

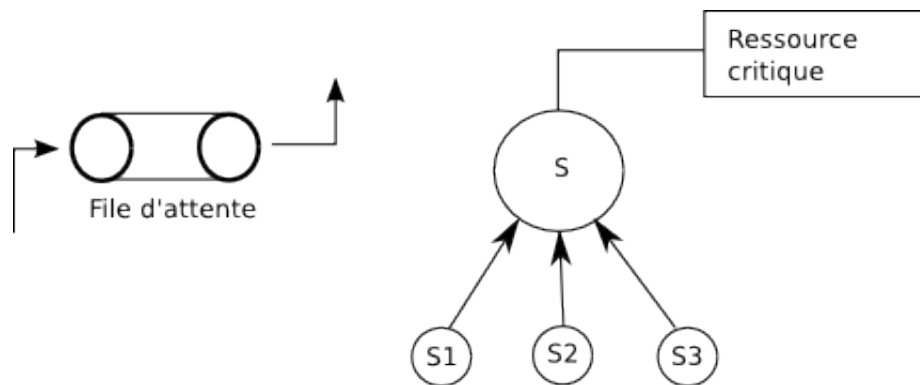
## L'exclusion mutuelle distribuée

### L'algorithme de L'Amport

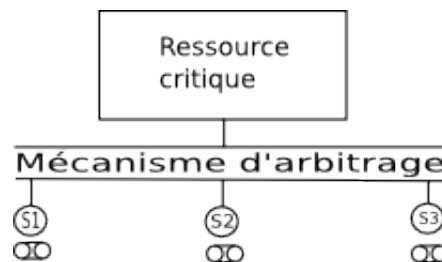
L'algorithme est basé sur 2 concepts :

- L'estampillage des messages
- La distribution d'une file d'attente sur l'ensemble des sites du système distribué

L'algorithme d'exclusion mutuelle sur un serveur centralisé est le suivant :



On souhaiterait obtenir :



Chaque site bénéficiera de sa propre file d'attente.

- Chaque site  $s_i$  possède une horloge logique  $h_i$
- $h_i$  vaut initialement 0
- Tout message envoyé par  $s_i$  est estampillé par son propre id et par  $h_i$  de la manière suivante :

Lors de la réception de MESSAGE(m, i,  $h_i$ ) Faire

$h_i \leftarrow h_i + 1$

Envoyer à son voisin MESSAGE(m, i,  $h_i$ )

Fin

```

Lors de la réception de MESSAGE(m,j, h_i) Faire
  h_i <- max(h_i, h_j) + 1
Fin

```

Ce mécanisme d'estampillage permet d'établir un ordre total et unique sur tous les messages échangés :

Un message  $(m, i, h_i)$  précède un message  $(m, j, h_j)$  si et seulement si :

$$(h_i < h_j) \cup (h_i = h_j \cap (i < j))$$

## Hypothèse

Le système centralisé est fiable, il vérifie :

- Pas de perte de messages
- Pas d'altération de messages
- Pas de duplication des messages
- Le délai de transmission est aléatoire, mais fini

## Principes

Un site voulant accéder à la ressource critique diffuse sa requête à tous les autres sites.

Les sites recevant la requête l'enregistrent dans leurs files d'attente locale et acquittent le message.

Le mécanisme d'estampillage permet d'ordonnancer totalement les messages (l'asynchronisme et la vitesse d'acheminement des messages est variable, ce qui justifie l'estampillage).

Lorsqu'un site quitte la section critique, il diffuse un message de libération à tous les autres sites.

Un site accède à la ressource critique quand sa requête est la plus ancienne dans sa propre file d'attente et donc dans toute les files d'attente de tous les autres sites (même ordre grâce à l'estampillage).

## Variables locales à chaque processus $P_i$

On suppose les sites numérotés de 0 à  $n - 1$ .

- $h_i \in \mathbb{N}$ , horloge logique locale de processus
- $f_i[0; n - 1]$ , vecteur de messages : désigne la file d'attente de  $P_i$ . Par exemple,  $f_i[j]$  contient le dernier message envoyé de  $P_i$  à  $P_j$ . Initialement,  $\forall j \in [0; n - 1], f_i[j] < -(lib, 0, j)$ .

## Messages utilisés

MESSAGE(type, h, n-site) : message envoyé par  $P_i$  à tous ses voisins.  $\text{type} \in \text{REQ}, \text{LIB}, \text{ACK}$ , où :

- REQ : demande d'accès à la ressource critique
- LIB : signification de libération de la ressource critique
- ACK : accusé de réception (acquiescement).
- h : horloge locale
- n-site : identifiant de processus

## Code de l'algorithme

Lors de l'accès à la ressource critique Faire

```
diffuser MESSAGE(REQ, h_i, i)
f_i[i] <- (REQ, h_i, i)
h_i <- h_i + 1
attendre;
Pour tout j != i Faire
    Si ESTAMPILLE(f_i[i]) < estampillage(f_j[i]) Alors
        << Section Critique >>
    Finsi
Fin pour
diffuser MESSAGE(LIB, h_i, i)
f_i[i] <- (LIB, h_i, i)
h_i <- h_i + 1
```

Fin

Lors de la réception de MESSAGE(REQ, k, j) Faire

```
h_i <- max(h_i, k) + 1
f_i[j] <- (REQ, k, j)
envoyer MESSAGE(ACK, h_i, i) à P_j
h_i <- h_i + 1
```

Fin

Lors de la réception de MESSAGE(LIB, k, j) Faire

```
h_i <- max(h_i, k) // ou min ? à vérifier
f_i[j] <- (LIB, h_i, i)
h_i <- h_i + 1
```

Fin

```
Lors de la réception de MESSAGE(ACK, k, j) Faire
  h_i <- max(h_i, k) // ou min, à vérifier
  Si (type(f_i[j]) != REQ) Alors
    f_i[j] <- (ACK, k, j)
  Finsi
Fin
```

```
Fonction estampille(t, h, num) Faire
  retourner(h, num)
Fin
```

```
Fonction type(t,h,num) Faire
  retourner t
Fin
```

## Appréciation de la complexité

En terme de nombre de message, on a  $3(n - 1)$ , soit équivalent à une complexité en  $O(n^2)$ .

## L'algorithme de Naimi-Tréhel

Cet algorithme est intéressant car le nombre de message est réduit à  $O(\log(n))$ , donc il est meilleurs que les algorithmes linéaires (i.e. en  $O(n)$ ).

## Hypothèses

On considère qu'il n'y a pas de faute lors de l'exécution (pas de perte de messages, pas de duplication et pas d'altération).

Le maillage est complet.

Un arbre est construit à partir du graphe. La racine est le processus ayant le plus petit identifiant.

## Principes

Les sites (processus) sont structurés sous forme d'une arborescence. Chaque site connaît son père (prédécesseur), à l'exception de la racine.

Une demande d'entrée en section critique est acheminée de père en père jusqu'à la racine. Celle-ci se trouve dans l'un des cas suivants :

- Elle possède le privilège d'utiliser le jeton mais ne l'utilise pas (pas d'entrée en Section Critique). Elle envoie une réponse positive au premier demandeur de la S.C.
- Elle possède le privilège et l'utilise. Elle met les demandeurs en attente.
- Elle attend le privilège. Elle met les demandes en attente avec mise à jour de son suivant.

### Variables locales

- $privilege_i$  : Booléen initialisé à Faux, sauf pour la racine. Il vaut Vrai si  $P_i$  possède le jeton.
- $pred_i$  : pointeur sur le dernier demandeur connu de  $P_i$ .  $pred_i$  vaut null si  $P_i$  est racine.
- $suivant_i$  : indique le successeur de  $P_i$  souhaitant entrer en S.C.  $suivant_i <-$  null lorsqu'aucun autre processus attend l'entrée en S.C.
- $demandeur_i$  : booléen initialisé à Faux, valant Vrai lorsque  $P_i$  est demandeur de la S.C.

### Messages utilisés

$REQUETE(P_k)$  : demande d'entrée en S.C. de la part du processus  $P_k$ .

JETON() : Message envoyé au suivant pour lui signifier son droit d'entrée en S.C.

### Code de l'algorithme

Lors de la demande d'entée en S.C. par P\_i Faire

```

demandeur_i <- Vrai
Si (pred_i != Nil) Alors
    envoyer REQUETE(P_i) à pred_i
    pred_i <- Nil
Finsi

```

```

attendre (privilege_i <- Vrai)

```

```

< S.C. >

```

```

privilege_i <- Faux
demandeur_i <- Faux
Si (suivant_i != Nil) Alors
    envoyer JETON() à suivant_i
    suivant_i <- Nil
Finsi

```

Fin

```
Lors de la réception de REQUETE(P_k) par P_i depuis P_j Faire
  Si (pred_i = Nil) Alors
    Si (demandeur_i = Vrai) Alors
      suivant_i <- P_k
    Sinon
      envoyer JETON() à P_k
      privilege_i <- Faux
    Finsi
  Sinon
    envoyer REQUETE(P_k) à pred_i
  Finsi
  pred_i <- P_k
Fin
```

```
Lors de la réception de JETON() par P_i depuis P_j Faire
  privilege_i <- Vrai
Fin
```