Algorithmique Distribuée
Exclusion mutuelle distribuée

Laurent PHILIPPE

Master 2 Informatique
UFR des Sciences et Techniques

2013/2014

Références

- Distributed Systems, Principle and Paradigms, A.Tanenbaum, Pearson Ed.
- M.Raynal, Une introduction aux principes des systèmes répartis (3 tomes), EyrollesEd. (1991)
- Cours de C.Kaiser du Cnam.

Sommaire

Sommaire

Exclusion mutuelle : rappels

- Une ressource partagée ou une section critique n'est accédée que par un processus à la fois
- Un processus est dans les trois états possibles, par rapport à l'accès à la ressource :
  - ***demandeur***
  - ***dedans***
  - ***dehors***
- Changement d'états par un processus :
  - de ***dehors*** à ***demandeur***
  - de ***dedans*** à ***dehors***
- Couche middleware gère le passage de l'état ***demandeur*** à l'état ***dedans***

Exclusion mutuelle : rappels

Propriétés

Accès en exclusion mutuelle doit respecter 2 propriétés :

- Sûreté : un processus au maximum en section critique (état dedans)
- Vivacité : toute demande d'accès à la section critique est satisfaite en un temps fini

Mécanismes d'exclusion mutuelle distribuée

En système distribué, nous présentons deux classes de mécanismes d'exclusion mutuelle

- les mécanismes centralisés
- les mécanismes distribués

Sommaire

Mécanisme de synchronisation centralisé

Synchronisation centralisée : définition

Mécanisme centralisé de synchronisation en système distribué regroupe les algorithmes qui utilisent un coordinateur pour gérer l'exclusion mutuelle. Ce coordinateur :

- centralise les requêtes des différents processus de l'application
- réalise l'exclusion mutuelle en accordant la permission d'entrée en Section Critique
- gère une file des requêtes en attente de Section Critique.

L'algorithme

Processus P_i

- Quand un processus P_i veut entrer en Section Critique, il envoie au coordinateur un message de "demande d'entrée en Section Critique". Il attend la permission du coordinateur, avant d'entrer en Section Critique.
- Lorsque P_j reçoit la permission, il entre en Section Critique.
- Quand P_i quitte la Section Critique, il envoie un message de sortie de Section Critique au coordinateur.

L'algorithme

Coordinateur

- Si personne en Section Critique, à la réception d'une demande d'accès de P_i, il retourne "permission d'accès" à P_i.
- Si un processus P_j est déjà en Section Critique. Le coordinateur refuse l'accès. La méthode dépend de l'implantation :
  - soit il n'envoie pas de réponse, le processus P_j est bloqué.
  - soit il envoie une réponse : "Permission non accordée".

  Dans les deux cas, le coordinateur dépose la demande de P_j dans une file d'attente.
- A la réception d'un message de sortie de Section Critique, le coordinateur prend la première requête de la file d'attente de la Section Critique et envoie au processus un message de permission d'entrée en Section Critique.

Mécanisme de synchronisation centralisé

Avantages

- Algorithme garantit l'exclusion mutuelle,
- Algorithme juste (les demandes sont accordées dans l'ordre de réception)
- Pas de famine (aucun processus ne reste bloqué)
- Solution facile à implanter
- Pas de supposition sur l'ordre des messages
- Complexité : maximum 2 ou 3 messages

Mécanisme de synchronisation centralisé

Inconvénients

- Si coordinateur a un problème, tout le système s'effondre
- Si processus bloqués lors de demande d'accès à une Section Critique occupée : impossibilité de détecter la panne du coordinateur
- Dans de grands systèmes, le coordinateur = goulet d'étranglement

Sommaire

Synchronisation distribuée

Rappel

- Processus ne communiquent que par échange de messages
- Chaque processus connaît l'ensemble des processus du système
- Chaque processus a ses propres variables, pas de partage de variables
- On ne considère pas les cas de pertes de messages ni les pannes de machines
- Les échanges de messsages sont FIFO : les messages ne se doublent pas.

Synchronisation distribuée

Principe

- Diffuser la demande
- Il faut l'accord de tous les autres
- Une fois le consensus obtenu on accède à la SC
- Difficulté : demandes concurrentes

Algorithme de Lamport

- Utilise les horloges logiques pour ordonner les évènements
- P_i incrémente son horloge entre deux évènements locaux ou distants
- A la réception d'un message par P_i d'horloge H, l'horloge H_i devient $max(H_i, H) + 1$

Algorithme de Lamport

Variables pour chaque processus P_i

- Horloge locale H_i
- Tableau des horloges des autres processus, initialisé à 0
- Chaque processus maintient un tableau de requêtes : état de demande des autres processus (ACK, REL, REQ)
- Initialement, chaque processus connaît tous les autres participants : liste des voisins

Algorithme de Lamport

Principe de l'algorithme(1)

- Quand P_i veut entrer en Section Critique, il incrémente son horloge, envoie le message $REQ(H_i : P_i)$ de demande de SC à tous les autres processus et dépose ce message dans tableau des requêtes
- Quand un processus P_i reçoit le message $REQ(H_j : P_j)$ de demande de SC, il synchronise son horloge, met le message dans son tableau de requêtes et envoie un message ACK daté de bonne réception à P_j

Algorithme de Lamport

Principe de l'algorithme (2)

- P_i accède à la SC quand les deux conditions suivantes sont réalisées :
  - Il existe un message $REQ(H_i : P_i)$ dans son tableau de requêtes qui est ordonné devant tout autre requête selon la relation de précédence
  - Le processus P_i a reçu un message de tous les autres processus ayant une date supérieure à H_i

  Ces 2 conditions sont testées localement par P_i
- Quand P_i quitte la SC, il enlève sa requête $REQ(H_i : P_i)$ de demande de SC de son tableau de requêtes, incrémente son horloge et envoie un message daté $REL(H_i : P_i$ pour signaler qu'il quitte la SC à tous les processus
- Quand P_i reçoit le message $REL(H_j : P_j)$ (quitte SC), il synchronise son horloge et enlève le message $REQ(H_j : P_j)$ (demande de SC) de son tableau de requêtes

Algorithme de Lamport

Variables utilisées par chaque processus P_i

H_i : entier indiquant l'horloge locale. Initialement H_i=0
$F\_H_i[]$: tableau contenant les horloges des différents processus. La taille du tableau correspond au nombre de participants.
$F\_M_i[]$: tableau contenant les messages correspondant
V_i : ensemble de tous les voisins de P_i

Messages utilisés

REQ(H) : demande d'entrée en SC
ACK(H) : acquittement d'une demande d'entrée en SC
REL(H) : sortie de SC

Algorithme de Lamport

Règle 1

Accepte la demande de Section Critique **Faire**

$H_i := H_i + 1$

$\forall$ j $\in V_i$ **Envoyer** REQ(H_i) à j

$F\_H_i[i] := H_i$

$F\_M_i[i] :=$ REQ

Attendre

$\forall j \in V_i((F\_H_i[i] < F\_H_i[j]) \vee ((F\_H_i[i] = F\_H_i[j]) \wedge i < j))$

<Section Critique>

Fait

Algorithme de Lamport

Règle 2

Accepte le message REQ(H) depuis j **Faire**

$H_i := \text{Max}(H_i, \text{H}) + 1$

$F\_H_i[j] = \text{H}$

$F\_M_i[j] = \text{REQ}$

Envoyer ACK(H_i) à j

Fait

Algorithme de Lamport

Règle 3

Accepte le message ACK(H) depuis j **Faire**
H_i := Max(H_i,H) + 1
Si $F\_M_i[j] \neq REQ$ **Alors**
$F\_H_i[j]$ = H
$F\_M_i[j]$ = ACK
Fsi
Fait

Algorithme de Lamport

Règle 4

Accepte la libération de la Section Critique **Faire**

$H_i := H_i + 1$

$\forall j \in V_i$ **Envoyer** REL(H_i) à j

$F\_H_i[i] = H_i$

$F\_M_i[i] = \text{REL}$

Fait

Règle 5

Accepte le message REL(H) depuis j **Faire**

$H_i := \text{Max}(H_i, \text{H}) + 1$

$F\_H_i[j] = \text{H}$

$F\_M_i[j] = \text{REL}$

Fait

Déroulement de l'algorithme de Lamport

Exercice avec 3 processus : P_0, P_1 et P_2

1. P_0 demande à entrer en section critique. Dérouler l'algorithme jusqu'à ce que P_0 sorte de SC.
2. P_1 demande. Dérouler l'algorithme jusqu'à ce que P_1 sorte de SC.
3. Les processus P_0 et P_2 demandent la SC en même temps. Dérouler l'algorithme jusqu'à ce que le premier des deux sorte de SC.
4. P_1 demande à entrer en SC à ce moment là. Dérouler l'algorithme jusqu'à ce que P_1 sorte de SC.
5. Les processus P_1 et P_2 demandent la SC en même temps. Dérouler l'algorithme jusqu'à ce que P_1 et P_2 sortent de SC.

Exercice

Que se passe-t-il si la propriété FIFO n'est pas respectée ?

Algorithme de Lamport

Propriétés de l'algorithme

- Sûreté : exclusion mutuelle garantie
- Vivacité garantie
- Algorithme exempt d'inter-blocage

Algorithme de Lamport : Preuve sûreté

Par contradiction :

- Supposons que P_i et P_j sont en même temps dans la SC à l'instant t,
- Alors la condition d'accès doit être valide sur les deux processus en même temps et ils sont tous les deux dans l'état REQ
- On peut supposer sans perte de généralité qu'à un instant t, $H(S_i) < H(S_j)$ (ou l'inverse, ce qui revient au même) puisque deux horloges logiques ne sont jamais identiques
- A partir de l'hypothèse de communication FIFO, il est clair que le message de requête de S_i est dans le tableau F_M_j au moment où P_j entre en SC puisque P_j attend toutes les réponses (ACK) des processus avant d'entrer, donc celui de de P_i.
- Or l'acquittement de P_i a été généré après que P_i ait émis le message REQ puisque $H(S_i) < H(S_j)$
- Donc P_j est entré en SC alors que $H(S_i) < H(S_j) \rightarrow$ contradiction

Algorithme de Lamport

Vivacité

- Toute demande d'entrée en Section Critique sera satisfaite au bout d'un temps fini.
- En effet, après la demande d'un processus P_i, au plus, $N-1$ processus peuvent entrer avant lui

Complexité

$3 * (N-1)$ messages par entrée en Section Critique, où N est le nombre de processus

- $N-1$ Requêtes
- $N-1$ Acquittements
- $N-1$ Libérations

Algorithme de Ricart et Agrawala

Objectif

Algorithme proposé dans le but de diminuer le nombre de messages par rapport à l'algorithme de Lamport

Changements

- Regrouper information de sortie de SC et accord
- Pas de retour systématique de ACK
- N'informer que les processus en attente à la sortie de SC

Algorithme de Ricart et Agrawala

Principe de l'algorithme

- Lorsqu'un processus P_i demande la Section Critique, il diffuse une requête datée à tous les autres processus.
- Lorsqu'un processus P_i reçoit une requête de demande d'entrée en Section Critique de P_j, deux cas sont possibles :
  - Cas 1 : si le processus P_i n'est pas demandeur de la Section Critique, il envoie un accord à P_j.
  - Cas 2 : si le processus P_i est demandeur de la Section Critique et si la date de demande de P_j est plus récente que la sienne, alors la requête de P_j est différée, sinon un message d'accord est envoyé à P_j.
- Lorsqu'un processus P_i sort de la Section Critique, il diffuse à tous les processus dont les requêtes sont différées, un message de libération.

Algorithme de Ricart et Agrawala

Variables utilisées par chaque processus P_i :

H_i : entier, estampille locale. Initialement $H_i = 0$
HSC_i : entier, estampille de demande de SC. Initialement $HSC_i = 0$
R_i : booléen, le processus est demandeur de SC. Init à $R_i =$ FAUX
X_i : ensemble, processus dont l'accord est différé. Init à $X_i := \emptyset$
$Nrel_i$: entier, nombre d'accords attendus. Init à $Nrel_i = 0$.
V_i : voisinage de i

Messages utilisés :

$REQ(H)$: demande d'entrée en SC
$REL()$: message de permission

Algorithme de Ricart et Agrawala

Règle 1

Accepte la demande de SC **Faire**

$R_i := \text{VRAI}$

$HSC_i := H_i + 1$

$Nrel_i := cardinal(V_i)$

$\forall j \in V_i$ **Envoyer** REQ(HSC_i) à j

tant que ($Nrel_i \neq 0$)

< SC>

Fait

Algorithme de Ricart et Agrawala

Règle 2

Accepte le message REQ(H) depuis j **Faire**
  H_i := Max(HSC_i,H) + 1
  Si $R_i \wedge ((HSC_i < H) \vee ((HSC_i = H) \wedge i < j))$ **Alors**
    $Xi := Xi \cup j$
  Sinon
    Envoyer REL() à j
  FSi
Fait

Algorithme de Ricart et Agrawala

Règle 3

Accepte le message REL( ) depuis j **Faire**

$Nrel_i := Nrel_i - 1$

Fait

Règle 4

Accepte la libération de la SC **Faire**

$R_i := FAUX$

$\forall j \in X_i$ **Envoyer** REL() à j

$X_i := \emptyset$

Fait

Algorithme de Ricart et Agrawala

Déroulement avec 3 processus : P_0, P_1, P_2

1. Le processus P_1 demandent l'accès à la Section Critique, jusqu'à sa sortie de SC
2. Le processus P_2 demande l'accès à la Section Critique, une fois qu'il est entré, P_1 demande, jusqu'à la sortie des deux.
3. P_0 demande la SC en même temps que P_2, lorsque l'un des deux processus est en SC, P_1 demande, jusqu'à la sortie des trois.

Question

Pourquoi deux horloges logiques ?

Algorithme de Ricart et Agrawala

Preuve

- Par contradiction
- On suppose que P_i et P_j sont les deux en section critique.
- On peut supposer sans perte de généralité que la requête de P_i a une horloge plus petite que celle de P_j
- Puisque l'horloge de P_i est plus petite que celle de P_j cela signifie qu'il a reçu la requête de P_j après avoir fait sa demande
- D'après l'algorithme P_j ne peut entrer en section critique que si P_i lui a envoyé un message *REL*
- Puisque P_i est en section critique alors il a envoyé ce message *REL* avant d'entrer en section critique
- Ce qui est impossible puisque l'horloge de P_j est supérieure à celle de P_i, or d'après l'algorithme P_i n'envoie *REL* que si l'horloge reçue

Algorithme de Ricart et Agrawala

Complexité

Une utilisation de la Section Critique nécessite $2 * (N - 1)$ messages :

- $N - 1$ Requêtes
- $N - 1$ Permissions

Synchronisation distribuée utilisant la notion d'horloges logiques

Algorithme de Carvalho et Roucairol

Algorithme proposé dans le but de diminuer le nombre de messages par rapport à l'algorithme de Ricart et Agrawala

Algorithme de Carvalho et Roucairol

Principe de l'algorithme

Soit le cas où P_i demande l'accès à la Section Critique plusieurs fois de suite (état ***demandeur*** plusieurs fois de suite) alors que P_j n'est pas intéressé par celle-ci (état ***dehors***)

- Avec l'algorithme de Ricart et Agrawala, P_i demande la permission de P_j à chaque nouvelle demande d'accès à la Section Critique
- Avec l'algorithme de Carvalho et Roucairol, puisque P_j a donné sa permission à P_i, ce dernier la considère comme acquise jusqu'à ce que P_j demande sa permission à P_i : P_i ne demande qu'une fois la permission à P_j

Algorithme de Carvalho et Roucairol

Variables utilisées par chaque processus P_i

HSC_i : entier, estampille de demande d'entrée en SC, initialisée à 0
H_i : entier, estampille locale, initialisée à 0
R_i : booléen, le processus est demandeur de SC, init à FAUX
SC_i : booléen, le processus est en SC, init à SC_i = FAUX
X_i : ensemble, processus dont l'envoi de l'avis de libération est différé. Init à $X_i := \emptyset$
XA_i : ensemble des processus desquels i attend une autorisation. Init à $XA_i := V_i$
$Nrel_i$: nombre d'avis de libération attendus, init à 0.
V_i : voisinage de i

Algorithme de Carvalho et Roucairol

Messages utilisés

REQ(H) : demande d'entrée en SC
REL() : message de permission

Algorithme de Carvalho et Roucairol

Règle 1

Accepte la demande de SC **Faire**

R_i := VRAI

$HSC_i := H_i + 1$

$Nrel_i := cardinal(XA_i)$

$\forall j \in XA_i$ **Envoyer** REQ(HSC_i) à j

Tant que ($Nrel_i \neq 0$)

SC_i=VRAI

< SC>

Fait

Algorithme de Carvalho et Roucairol

Règle 2

Accepte le message REQ(H) depuis j **Faire**
  H_i := Max(HSC_i,H) + 1
  Si $SC_i \vee (R_i \wedge ((HSC_i < H) \vee ((HSC_i = H) \wedge i < j)))$ **Alors**
    $X_i := X_i \cup j$
  Sinon
    Envoyer REL() à j
    Si $R_i \wedge (\neg SC_i) \wedge j \notin XA_i$ **Alors**
      Envoyer REQ(HSC_i) à j
    FSi
    $XA_i := XA_i \cup j$
  FSi
Fait

Algorithme de Carvalho et Roucairol

Règle 3

Accepte le message REL( ) depuis j **Faire**

$Nrel_i := Nrel_i - 1$

Fait

Règle 4

Accepte la libération de la SC **Faire**

$R_i := FAUX$

$SC_i := FAUX$

$XA_i := X_i$

$\forall j \in X_i$ **Envoyer** REL() à j

$X_i := \emptyset$

Fait

Algorithme de Carvalho et Roucairol

Déroulement de l'algorithme avec 3 processus : P_0, P_1 et P_2

1. Les horloges logiques de chaque processus sont à 0
2. Les processusP_2 et P_1 demandent l'accès à la Section Critique en même temps
3. Déroulement de l'algorithme jusqu'à ce que les deux processus P_1 et P_2 soient entrés et sortis de Section Critique
4. Les processus P_0 et P_1 demandent la section critique
5. Arrêt du déroulement de l'algorithme lorsque les trois processus P_0 et P_1 seront entrés et sortis de Section Critique

Algorithme de Carvalho et Roucairol

Question

Pourquoi un processus qui reçoit une REQ renvoie-t-il parfois une REQ ?

Complexité

Le nombre de messages requis pour une utilisation de la Section Critique est :

- pair car à toute requête d'entrée en SC correspond un envoi de permission
- fonction de la structure du système : varie entre 0 et 2*(N-1)

Sommaire

Synchronisation distribuée utilisant un jeton

Algorithmes

Différentes structures de communication :

- Anneau : Le Lann
- Diffusion : Suzuki-Kazami
- Arbre : Naimi-Trehel

Algorithme de Le Lann

Système structuré en anneau

- Algorithme utilisant un jeton pour gérer l'accès à une Section Critique
- Jeton (unique pour l'application) = permission d'entrer en Section Critique
- Un seul processus détient le jeton à un moment donné

Algorithme de Le Lann

Construction de l'anneau

- Construction d'un anneau avec l'ensemble des processus de l'application de façon logique
- Attribution d'une place à chacun des processus
- Chaque processus connaît son successeur dans l'anneau
- Initialisation : jeton attribué à un processus (exemple : Processus P_0)

Algorithme de Le Lann

Principe de l'algorithme

- Quand un processus reçoit le jeton de son prédécesseur dans l'anneau :
  - s'il est demandeur de Section Critique, il garde le jeton et entre en Section Critique
  - s'il n'est pas demandeur de Section Critique, il envoie le jeton à son successeur dans l'anneau
- A la sortie de Section Critique, le processus envoie le jeton à son successeur dans l'anneau
- Pour des raisons d'équité, un processus ne peut pas entrer deux fois de suite en Section Critique

Algorithme de Le Lann

Exercice

- Quelles sont les données nécessaires ?
- Quels sont les messages échangés ?
- Quelles règles doivent être mises en place ?
- Écrire les règles

Algorithme de Le Lann

Avantages de l'algorithme

- Simple à mettre en œuvre
- Intéressant si nombreux demandeurs de la ressource
- Équitable en terme de nombre d'accès et de temps d'attente

Algorithme de Le Lann

Inconvénients (1)

- Nécessite des échanges de messages même si aucun processus ne veut accéder à la Section Critique
- Temps d'accès à la Section Critique peut être long
- Perte du jeton :
  - difficulté de détecter un tel cas
  - impossibilité de prendre en compte le temps écoulé entre deux passages du jeton (temps passé en Section Critique est très variable)
  - des algorithmes gèrent ce problème de perte et régénération de jeton sur un anneau (non vu dans ce cours)

Algorithme de Le Lann

Inconvénients(2)

- Problème de la panne d'un processus
  - plus facile à gérer que dans les algorithmes précédents
  - envoi d'un acquittement à la réception du jeton : permet de détecter la panne de l'un des processus
  - le processus mort peut être retiré de l'anneau et le suivant prend alors sa place
  - pour implanter ceci : chaque processus doit connaître la configuration courante de l'anneau.

Synchronisation distribuée utilisant un jeton

Modification de l'algorithme avec anneau

- Jeton symbolise le droit d'entrée en section critique
- Un seul processus détient le jeton à un moment donné
- Pas forcément besoin de la structure d'anneau

Exercice

- Ecrire un algorithme où les participants échangent un jeton pour accéder à la section critique.
- Définir les données, les messages et les règles.
- Quelles propriétés possède cet algorithme : sûreté, vivacité, équité ?

Algorithme de Suzuki-Kasami

Principe de l'algorithme

- Chaque processus connaît l'ensemble des participants
- Quand P_i veut accéder à la Section critique, il envoie un message ***REQ(H)*** à tous les participants
- Le jeton contient l'estampille de la dernière visite qu'il a effectué à chacun des processus
- Quand P_i sort de Section Critique, il cherche le premier processus P_k tel que l'estampille de la dernière requête de P_k soit supérieure à celle mémorisée dans le jeton (correspondant à la dernière visite du jeton à P_k)

Synchronisation distribuée utilisant un jeton

Exercice

- Quelles sont les propriétés de l'algorithme ?
- Comment améliorer l'équité ?

Synchronisation distribuée utilisant un jeton

Algorithme de Naimi-Trehel

Eviter de diffuser la requête initiale

Principe

- Les processus sont logiquement organisés en arbre construit à partir des requêtes de demande d'accès en Section Critique
- Seul le processus qui possède le jeton peut accéder à la section critique

Algorithme de Naimi-Trehel

Structures de données

Ces structures de données sont gérées de façon distribuée :

- Une file de requêtes : ***next***
- Un arbre de chemins vers le dernier demandeur de SC : ***owner***

Algorithme de Naimi-Trehel

File de requêtes : *next*

- Processus en tête de file possède le jeton
- Processus en queue de file est le dernier processus qui a fait une demande d'entrée en SC
- Chaque nouvelle requête de SC est placée en queue de file

Algorithme de Naimi-Trehel

Arbre de chemins vers le dernier demandeur de SC : *owner*

- Racine de l'arbre : dernier demandeur de SC (= queue de la file des ***next***)
- Une nouvelle requête est transmise suivant le chemin des pointeurs de ***owner*** jusqu'à la racine de l'arbre (***owner*** = nil)
  - reconfiguration dynamique de l'arbre : nouveau demandeur devient la racine de l'arbre
  - les processus faisant partie du chemin entre la nouvelle racine et les anciennes racines modifient leurs pointeurs ***owner*** : ***owner*** = nouvelle racine

Algorithme de Naimi-Trehel

Principe de l'algorithme (1)

Chaque processus P_i gère localement les éléments suivants :

- ***owner*** : processus qui possède probablement le jeton. Toute demande de P_i d'accès à la Section Critique sera envoyée à ce processus.
- ***next*** : processus à qui P_i devra transmettre le jeton à sa sortie de Section Critique. ***next*** est le dernier processus qui a envoyé à P_i une requête d'entrée en Section Critique.

Algorithme de Naimi-Trehel

Principe de l'algorithme (2)

Initialisation :

- ***elected-site*** : processus désigné comme étant le propriétaire du jeton. C'est la racine de l'arbre.
- ***owner*** : initialisé à ***elected-site***
- ***next*** : initialisé à ***nil***

Algorithme de Naimi-Trehel

Principe de l'algorithme (3)

Quand un processus P_i demande l'accès à la Section Critique

- S'il ne possède pas le jeton, il envoie sa requête à ***owner*** et attend de recevoir le jeton
- S'il possède déjà le jeton, il entre en Section Critique

Algorithme de Naimi-Trehel

Principe de l'algorithme (4)

Quand un processus P_i reçoit une requête d'entrée en SC de la part de P_j

- S'il possède le jeton et qu'il n'est pas en SC alors il envoie le jeton à P_j et met à jour sa variable ***owner***
- S'il possède le jeton et s'il est en SC, il met à jour ***next***
- S'il ne possède pas le jeton et qu'il attend la section critique : il transmet la requête à ***owner***
- S'il ne possède pas le jeton et qu'il attend la section critique : il enregistre la demande dans ***next***

Algorithme de Naimi-Trehel

Principe de l'algorithme (5)

Quand un processus P_i sort de SC, si ***next*** est initialisé il envoie le jeton à ***next*** et réinitialise ***next*** à nil

Algorithme de Naimi-Trehel

Déroulement de l'algorithme

Soient 5 processuss A, B, C, D et E

- Initialisation : le processus A possède le jeton (electednode)
- Le processus B demande l'accès à la section critique
- Lorsque le processus B est en Section Critique, le processus C en demande l'accès
- Lorsqu'ils sont sortis de la section critique le processus E demande.
- Quel est l'état de l'arbre des demandeurs au cours de ces demandes ?