPierre.Sens%2Fpublications%2FCFSE05.ps&usg=AOvVaw2UubKz9J7ohg9jjcarWqDL
- Marin Bertier, Luciana Bezerra Arantes, and Pierre Sens. Hierarchical token based mutual exclusion algorithms. In CCGRID, pages 539–546, 2004.