



Université des sciences et technologie d'Oran USTO-MB
Faculté mathématique et informatique
Département d'informatique



Polycopié du cours

Module ASR

« Algorithmes Systèmes Répartis »

Master 1 option RSID,SID,IA

Chargée de cours

Dr fz MOSTEFA

Année universitaire 2024-2025

Semestre : 1

Intitulé de l'UE : UM211

Intitulé de la matière : Algorithmique et Systèmes Répartis

Crédits : 3

Coefficients : 2

Objectifs de l'enseignement : L'une des tendances majeures des systèmes informatiques est la répartition des traitements entre des "entités" coopératives. Celles-ci peuvent être soit des processeurs d'une machine multi-processeurs, soit des stations de travail d'un réseau local, soit des serveurs d'application connectés par l'Internet. L'objectif de ce cours est donc de décrire les principaux services nécessaires à la conception d'applications réparties.

Connaissances préalables recommandées

Connaissances acquises durant le cursus de formation de la licence : Systèmes informatiques (SI) ou Ingénierie des Systèmes d'Information et du Logiciel (ISIL)

Contenu de la matière

- 1- Généralités
 - 1- modèles de répartition,
 - 2- Scénario, Evaluation et vérification d'algorithmes répartis
- 2- La communication
 - 1- Contrôle de flux
 - 2- Communication synchrone avec RdV
 - 3- Qualité de service : réseau Fifo
- 3- Le temps
 - 1- Temps horloge, temps environnement
 - 2- Environnement synchrone
 - 3- Temps physique
 - 4- Horloges physiques
- 4- Les algorithmes de concurrence
 - 1- Algorithme de Lamport
 - 2- Algorithme de Ricart et Agrawala
 - 3- Algorithme de Carvalho et Roucairol
- 5- L'observation
 - 1- Etat d'une application répartie
 - 2- Détection des propriétés d'une application stables et paisibles
- 6- Les algorithmes d'élection
 - 1- Algorithmes de Chang et Roberts
 - 2- Algorithme de Franklin
- 7- La mémoire virtuelle répartie et linéarisabilité
 - 1- Linéarisabilité par exclusion mutuelle
 - 2- Linéarisabilité avec gestionnaire statique
 - 3- Linéarisabilité avec gestionnaire dynamique
 - 4- Propagation des écritures
- 8- L'autostabilisation
 - 1- Routage auto-stabilisant
 - 2- Gestion de la mémoire virtuelle répartie
 - 3- Exclusion mutuelle

Table des matières

Chapitre 1 : Généralités.....	
1.1 introduction.....	5
1.2 Les objectif de la repartition.....	5
1.3 Les avantages d'un systeme réparti	5
1.4 Modele de répartition.....	5
1.5 L'environnement de répartition.....	6
1.6 Fonctionnement par couche.....	8
1.7 Déroulement de l'application répartie.....	10
1.8 Scénario d'exécution, evaluation, vérification de l'application répartie.....	11
Chapitre 2 : Communication.....	
2.1 introduction.....	13
2.1.1 Contrôle de flux point à point : le modèle producteur / consommateur.....	13
2.2 Communication synchrone : le rendez-vous	18
2.3 Qualité de service.....	20
Chapitre 3 : Le temps.....	
3.1 introduction	22
3.2 Le temps interne.....	22
3.3 Le temps de l'environnement (temps réel).....	22
3.4 Le temps universel	23
3.5 Le temps physique	23

3.6 Le temps logique (les differentes horloges logiques).....	24
Chapitre 4 : La concurrence	
4.1.Introduction.....	33
4.2 Algorithme de Lamport.....	34
4.3.Algorithme de Ricard et Agrawala.....	35
4.4.Algorithme de Carvalho et Roucairol	37
Chapitre 5 : L'observation.....	
5.1 Introduction.....	40
5.2 Algorithme de calcul de l'etat global coherent.....	40
5.3 Détection de propriétés stables et paisibles.....	43
Chapitre 6 : L'election.....	
6.1 Introduction.....	45
6.2 Algorithme de Chang et Robert	45
6.3 Algorithme de Franklin.....	47
Chapitre 7 : Memoire virtuelle réparti	
7.1 Introduction.....	51
7.2 La linéarisabilité.....	52
7.3 La consistance sequentielle.....	71
Chapitre 8 : Auto stabilisation	
8.1 Introduction.....	75
8.2 Routage auto-stabilisant	76
8.3 Gestion de la mémoire virtuelle répartie.....	79
8.4 L'exclusion mutuelle.....	83

Pré-requis : Réseaux, système d'exploitation, algorithmique.

Chapitre 1: Généralités

1.1 Introduction

« Un système réparti est un ensemble d'ordinateurs indépendants qui apparaît à un utilisateur comme un système unique et cohérent. » [Tan94]

La problématique de la répartition est née avec l'idée de faire communiquer un ensemble des ordinateurs via un réseau de communication.

1.2 Objectifs de la répartition

- Partage de ressources
- Tolérance aux fautes
- Plus de capacités de stockage
- Plus de capacités de traitement
- Réduction des coûts

1.3 Avantages d'un système réparti

Parmi les avantages d'un système réparti :

- Performance : puissance de calcul et de stockage importante
- Fiabilité : résistance aux pannes logicielles et matérielles.
- passage à l'échelle

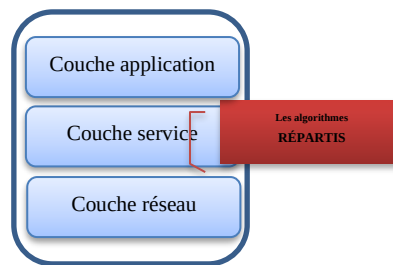
1.4 Modèle de répartition

Avant de réaliser un système réparti, nous devons réfléchir tout d'abord à créer un modèle. En général un modèle est composé d'un ensemble d'hypothèses. Notre modèle de répartition est composé de cinq hypothèses étalé sur un ensemble de couches.

➤ La conception d'applications réparties se heurte à des difficultés algorithmiques spécifiques à l'activité concurrente des entités et à l'absence de mémoire partagée.

Le concepteur doit résoudre des problèmes liés à son application (messagerie, forum, etc.) mais aussi des problèmes liés à son modèle d'application comme :

- La communication entre entités,
- La portabilité et l'interopérabilité des codes,
- L'exclusion mutuelle entre sections critiques, Une manière naturelle d'aborder la conception consiste à construire l'application en couches.
- Une couche applicative dont les traitements sont spécifiques aux fonctionnalités recherchées.
- Une couche service qui résout un ensemble des problèmes liés dans le domaine de la répartition.



1.5 L'environnement réparti :

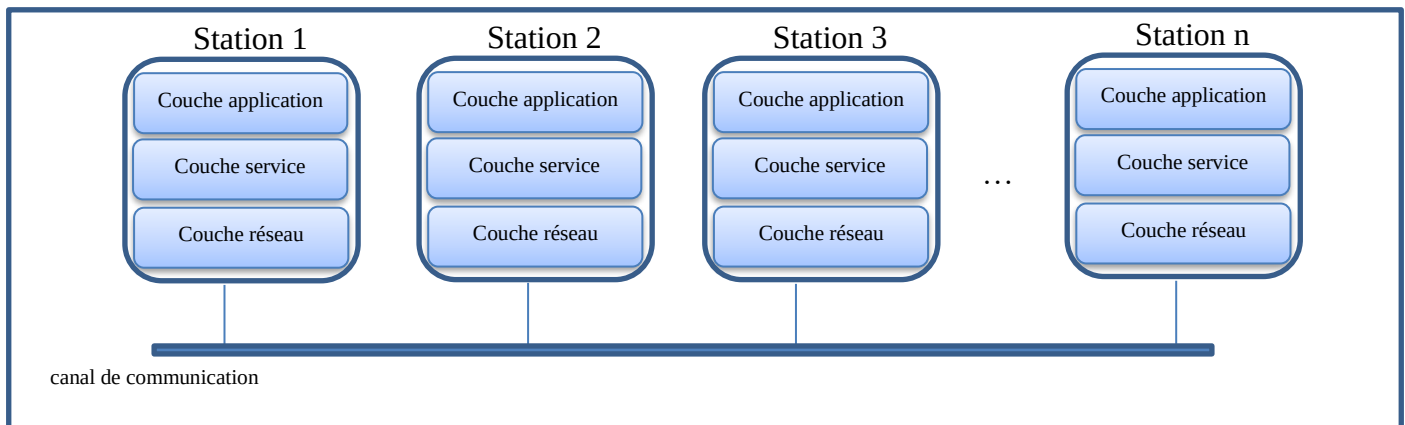


figure 1.1 l'environnement réparti

Un environnement réparti est composé de : [Tan94]

- Entités actives (machines, sites, stations, processus ou processeur) disposant d'une mémoire dédiée non accessible aux autres entités. Chaque station dispose d'une identité unique (cette adresse pourrait être l'adresse MAC de sa carte réseau) ou son adresse IP.
- Lignes de communication bipoints (reliant deux stations) et bidirectionnelles (la transmission d'information est possible dans les deux directions) appelée canal.

Ces lignes sont le seul moyen d'échange d'information entre les stations. Un graphe de communication non orienté est obtenu en considérant les stations comme des nœuds et les lignes de communications comme des arêtes.

Hypothèse n°1 :

- Nous supposons que le graphe de communication est cyclique
- Une station est découpé en trois couches :
 - Couche application : réalise les traitements spécifiques et fait appel à la couche service pour certaines fonctionnalités génériques.
 - Couche service : la réalisation de cette couche est l'objet de cette partie du cours désignée par l'ensemble des algorithmes réparti.
 - Couche réseau : permet l'échange des messages, la couche service se repose sur l'interface de la couche réseau.

1.6 Fonctionnement par couche [Ray91]

Couche réseau

Hypothèse n°2 :

un canal de communication reliant un site S_i à un autre site S_j a les propriétés suivantes :

- Les messages ne sont pas altérés,
- Les messages ne sont pas perdus,
- Le canal est FIFO (First In First Out)

Hypothèse n°3 :

➤ La couche réseau fournit un mécanisme de transfert de message pour la couche service. Deux options sont possibles pour ce mécanisme :

-Communication synchrone : primitives synchrones (bloquantes).

-Communication asynchrone : primitives asynchrones (non bloquantes).

➤ Une émission est synchrone (resp. asynchrone) si le processus émetteur est bloqué (resp. n'est pas bloqué) jusqu'à la réception d'un acquittement de son message.

➤ Une réception est synchrone (resp. asynchrone) si le processus destinataire appelle explicitement (resp. n'appelle pas) une primitive pour recevoir un message et se bloque en attente de ce message.

Hypothèse n°4 :

Les primitives d'émission seront :

➤ ***Envoyer_à (destinataire,message)*** : permet d'envoyer un message au destinataire indiqué et de poursuivre son exécution. Le destinataire est désigné par son identité et le message peut être structuré selon les besoins de la couche service.

➤ **Diffuser (message)** : permet d'envoyer un message à tous les autres sites participant à l'application. L'utilisation de cette primitive n'est possible que si le graphe de communication est sous forme d'un cycle.

Les primitives d'émission seront appelées dans l'algorithme réparti par la couche service. Le trajet d'un message m :

➤ A l'émission : la couche service de S_i appelle la primitive Envoyer_à (S_j, m), puis le message circule sur le canal.

➤ A la réception : le message parvient à la couche réseau de S_j . Le traitement du message par S_j dépend du service mais puisque la réception est asynchrone, aucune primitive n'est appelée par la couche service.

➤ La couche réseau doit appeler une primitive "écrite" par le service qui fait partie de l'algorithme réparti. Cette primitive est une primitive de l'interface service/réseau.

➤ Une primitive montante est une primitive écrite par la couche supérieure et appelée par la couche inférieure.

➤ Une primitive descendante est une primitive écrite par la couche inférieure et appelée par la couche supérieure (comme Envoyer_à). La primitive de réception sera :

➤ **Sur_réception_de (émetteur,message)** : spécifie le traitement à effectuer sur réception d'un message. L'émetteur est désigné par son identité et le message peut être structuré selon les besoins de la couche service.

Couche service

➤ Des variables qui lui sont propres. Une variable var d'une station S_p sera désignée dans l'algorithme par var_p pour insister sur le caractère local de la mémoire.

➤ Des primitives utilisées pour les traitements de l'algorithme.

➤ Des primitives descendantes de l'interface application-service et des primitives montantes (Sur_réception_de) de l'interface service-réseau.

- Un processus d'application ou de service peut être suspendu jusqu'à ce qu'une condition soit vérifiée. La primitive utilisée sera **Attendre (Expression booléenne)**.
- Un processus peut aussi être suspendu jusqu'à l'expiration d'un timeout. Dans ce cas, la primitive utilisée sera **Attendre()**. La primitive qui arme le temporisateur sera Armer (délai).

Couche application

Hypothèse n°5 :

Nous supposons que l'application est exécutée sur un site par un seul processus.

1.7 Déroulement de l'application répartie [Ray91]

- Système d'exploitation capable de mettre en oeuvre 4 règles pour l'interaction entre les différentes couches lors du déroulement de l'application :

-R1 : si le processus applicatif exécute du code de la couche application, toute réception d'un message donne lieu à un déroutement et à l'exécution de la primitive “Sur_réception_de” correspondante. Le traitement applicatif se poursuit ensuite.

-R2 : si le processus exécute du code de la couche service, l'exécution de la primitive “Sur_réception_de” suite à une réception de message est retardée jusqu'à un blocage du processus (par la primitive Attendre) ou jusqu'au retour en couche application.

-R3 : Le code des primitives sur_réception_de ne comporte pas d'appel à la primitive Attendre.

-R4 : Si le système exécute une primitive “sur_réception_de” suite à une réception de message, toute exécution de “sur_réception_de” suite à une autre réception de message est retardée jusqu'à la fin de l'exécution de la première primitive..

1.8 Scénario d'exécution, évaluation et vérification d'algorithmes répartis

1.8.1 Scénario d'exécution

Les algorithmes sont représentés par des chronogrammes. l'activité de chaque site est représentée sur un axe temporel (ici horizontal) orienté de gauche à droite. Nous noterons de plus le début et la fin d'une primitive sur un site par des crochets ouvrant et fermant. Les transferts de message sont dénotés par des segments orientés dont l'origine et l'extrémité sont respectivement associées à l'envoi et à la réception de message.

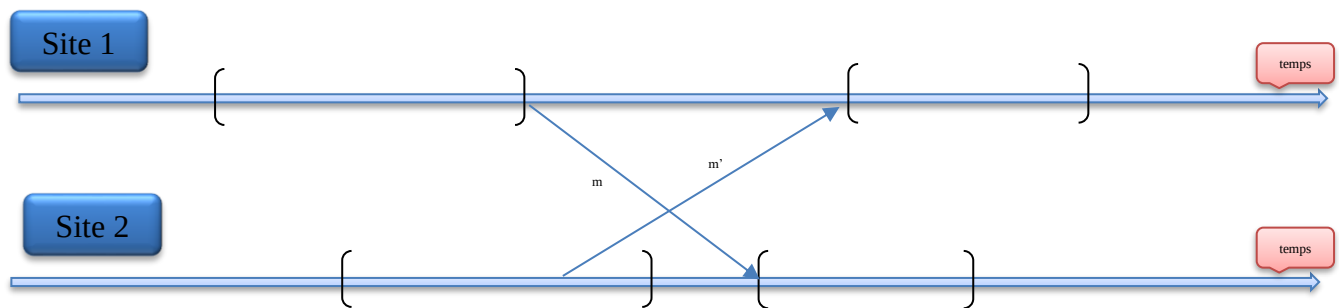


figure 1.2 exemple d'un chronogramme.

1.8.2 Evaluation

Les mesures usuelles de complexité en algorithmique standard sont la taille de l'espace occupé par l'algorithme et le temps d'exécution (ou de manière équivalente le nombre d'instructions exécutées) de l'algorithme.

En algorithmique répartie, sauf cas particulier (comme dans la gestion des tampons), l'espace n'est pas un facteur déterminant. De même, on considère que le temps d'exécution des primitives est négligeable devant le temps de transit des messages.

Ceci nous conduit à deux mesures de complexité pertinentes pour un algorithme réparti qui sont:

- Le trafic sur le réseau
- Le temps d'exécution maximum de l'algorithme en fonction du nombre de sites de l'application réparti.

1.8.3 Vérification

- Un algorithme réparti donne lieu à plusieurs exécutions possibles ne serait-ce qu'en raison du délai aléatoire de transit d'un message. On peut considérer un arbre d'exécution dont les noeuds sont les états accessibles et les arcs sont les événements possibles dans cet état.
- La vérification peut s'effectuer de manière manuelle en utilisant des techniques élémentaires comme l'induction qui consiste à vérifier que la propriété est vérifiée à l'état initial puis que, si elle est vérifiée en un état, elle est vérifiée dans l'état qui le suit. De manière plus élaborée, la vérification peut se faire à travers un logiciel d'aide à la preuve (le plus souvent basé sur la logique). Enfin, dans le cas de systèmes à nombre d'états finis, la validation peut être entièrement automatisée par les techniques dites de "model-checking".

D'une manière générale, les systèmes répartis ne sont pas dotés de aucune mémoire commune et aucune horloge commune d'où l'objectif du modèle de répartition est d'avoir :

- Un comportement global correct du système
- Ordonner les envois et les réceptions des messages

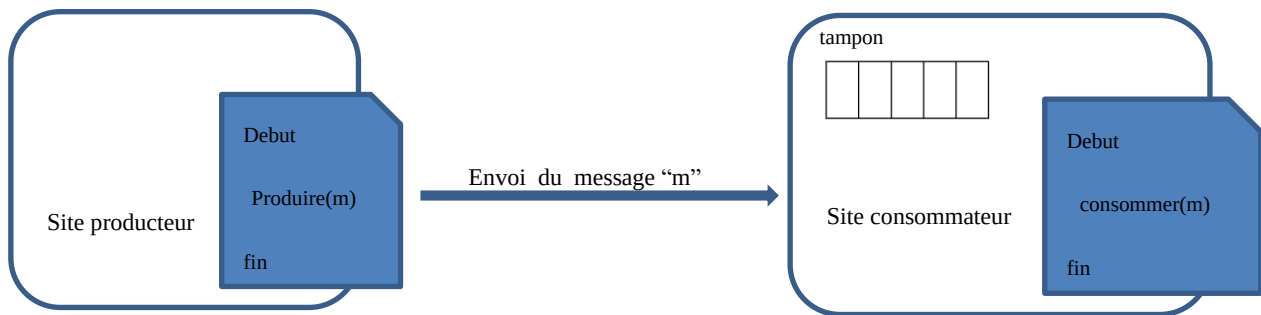
Chapitre 2: Communication

2.1 Introduction : Contrôle de flux

Contrôle point à point : le modèle producteur / consommateur

Description du problème

Considérons deux sites, P le producteur et C le consommateur, tels que le producteur envoie des messages au consommateur via un canal. Lorsque l'application du producteur veut envoyer un message au consommateur, elle appelle la primitive produire(m) où m est un paramètre par valeur de message. Lorsque l'application du consommateur veut recevoir un message du producteur, elle appelle la primitive consommer(m) où m est un paramètre par référence de message. Un problème va se produire soit le nouveau message est rejeté, soit l'un des messages stockés est écrasé par le nouvel message arrivé. Nous devons donc mettre en place un contrôle de flux pour prévenir ce problème.



Description de la solution

Afin de résoudre ce problème, le producteur disposera ainsi d'autorisations de produire (initialement égales à la taille du tampon du consommateur). A chaque envoi, celui-ci consomme une autorisation. De son côté, le consommateur envoie une autorisation à la fin de chaque appel à consommer signifiant qu'une place s'est libérée en envoyant un ACK vers le producteur, le consommateur ne peut consommer que si son tampon n'est pas vide. [Boc79]

2.1.1 Algorithme un seul producteur et un seul consommateur

Nous débutons notre spécification par les deux variables de contrôle de chacun des sites :

- Nbconsom, la variable du consommateur indiquant le nombre de messages dans le tampon
- Nbprod, la variable du producteur indiquant le nombre d'autorisations D'autre part la gestion du tampon chez le consommateur définit les variables suivantes :
- T, le tampon de taille N contenant les messages à consommer
- in : l'indice d'insertion dans le tampon
- extr : l'indice d'extraction du tampon Le tampon est géré de manière circulaire : lorsqu'un indice est en fin de tableau, il est remis à zéro.

Algorithme du producteur

Var Nbprod : entier initialisé à N;

produire(m)

Début

Attendre(Nbprod>0);

envoyer_à(C,m);

Nbprod = Nbprod - 1;

Fin

sur_réception_de(C,Ack)

Début

Nbprod = Nbprod + 1;

Fin

Algorithme du consommateur

Var Nbconsom : entier initialisé à 0;

in,extr : 0..N-1 initialisé à 0;

consommer(m)

Début

Attendre(Nbconsom > 0);

m = T[extr]; extr = (extr + 1) % N;

Nbconsom = Nbconsom - 1;

envoyer_à(P,Ack);

Fin

sur_réception_de (P,m)

Début

T[in] = m;

in = (in + 1) % N;

Nbconsom = Nbconsom + 1;

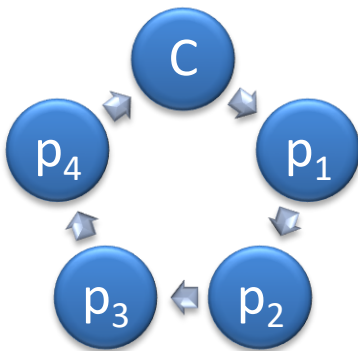
Fin

2.1.2 Généralisation aux producteurs multiples

On considère une généralisation du modèle producteur-consommateur dans lequel on a p sites de producteurs $P_1, \dots, P_p$ avec un seul site de consommation C . Comme dans le modèle précédent, le consommateur dispose d'un tampon de stockage de taille N qui repose sur un modèle client/serveur : les producteurs sont les clients et le consommateur est le serveur.

Le principal problème soulevé par ce modèle est le partage du tampon entre les différents producteurs. Comme solution à ce problème est la distribution d'autorisations sur un anneau logique Token Ring.

Cette solution consiste à faire circuler des autorisations (correspondant à des cellules disponibles) sur un anneau logique constitué de la façon suivante : $C, P_1, P_2, \dots, P_p$.



Algorithme du producteur P_i

Chaque producteur P_i maintient les variables locales suivantes :

- $Ti[0..Ni-1]$: tableau contenant les messages produits par le producteur P_i .
- ini : indice d'insertion du tampon Ti , initialisé à 0.
- $outi$: indice d'extraction du tampon Ti , initialisé à 0.
- $nbmessi$: nombre de messages stockés dans Ti , initialisé à 0.
- $nbauti$: nombre d'autorisations d'envoi de messages initialisé à 0.
- $Succi$: identificateur du site successeur du site i . Si $i < m$ alors $Succi$ prend la valeur P_{i+1} sinon la valeur de cette variable sera l'identificateur du site consommateur.
- $tempi$: variable de calcul temporaire. Elle prend la valeur minimale entre le nombre de cellules que le site désire réserver et le nombre de cellules libres associé au jeton.

Remarque : d'après la gestion de l'algorithme on aura à tout instant $\text{nbmessi} \geq \text{nbauti}$.

produire (m)

Début

```
    Attendre( $\text{nbmessi} < N_i$ )
     $T_i[\text{ini}] = m$ ;  $\text{ini} = (\text{ini} + 1) \% N_i$ ;
     $\text{nbmessi}++$ 
```

Fin

Lorsqu'un site P_i reçoit le jeton de son prédécesseur sur l'anneau, il calcule tempi , le minimum entre le nombre d'autorisations qu'il veut obtenir et la valeur associée au jeton. A partir de tempi , il met à jour la variable "val" associée au jeton. Ensuite, il expédie ce dernier à son successeur sur l'anneau.

sur_réception_de(j,(jeton,val))

Début

```
 $\text{tempi} = \text{Min}((\text{nbmessi} - \text{nbauti}), \text{val})$ ;
 $\text{nbauti} += \text{tempi}$ ;
 $\text{val} -= \text{tempi}$ ;
envoyer_à(succ, (jeton, val));
```

Fin

Comme pour la plupart des processus de service, le code de Proci_i consiste en une boucle infinie où chaque tour consiste en une attente (d'autorisations) suivi d'un traitement (l'envoi d'un message).

Proc_i

Début

```
Tant que(vrai)
    Attendre( $\text{nbauti} > 0$ );
    envoyer_à(consommateur, (app,  $T_i[\text{outi}]$ ));
     $\text{outi} = (\text{outi} + 1) \% N_i$ ;
     $\text{nbauti}--$ ;
     $\text{nbmessi}--$ ;
```

Fin tant que

Fin

Algorithme du consommateur

Le site consommateur possède les variables suivantes :

- $T[0..N-1]$: un tableau contenant les messages des sites producteurs (le tampon du consommateur).
- in : l'indice d'insertion du tampon T , initialisé à 0.
- out : l'indice d'extraction du tampon T , initialisé à 0.
- $nbmess$: nombre de messages stockés dans T et non encore consommés initialisé à 0.
- $nbcell$: nombre de cellules libérées entre deux passages du jeton initialisé à 0.

sur_réception_de($j, (app, m)$)

Début

```
T[in] = m; in = (in + 1) % N;
    nbmess++;
```

Fin

consommer(m)

Début

```
Attendre(nbmess>0);
m = T[out];
out = (out+1) % N;
nbmess--;
nbcell++;
```

Fin

Sur_réception_de($j, (jeton, val)$)

Début

```
val += nbcell;
nbcell = 0;
envoyer_à(P1, (jeton, val));
```

Fin

2.2 Communication synchrone : le rendez-vous [Bag89]

Algorithme du rendez-vous

Variables du site i

- E_i : ensemble des identités des sites avec lesquels i désire un rendez-vous.
- étati : état du site à valeur dans $\{\text{repos}, \text{encours}, \text{attente}, \text{succès}\}$ initialisé à repos
- retardé_i : identité du site retardé par i . Par convention lorsque i ne retarde aucun site, sa valeur est i (sa valeur initiale).
- $\text{jetoni}[1..N]$: tableau de booléens indiquant la présence des jetons. Initialement, $\text{jetoni}[j] = (i \geq j)$.
- candidati : identité du site candidat courant. L'interface de service se résume à la primitive $\text{RV}(E_i)$ qui doit renvoyer l'identité d'un site de E_i avec lequel le rendez-vous est engagé.

RV(E_i)

Début

$\text{étati} = \text{encours};$

 Répéter

 (1) Attendre($\exists \text{candidati} \in E_i$ et $\text{jetoni}[\text{candidati}] = \text{Vrai}$);

 Si $\text{étati} = \text{encours}$ Alors envoyer_à($\text{candidati}, \text{req}$);

$\text{jetoni}[\text{candidati}] = \text{Faux};$

$\text{étati} = \text{attente};$

 (2) Attendre($\text{étati} \neq \text{attente}$); Fsi

 Jusqu'à $\text{étati} = \text{succès};$

$\text{étati} = \text{repos};$

 renvoyer(candidati);

Fin

sur_réception_de(j,ack)

Début

étati=succès;

Si retardéi!=i Alors envoyer_à(retardéi,rej);

retardéi=i;

Fsi

Fin

sur_réception_de(j,rej)

Début

Si retardéi=i Alors étati=encours;

Sinon étati=succès;

envoyer_à(retardéi,ack);

candidati=retardéi;

retardéi=i;

Fsi

Fin

sur_réception_de(j,req)

Début

jetoni[j]=Vrai;

 Si (étati==repos) ou ($j \notin E_i$) Alors envoyer_à(j,rej);

Sinon si étati==encours Alors étati=succès;

envoyer_à(j,ack);

Sinon si (retardéi!=i) ou ($j < i$) alors envoyer_à(j,rej);

Sinon retardéi=j;

Fsi;

Fin

2.3 Qualité de service : reseau FIFO

Dans un réseau fifo , Supposons qu'une base de données soit dupliquée sur trois sites afin d'assurer une tolérance aux pannes et de diminuer les temps de réponse par répartition de la charge. Une lecture ne pose aucun problème de gestion tandis qu'une mise à jour de la base de donnée nécessite un mécanisme de synchronisation pour garantir la cohérence entre les différentes copies.

Interface du service

Le service doit émuler un réseau. Il y aura donc une primitive descendante et une primitive montante :

- `émettre_vers(j,m)` : primitive du service, appelée par l'application pour envoyer un message sur le réseau émulé
- `délivrer_de(j,m)` : primitive de l'application pour traiter les réceptions de message sur le réseau émulé, appelée par le service lorsqu'un message doit être délivré à l'application.

Variables du service

- `connaissancei[1..N,1..N]` : tableau d'entiers à deux dimensions. Initialement, tout les éléments de ce tableau sont nuls. N est le nombre de sites.
- `tamponi` : tampon des message reçus non encore délivrés initialement vide. Chaque message est stocké avec l'identité de son émetteur et le tableau envoyé avec lui.

Le tampon dispose de trois primitives :

1. `insérer(élément)` qui ajoute un nouvel élément au tampon
2. `tester(T,cond)` où T est un paramètre formel de tableau et cond, une condition portant sur T et sur `connaissancei`.
3. `extraire(élément)` qui extrait l'élément sélectionné.

Algorithme

L'émission d'un message applicatif est précédée de l'encapsulation du tableau connaissancei . Après l'envoi, celui-ci est mis à jour.

émettre_vers(j,m)

Début

envoyer_à(j,(connaissancei,m));

connaissancei[i,j]++;

Fin

La réception se limite à insérer le message dans le tampon..

sur_réception_de(j,(c,m))

Début

tamponi.insérer();

Fin

Proci

Début

Tant que(Vrai)

Attendre(tamponi.testeur(T, $\forall k, \text{connaissancei}[k,i] \geq T[k,i]$));

Tamponi.extraire();

connaissancei=Max(connaissancei,c);

connaissancei[j,i]++;

délivrer_de(j,m);

Fin tant que

Fin

Chapitre 3 Le temps

3.1 Introduction

Le temps est un concept fondamental des applications répartis et les systèmes informatiques. [Lam78]

3.2 Le temps interne

Lorsqu'un ordinateur exécute les instructions d'un ou plusieurs programmes, il s'appuie sur des durées fournies par son horloge sans qu'il y ait besoin que ce référentiel ait une cohérence avec un autre référentiel. Citons quelques exemples d'utilisation :

- Le partage du temps entre les différents processus de la machine se fait par des techniques de quantum mesurés par cette horloge.
- L'interruption horloge, qui se déclenche avec une périodicité définie à partir de cette horloge.
- Le délai de suspension d'un processus avant sa réactivation .

3.3 Le temps de l'environnement (temps réel)

On parle de temps réel pour désigner le rythme d'évolution de l'environnement. En règle générale, cette interaction se traduit par :

1. la prise en compte de signaux provenant de l'environnement,
2. l'émission de commandes permettant d'agir sur l'environnement. Par exemple, un ordinateur de bord d'un avion de chasse pourrait à l'aide de capteurs, détecter l'approche d'un missile, analyser la situation et ordonner à l'avion des actions telles que le changement de trajectoire ou l'envoi d'un missile anti-missile. Le point important du fonctionnement de l'ordinateur de bord est une suite de cycles "calcul-commandes-signaux". De plus, ces cycles sont synchrones au sens où le calcul doit être réalisé entre deux prises en compte consécutives de signaux. Un tel fonctionnement est schématisé sur la figure 3.1.

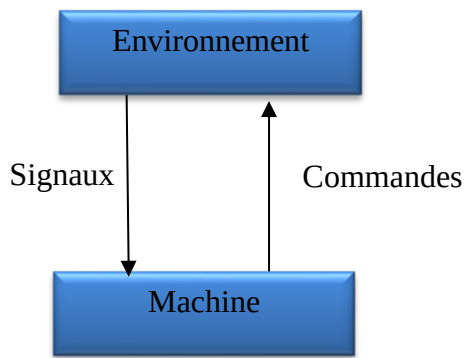


Figure 3.1 une application temps réel

3.4 Le temps universel

Les deux notions précédentes du temps reposent essentiellement sur la durée. Le temps sert aussi à dater les événements (comparaison, tri, etc.). Ainsi un utilisateur a la possibilité de trier les fichiers d'un répertoire selon la date de dernière modification. La pratique usuelle est de se référer au temps "universel" (délivré par des organismes officiels). Dans un environnement réparti, chaque processeur est doté de sa propre horloge physique qui fournit une approximation du temps universel. Dans une application répartie, il est nécessaire de synchroniser ces horloges de façon à ce que l'écart entre deux horloges soit borné à tout instant par une valeur connue. Cette borne dépendra bien sûr des incertitudes sur les délais de transfert des messages et des dérives des horloges par rapport au temps universel.

3.6 Le temps physique

3.6.1 Synchronisation d'horloges

Dans le cadre d'un système réparti où chaque site i dispose d'une horloge physique h_i . Cette horloge est un dispositif permettant à chaque instant du temps t de fournir une date $h_i(t)$ de préférence la plus proche possible de t . Nous supposons l'existence d'un référentiel universel du temps qui n'est pas accessible aux sites. La valeur de l'horloge (si elle n'est réajustée) croît linéairement en fonction du temps et nous notons C_i son coefficient de croissance. Autrement dit entre deux instants t_1 et t_2 , on a :

$h_i(t_2) - h_i(t_1) = C_i \cdot (t_2 - t_1)$ Le fournisseur des horloges garantit que la dérive des horloges vis à vis du temps universel reste borné en valeur relative par α autrement dit:

$$1 - \alpha \leq C_i \leq 1 + \alpha$$

Les horloges physiques sont difficiles à synchroniser avec précision et toutes les applications n'ont pas besoin d'avoir une mesure de temps très précise de ce fait il suffit juste savoir si un événement à eu lieu avant un autre événement d'où la notion de précédence temporelle.

3.7 Le temps logique

Dans un algorithme séquentiel, si l'exécution d'une instruction précède temporellement l'exécution d'une autre instruction, le résultat de la première peut influencer le résultat de la seconde. Chaque processus P_i a une horloge H_i

3.7.1 Les Horloges logiques

Objectif : dater les événements du système.

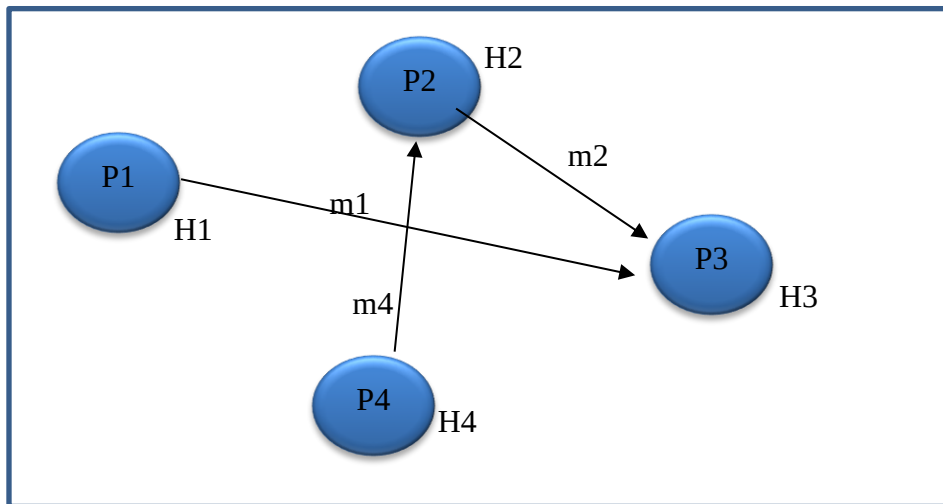
Principe : respecter les dépendances causales entre les événements.

Familles d'horloge :

- Estampille (horloge de Lamport) : une donnée par événement.
- Vectorielle (horloge de Mattern) : un vecteur par événement.
- Matricielle : une matrice par événement

Trois types d'événements signalés sur une ligne :

- Émission d'un message à destination d'un autre processus.
- Réception d'un message venant d'un autre processus.
- Événement interne (local) dans l'évolution du processus



Estampille (horloge de Lamport)

- Introduit en 1978 par Leslie Lamport.
- C'est le premier type d'horloge logique introduit en informatique.
- Une date (estampille) est associée à chaque événement.
- Localement, chaque processus P_i possède une horloge locale logique H_i , initialisée à 0.
- Sert à dater les événements.

fonctionnement de l'horloge :

Pour chaque événement local de P_i (evenement local):

- $H_i = H_i + 1$: on incrémente l'horloge locale et l'événement est daté localement par H_i .

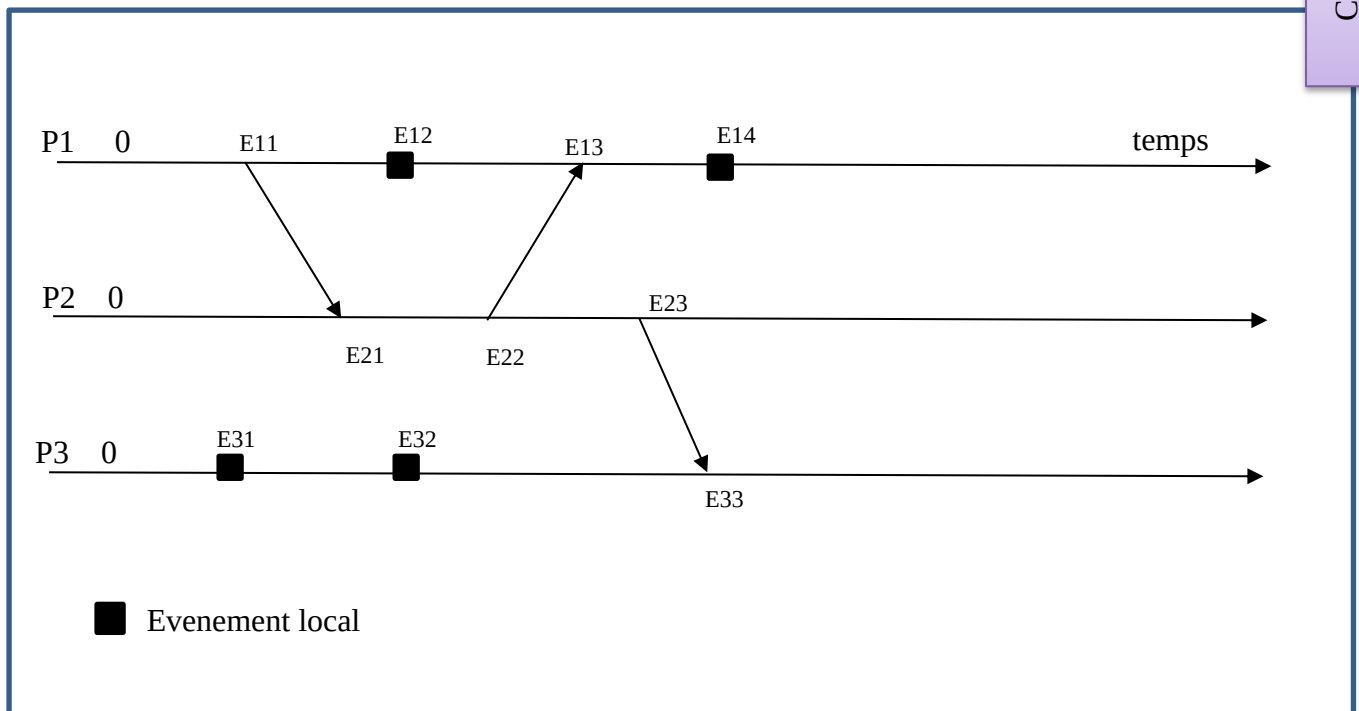
Émission d'un message par P_i (evenement d'émission):

- $H_i = H_i + 1$ puis on envoie le message avec (i, H_i) comme estampille.

Réception d'un message m avec estampille (i, nb) (evenement de reception) :

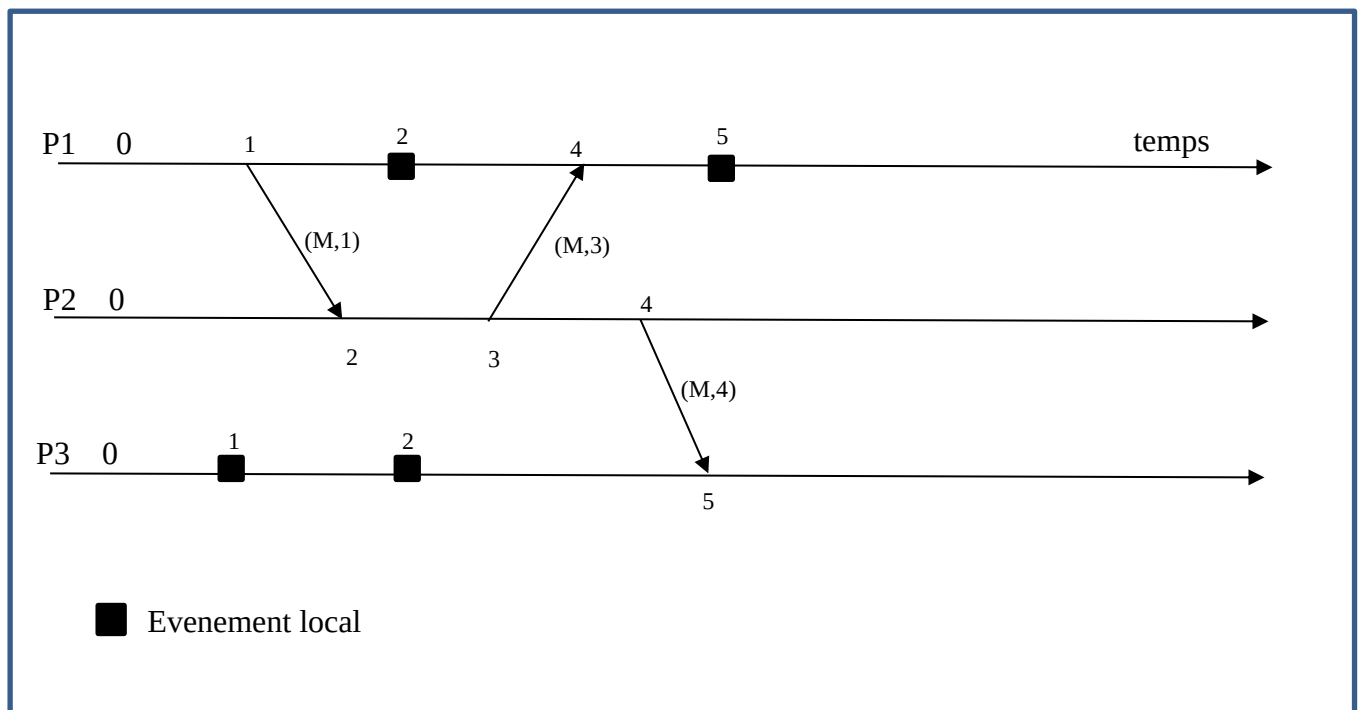
- $H_j = \max(H_j, nb) + 1$ et marque l'événement de réception avec H_j .

Exemple 1: horloge de lamport



Definir les differents evenements E_{ij} des differents processus P1,P2,P3 en utilisant le principe de l'horloge lamport?

solution



Horloge Vectorielle (horloge de Mattern)

Introduit indépendamment en 1988 par Colin Fidge et Friedemann Mattern.

Principe :

- Utilisation de vecteur V de taille égale au nombre de processus.
- Localement, chaque processus P_i a un vecteur V_i .
- Un message est envoyé avec un vecteur de date.

Fonctionnement de l'horloge :

Initialisation : pour chaque processus P_i , $V_i = (0, \dots, 0)$.

Pour un processus P_i , à chacun de ses événements (local, émission, réception) :

$$V_i[i] = V_i[i] + 1.$$

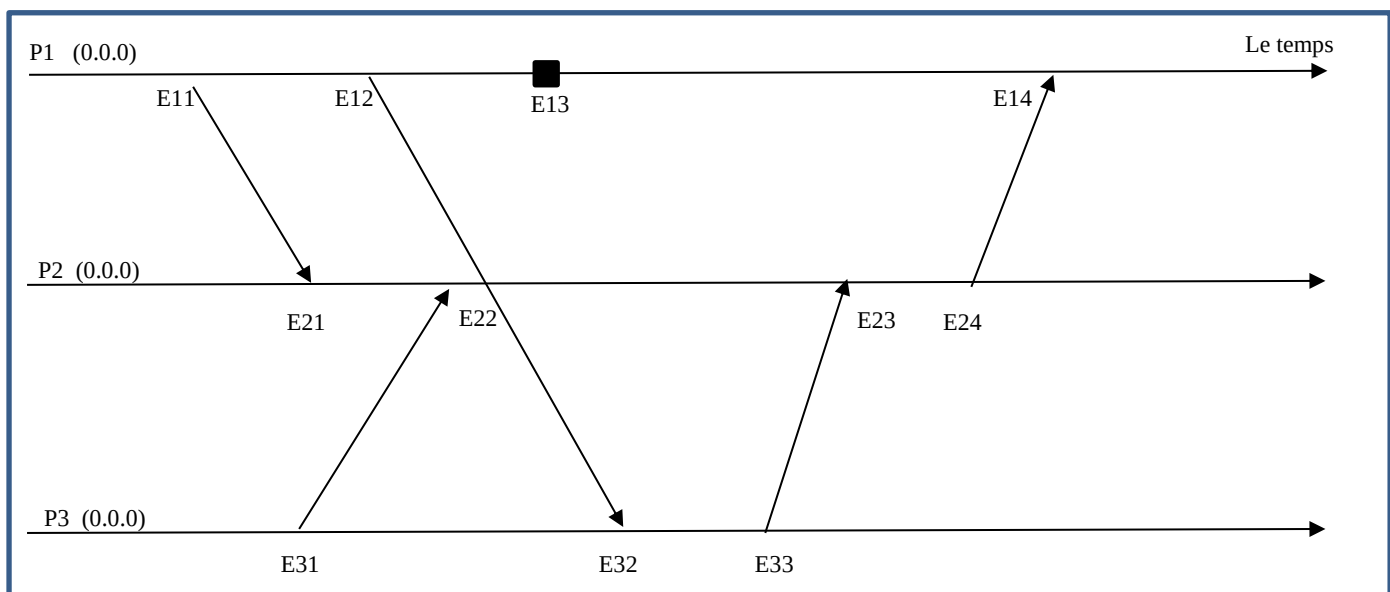
Incrémentation du compteur local d'événement (garder les autres compteurs si événement local ou émission).

Si émission d'un message, alors V_i est envoyé avec le message.

Pour un processus P_i , à la réception d'un message m contenant un vecteur V_m provenant du processus P_j , on met à jour les cases $k \neq i$ de son vecteur local V_i .

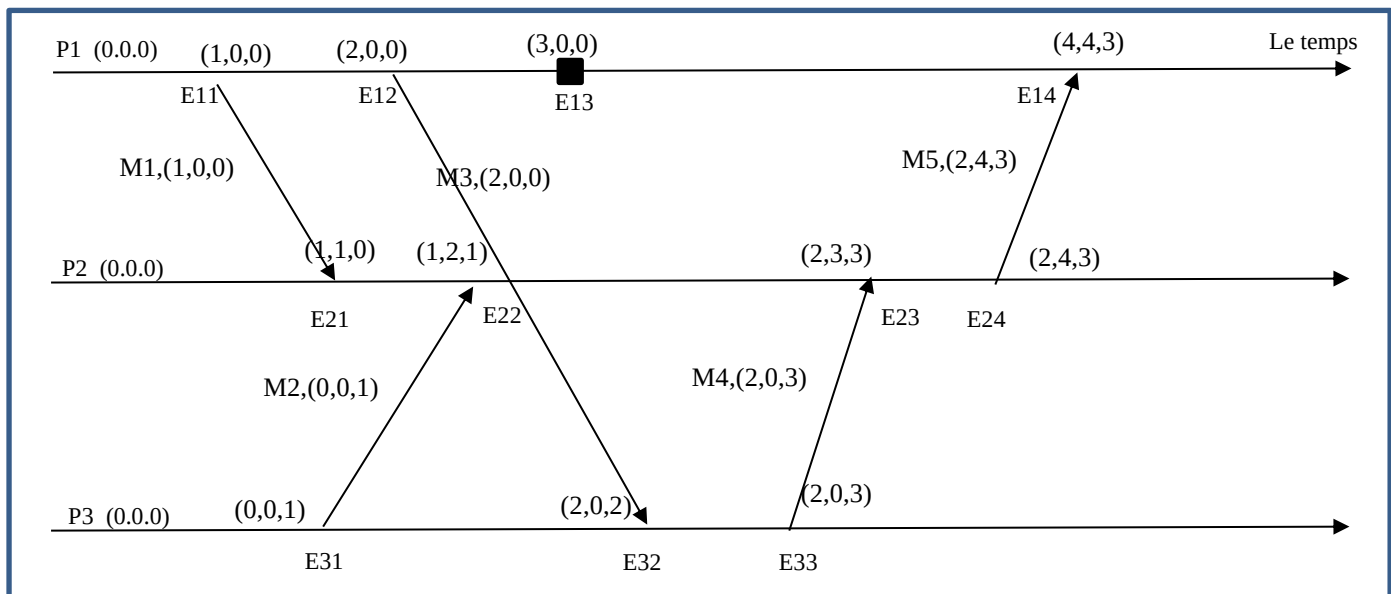
$$\forall k \neq i : V_i[k] = \max(V_m[k], V_i[k])$$

Exemple 2: horloge vectorielle



Definir les differents evenements E_{ij} des differents processus $P1, P2, P3$ en utilisant le principe de l'horloge vectorielle?

solution



Quelques explications:

Les valeurs de chaque vecteur de date est relative respectivement aux differents processus:

Dans P3, E32 le vecteur de date=(2,0,2): (2 relatif à p1, 0 relatif à p2, 2 relatif à p3)

E33 evenement d'emission dans le processus P3 ($i=3$) : E33 est un evenement dans $P_i = P3$ donc $i=3$ en appliquant l'horloge vectorielle automatiquement on incremente la Valeur relative à P_i qui est P3.

la Valeur relative à P1 à l'evenement d'emission E33 en effectuant l'incrementation de la valeur precedente relative à P3 égale à 2 du vecteur de date precedent qui E32=(2,0,2) , la valeur relative à P3 dans E32=2 en effectuant une incrementation :

E33

valeur relative à $p3=2+1=3$, le reste des valeurs respectives à P1 et p2 prennent les memes valeurs precedente relatives à p1 et p2 du vecteur de date qui precede l'evenement de date E33 qui est E32 qui sont respectivement 2 relative à p1, 0 relative à p2 donc le vecteur de date E33= (2,0,3)

E14 événement de réception dans $P_1(i=1)$:

E14 est un événement dans $P_i = P_1$ donc $i=1$ automatiquement on incrémente la Valeur relative à p_1 qui est p_1 .

valeur relative à p_1 est l'Incrementation de la valeur relative à p_1 de l'événement précédent de E14 qui est $E_{13}=(3,0,0)$

valeur relative à p_1 = valeur relative à P_3 du $E_{13} + 1 = 3 + 1 = 4$, le reste des valeurs respectives à P_2 et p_3 du E14 prennent le maximum entre les valeurs précédente relatives à p_1 et p_2 du vecteur de date qui précède l'événement de date E14 qui est E_{13} avec les valeurs relatives à p_2 et p_3 du vecteur envoyé par E_{24} du P_2 qui est $(2,4,3)$:

valeur relative à $p_2 = \max(\text{Valeur relative à } P_2 \text{ du } E_{13}, \text{Valeur relative à } P_2 \text{ du } E_{24}) = \max(3,4) = 4$

valeur relative à $p_3 = \max(\text{Valeur relative à } P_3 \text{ du } E_{13}, \text{Valeur relative à } P_3 \text{ du } E_{24}) = \max(0,3) = 3$.

Donc $E_{14}=(4,4,3)$

Horloge matricielle

n processus : matrice M de $(n \times n)$ pour dater chaque événement. Sur processus P_i :

Ligne i : informations sur événements de P_i :

$M_i[i, i]$: nombre d'événements réalisés par P_i .

$M_i[i, j]$: nombre de messages envoyés par P_i à P_j (avec $j \neq i$).

Ligne j (avec $j \neq i$) :

$M_i[j, j]$: nombre d'événements que l'on connaît sur P_j .

$M_i[j, k]$: nombre de messages que l'on sait que P_j a envoyé à P_k (avec $j \neq k$)

-Évènement local à p_i : $M_i[i, i] \leftarrow M_i[i, i] + 1$

-Évènement d'émission de m vers p_i :

$M_i[i, i] \leftarrow M_i[i, i] + 1$

$M_i[i, j] \leftarrow M_i[i, j] + 1$ m est estampillé : $EM_m = M_i$

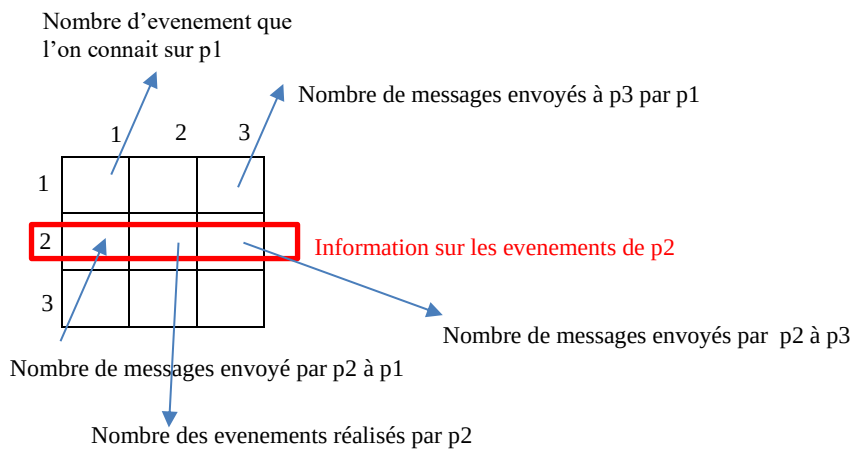
- Réception de (m, EMm) en provenance p_j :

Quand un message peut-il être délivré ?

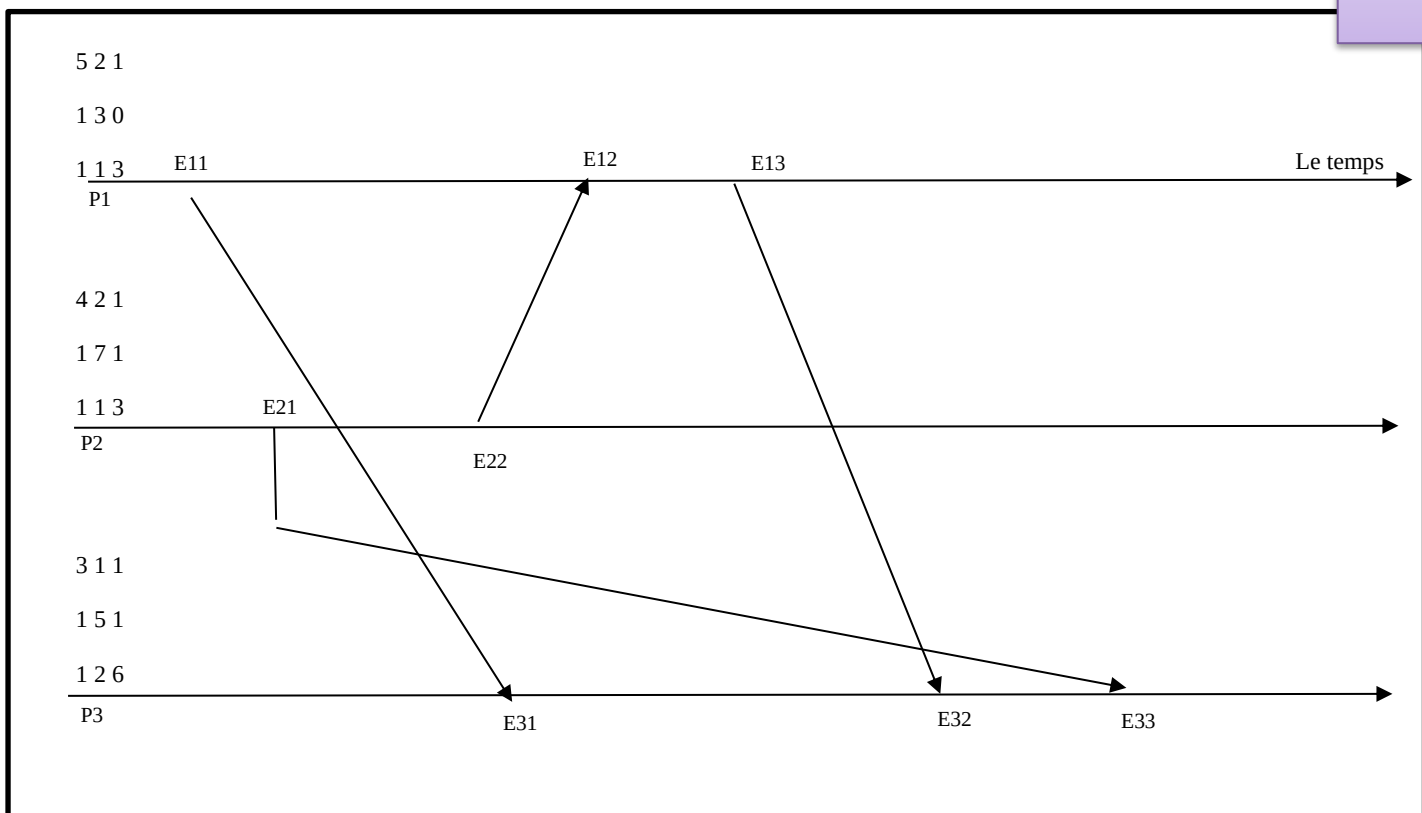
quand tous les messages antérieurs à lui ont été délivrés, i.e :

- Respect de l'ordre FIFO sur le canal $j \rightarrow i$: $EMm[j, i] = Mi[j, i]$
- Respect de l'ordre de réception : $EMm[k, i] = Mi[k, i], \forall k \neq i$
- Lors de la délivrance du message (m, EMm) en provenance de p_j , mettre à jour l'horloge
- $Mi[i, i] \leftarrow Mi[i, i] + 1$
- $Mi[i, j] \leftarrow Mi[i, j] + 1$
- pour tout $k \neq i$, pour tout $f \neq i$: $Mi[k, f] \leftarrow \max(Mi[k, f], EMm[k, f])$

Avec trois processus (sur le processus p_2) :

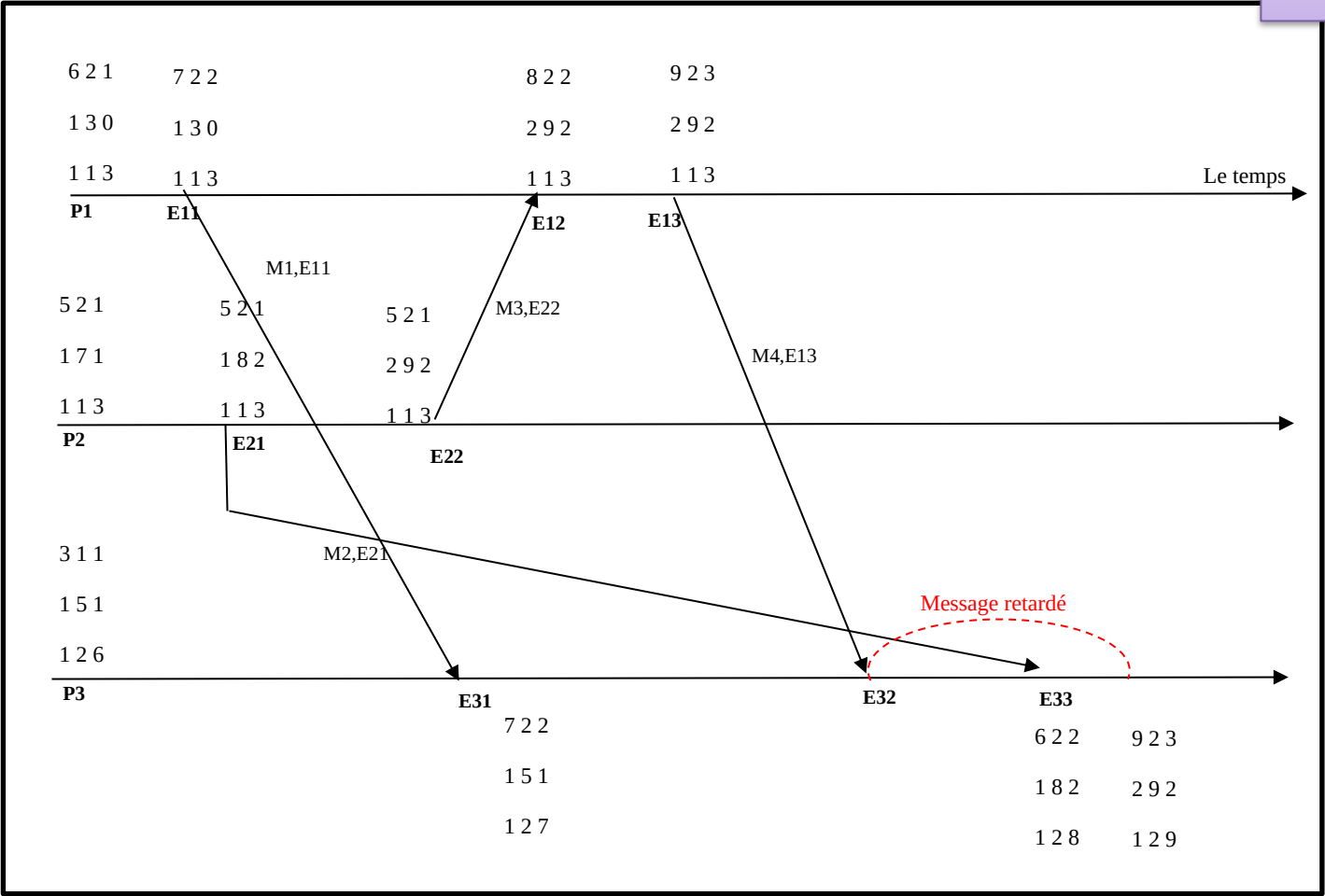


Exemple 3 : horloge matricielle



Definir les différents événements E_{ij} des différents processus $P1, P2, P3$ en utilisant le principe de l'horloge matricielle?

solution



Chapitre 4 La concurrence

4.1 Introduction

Problématique

La concurrence des processus lors de l'accès à des ressources partagées peut provoquer l'incohérence de celles-ci. Gérer la concurrence revient à contrôler que des accès concurrents s'exécutent de manière cohérente.

Afin de garantir l'exclusion mutuelle entre sections critiques, lorsqu'un processus applicatif désirera exécuter une section critique, son programme se présentera comme suit :

Verouiller();

Section critique;

déverouiller()

Verouiller et déverouiller sont des primitives du service qui devront garantir les deux propriétés caractéristiques de l'exclusion mutuelle :

- A tout instant, au plus un processus est en cours d'exécution d'une section critique.
- Tout processus demandant à exécuter une section critique (par appel à Verouiller), pourra l'exécuter (par retour de Verouiller) au bout d'un temps fini. Ce type de propriété est appelée propriété de vivacité..

4.2.Algorithme de Lamport [Lam78]

Cet algorithme repose sur le mécanisme des horloges logiques qui utilise trois types de message :

- Des requêtes d'exécution de section critique diffusées à tous les autres sites par le site demandeur,
- Des acquittements de requête pour indiquer que le site a "enregistré" la requête,
- Des libérations diffusées à tous les autres sites par un site qui a terminé l'exécution d'une section critique.

Variables du site i

- hi : variable contenant la valeur de l'horloge locale du site i. Elle est initialisée à 0.
- $Messi[1..N]$: tableau des messages. Chaque cellule de ce tableau est composée des deux champs . type prend ses valeurs parmi {req, acq, lib} et est initialisé à lib.

heure est initialisé à -1.

Algorithme du site i

Avant d'entrer en section critique, un site enregistre sa requête courante, la diffuse et attend que la requête soit le message le plus âgé du tableau.

verouiller()

Début

$Messi[i].heure = hi; Messi[i].type = req;$

$Diffuser(req, hi);$ // hi est incrémenté (par la couche réseau) $Attendre(\forall j \neq i, (Messi[i].heure, i) < (Messi[j].heure, j));$

Fin

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

déverrouiller()

Début

Diffuser(lib,hi);

Fin

La réception d'une requête provoque l'envoi d'un accusé de réception. Le message est enregistré dans le tableau excepté dans le cas discuté plus haut.

sur_réception_de(j,(tp,h))

Début

Si tp == req Alors envoyer_à(j,(acq,hi));

Finsi ;

Si (tp != acq) || (Messi[j].type != req) Alors Messi[j].heure = h;

Messi[j].type = tp;

Finsi;

Fin.

4.3.Algorithme de Ricart et Agrawala[Ric81]

Dans l'algorithme de Lamport, un acquittement s'interprète comme le fait d'enregistrer la requête. Le principal apport de ce nouvel algorithme consiste à modifier la sémantique de l'acquittement.

Variables du site i

- hi : valeur de l'horloge logique. Elle est initialisée à 0.
 - étati : état du site. Cette variable peut prendre l'une des deux valeurs {repos, en_cours}.
- Initialement, tous les sites sont dans l'état repos.
- hreqi : heure de la requête courante i
 - nbacqi : compteur des acquittements reçus.

- Retardéi[1..N] : tableau de booléens. Retardéi[j] est vrai lorsque i retarde l'acquittement d'une requête j.
- tempi : variable temporaire.

Algorithme du site i

Lorsqu'un site désire entrer en section critique, il diffuse une requête, puis se met en attente de tous les acquittements.

vrouiller()

Début

hreqi = hi;

nbacqi = 0;

Pour tempi de 1 à n faire // n est le nombre de sites

Retardéi[tempi] = Faux;

Finpour

étati = en_cours;

Diffuser(req,hi);

Attendre(nbacqi == (n-1));

Fin

A la sortie de la section critique, le site envoie des acquittements aux sites qu'il a retardés.

déverrouiller()

Début

Pour tout tempi de 1 à n faire

Si Retardéi[tempi] Alors envoyer_à(tempi,(acq,hi));

Fin

si Finpour étati = repos;

Fin

sur_réception_de(j,(req,h))

Début

Si (étati == en_cours)&&((hreqi,i) <(h,j)) Alors Retardéi[j] = Vrai;

Sinon envoyer_à(j,(acq,hi));

Finsi;

Fin

La réception d'un acquittement incrémente le compteur des acquittements reçus.

sur_réception_de(j,(acq,h))

Début

nbacqi++;

Fin

4.4 Algorithme de Carvalho et Roucairol[Car83]

Dans l'algorithme de Ricart et Agrawala, un acquittement s'interprète comme le fait d'autoriser l'entrée en section critique. Ici aussi l'apport de ce nouvel algorithme consiste à modifier la sémantique de l'acquittement.

Variables du site i

- hi : valeur de l'horloge logique. Elle est initialisée à 0.
- étati : état du site. Cette variable peut prendre l'une des deux valeurs {repos, attente,en_SC}. Initialement, tous les sites sont dans l'état repos.
- hreqi : heure de la requête courante i
- Jetoni[1..N] : tableau de booléens indiquant la présence des jetons. Jetoni[j] est initialisé à la valeur (i ≥ j).
- Retardéi[1..N] : tableau de booléens. Retardéi[j] est vrai lorsque i retarde l'acquittement d'une requête j.

- **tempi** : variable temporaire. Algorithme du site i Le site réclame les jetons qui lui manquent et attend de posséder tous les jetons pour entrer en section critique.

verouiller()

Début

$hreq_i = h_i$;

Pour $tempi$ de 1 à n faire // n est le nombre de sites

$Retardé_i[tempi] = \text{Faux}$;

Si $Jetoni[tempi] == \text{Faux}$ Alors envoyer_à($j, (req, hreq_i)$);

Finsi

Finpour

$étati = \text{attente}$; Attendre($\forall j, Jetoni[j] == \text{Vrai}$);

$étati = \text{en_SC}$;

Fin

A la sortie de la section critique, le site envoie les jetons aux sites qu'il a retardés.

déverouiller()

Début

Pour $tempi$ de 1 à n faire

 Si $Retardé_i[j]$ Alors envoyer_à($j, (acq, h_i)$);

$Jetoni[j] = \text{Faux}$;

Finsi

Finpour $étati = \text{repos}$;

Fin

Lors de la réception par le site i d'une requête émise par le site j , la décision prise par i dépend de son état :

- Si le site i n'est pas intéressé par une section critique, il envoie le jeton à j .
- Si le site i exécute sa section critique ou si sa requête est plus âgée que la requête de j , il retarde l'envoi du jeton jusqu'à la fin de sa section critique.
- Sinon il envoie le jeton à j et le lui réclame aussitôt.

sur_réception_de($j, (req, h)$)

Début

Si $étati == repos$ Alors envoyer_à($j, (acq, hi)$);

Jetoni[j] = Faux;

Sinon si ($étati == en_SC$) || (($hreq, i$) < (h, j)) Alors Retardéi[j] = Vrai;

Sinon envoyer_à($j, (acq, hi)$);

Jetoni[j] = Faux;

envoyer_à($j, (req, hreqi)$);

Finsi

Fin

sur_réception_de($j, (acq, h)$)

Début

Jetoni[j] = Vrai;

Fin

Chapitre 5 L'OBSERVATION

5.1.Introduction

Problématique

Au cours des chapitres précédents, nous avons perçu la nécessité de raisonner sur l'état d'une application répartie. Lors de la construction d'un arbre de plus courts chemins, il est indispensable de détecter la terminaison de la construction pour débiter la transaction répartie au dessus de cet arbre. D'autre part, une manière de rendre tolérante aux fautes une application répartie consiste à construire des points de reprise de telle sorte qu'après une panne, l'application puisse redémarrer dans un état le plus proche possible de celui qu'elle avait au moment de la panne. Autrement dit, les services proposés dans ce chapitre sont de l'ordre de l'observation. Il s'agit d'obtenir une information sur l'état de l'application sans perturber, contraindre ou stopper le déroulement de l'application en définissant un état global.

Définition d'un état global cohérent :

Un état global est donné par un état de l'application sur chaque site i à un instant t_i et un état des canaux. Un état global sera cohérent si et seulement si : Tout message émis par le site j et reçu par le site i avant t_i a été émis avant t_j . L'état du canal de i vers j contient - dans l'ordre d'émission – les messages émis par j avant t_j et reçus par i après t_i . La première condition exprime le fait qu'un message doit être émis pour être reçu et la deuxième condition exprime le fait que les canaux contiennent les messages émis et non encore reçus

5.2 Algorithme de construction d'un état global coherent [Cha85]

les primitives utilisés :

- enregistrer() appelée par le service pour enregistrer l'état courant de l'application.
- délivrer_de(j, m) appelée par le service pour délivrer un message d'application m provenant de j .
- construire() appelée sur l'initiateur par l'application (ou une autre couche) pour construire un état global de l'application.
- émettre_vers(j, m) appelée par l'application pour émettre un message m vers l'application de j

Variables du site i

- si : variable contenant l'état local de l'application du site i .
- $canali[1..N]$: tableau de suites de messages. Chaque cellule j de ce tableau (exceptée la cellule i) correspond à l'état du canal de j vers i dans l'état construit.
- $étati$: (avant,pendant,après) état local du service initialisé à avant.
- $àenregistreri[1..N]$: tableau de booléens. Indique si un message venant d'un canal doit être enregistré pour compléter l'état global. (N est le nombre de sites)
- $nbconsi$: nombre de messages de construction reçus.
- $tempi$: variable temporaire

Algorithme du site i

La construction consiste à initialiser les variables et diffuser un message de construction aux autres sites.

construire()

Début

$si = enregistrer();$

$étati = pendant;$

$nbconsi=0;$

Pour $tempi$ de 1 à N faire

$canali[tempi] = \emptyset;$

$àenregistreri[tempi] = \text{vrai};$

Fin pour

$Diffuser(cons);$

Fin

L'émission d'un message d'application est accompagnée d'une encapsulation.

émettre_vers(j,m)

Début

envoyer_à(j,(app,m));

Fin

A la réception d'un message de construction, on débute éventuellement la construction puis on incrémente le nombre de messages de construction reçus et on "clôt" l'enregistrement du canal correspondant. A la réception du dernier message de construction, la construction est terminée.

sur_réception_de(j,cons)

Début

Si étati == avant Alors construire();

Fin

si nbconsi++;

àenregistreri[j]= faux;

Si nbconsi == N-1 Alors étati=après;

Fin si

Fin

La réception d'un message d'application provoque sa délivrance éventuellement accompagnée d'un enregistrement dans le canal correspondant.

sur_réception_de(j,(app,m))

Début

Si àenregistreri[j] == vrai Alors canali[j]=canali[j]& m; // concaténation du message Fin si
délivrer_de(j,m);

Fin

5.3 Détection de propriétés stables et paisibles [Cha86]

Propriétés stables

Nous appelons π la propriété stable, $\pi(t)$ la valeur de π à l'instant t et $\pi(s)$ la valeur de π dans l'état s et π^* évaluation de la valeur de π .

Une propriété stable évolue de la manière suivante :

- Elle est initialement fausse.
- Soit elle reste indéfiniment fausse, soit elle devient vrai et le reste indéfiniment.

L'algorithme de construction d'état global pour détecter une propriété stable, nous allons considérer un site i_0 qui se chargera de l'évaluation de la propriété. Le site i_0 lance périodiquement une construction d'état global. Une fois celle-ci terminée, les sites transfèrent leur partie d'état global vers i_0 . Appelons s^* l'état global construit, calcule $\pi(s^*)$ pour mettre à jour π^* . Si π^* devient vrai, il n'est plus nécessaire de continuer à construire l'état global.

Algorithme

Début

Répéter périodiquement

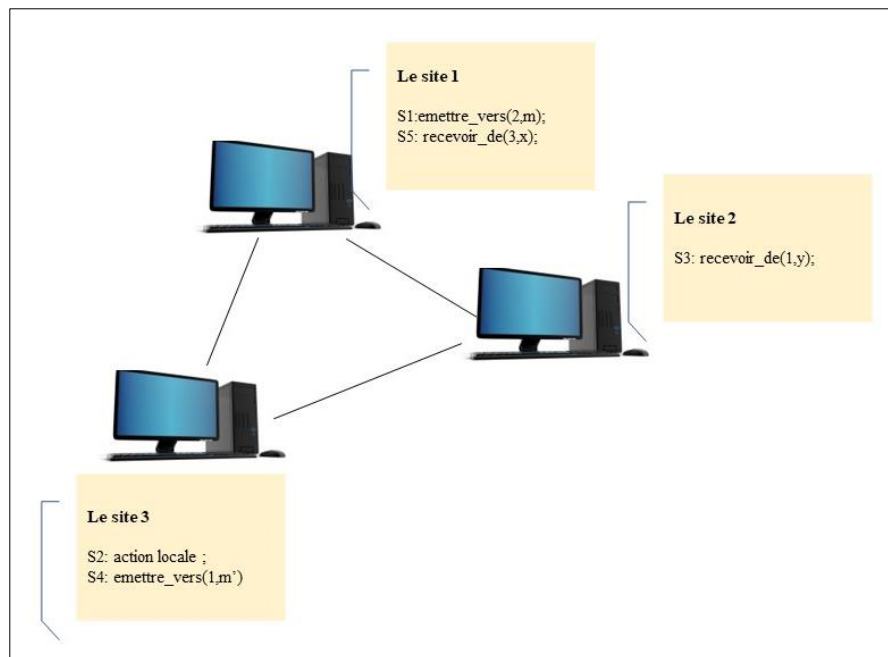
- (1) on initie une construction d'état global s^*
- (2) les sites transfèrent leur partie de s^* vers i_0
- (3) calcule $\pi(s^*)$ et $\pi^* = \pi(s^*)$ Jusqu'à $\pi^* = \text{vrai}$;

Fin

A partir de la construction d'un état global de la section précédente, on montrera comment obtenir un algorithme de détection de telles propriétés. Cependant cette nouvelle construction n'est que partiellement répartie. En spécialisant le concept de propriété stable à l'aide de celui de propriété paisible, on sera en mesure d'établir un algorithme de construction entièrement réparti qui, de plus, ne nécessite pas la construction d'un état global.

Exemple:

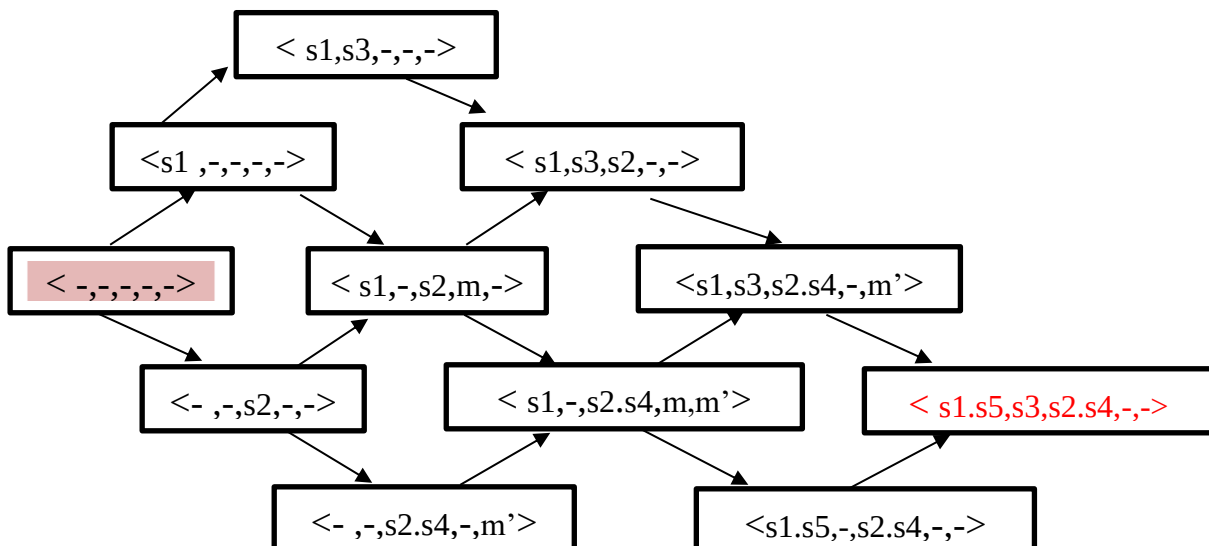
Supposons que l'application répartie soit composée de trois (03) sites dont le code source respectif pour chaque site est le suivant :



- Définissez les différents états possibles (S1 jusqu'à S5) ainsi que l'état global cohérent de cette application répartie?

Solution:

le vecteur vide est l'état initial, les premiers éléments du vecteur décrivent les états de chaque site qui exécute ses instructions d'une manière séquentielle et les derniers éléments du vecteur représentent les différents messages qui se trouvent dans le canal de communication. Le dernier vecteur de l'arbre est l'état global cohérent.



Chapitre 6: L'élection

6.1 Introduction

Problématique

Quelques stations d'une application répartie ont des fonctionnalités similaires, certaines tâches spécifiques doivent être réalisées par une station unique. Avant de démarrer l'application, les sites doivent se mettre d'accord sur l'identité de cette station. Cette opération est communément appelée élection et fait l'objet du présent chapitre.

6.2 Algorithme de Chang et Roberts[Cha79]

Principe et réalisation de l'algorithme

Cet algorithme s'applique à un anneau unidirectionnel. Chaque site n'émet des messages que vers le site suivant sur l'anneau. Lorsque l'application « réclame » l'identité du leader, trois cas se présentent :

- Le processus d'élection est terminé et le service renvoie immédiatement l'identité du site élu.
- Le processus d'élection est en cours et connu du service. Dans ce cas, le service attend la terminaison du processus afin de renvoyer l'identité du leader.
- Le service n'est pas au courant d'un processus d'élection engagé. Dans ce cas, il envoie une requête circuler sur l'anneau afin de devenir le leader. Si la requête lui revient, il conclut que son offre est acceptée et envoie un message de confirmation et fait circuler sur l'anneau pour indiquer aux autres sites qu'il est le leader.

Lorsqu'une requête parvient à un autre site, deux cas se présentent :

- Le service du site était au repos ou avait provisoirement choisi un leader d'identité supérieure; il adopte l'initiateur de la requête comme nouveau leader potentiel et retransmet la requête au site suivant de l'anneau.
- Le service du site avait provisoirement choisi un leader d'identité inférieure; il ne donne pas suite à la requête.

Variables du site i

- suivant_i : constante contenant l'identité du site successeur de i sur l'anneau.
- état_i : état du service. Cette variable prend une valeur parmi l'ensemble des valeurs (repos, en_cours, terminé). Cette variable est initialisée à repos.
- chef_i : identité du site élu.

Algorithme du site i

Si aucun processus d'élection n'a atteint le site i , le service initialise un tel processus. Dans tous les cas, il attend que le processus d'élection soit terminé pour renvoyer l'identité du site élu.

leader()

Début

Si ($\text{état}_i == \text{repos}$) Alors $\text{état}_i = \text{en_cours}$;

$\text{chef}_i = i$;

envoyer_à($\text{suivant}_i, (\text{req}, i)$);

Finsi

Attendre($\text{état}_i == \text{terminé}$);

renvoyer(chef_i);

Fin

Si le site est au repos, la réception d'une requête provoque le changement d'état et la retransmission de la requête. Dans le cas où le processus reçoit une "meilleure" requête, il en tient compte et retransmet aussi la requête. Enfin si une requête parvient à son initiateur, ce site est élu et il informe les autres sites.

sur_réception_de(j,(req,k))

Début

Si (étati==repos || k<chefi) Alors étati=en_cours;

chefi=k;

envoyer_à(suivanti,(req,k));

Sinon si (i==k) Alors étati=terminé;

envoyer_à(suivanti,(conf,i));

Finsi;

Fin

La confirmation du site élu fait un tour de l'anneau.

sur_réception_de(j,(conf,k))

Début

Si (i!=k) Alors envoyer_à(suivanti,(conf,k));

étati=terminé;

Finsi ;

Fin

6.3. Algorithme de Franklin[Fra82]

Principe de l'algorithme

Cet algorithme s'applique à un anneau bidirectionnel. Il se décompose en "tours" où les compétiteurs sont les initiateurs (au même sens que dans l'algorithme précédent).

Variables du site i

- suivanti : constante contenant l'identité du site successeur de i sur l'anneau.
- précédenti : constante contenant l'identité du site prédécesseur de i sur l'anneau.
- étati : état du service. Cette variable prend une valeur parmi (repos, en_cours, attente, terminé). Cette variable est initialisée à repos.

- nbreqi : nombre de requêtes reçues au cours d'un tour sur l'anneau.
- conci : identité du site dont on reçoit la première des deux requêtes du tour courant. Lorsqu'elle vaut i, cela signifie qu'aucune requête en avance n'est reçue.
- diri : booléen indiquant la direction d'où vient la première des deux requêtes du tour courant
- concavi : identité du site dont on reçoit la première des deux requêtes du tour suivant
- chefi : identité du site (provisoirement) élu.

Algorithme du site i

leader()

Début

Si (étati==repos) Alors étati=en_cours;

chefi=i;

concavi=i;

Répéter nbreqi=0;

Si concavi!=i Alors nbreqi=1;

conci=concavi; Si (conci<chefi) Alors chefi=conci;

Finsi;

concavi=i;

Finsi;

envoyer_à(suivanti,(req,i));

envoyer_à(précédenti,(req,i));

Attendre(nbreqi==2);

Jusqu'à(étati !=en_cours);

Si concavi!=i Alors Si (diri) Alors envoyer_à(suivanti,(req,concavi));

Sinon envoyer_à(précédenti,(req,concavi));

Finsi;

Finsi;

Finsi;

Attendre(étati==terminé);

renvoyer(chefi);

Fin

sur_réception_de(j,(req,k))

Début

Si (étati == repos || k<chefi) Alors chefi = k;

Finsi

Si (étati == en_cours) Alors Si (nbreqi==0) Alors concik=k;

nbreqi=1;

diri=(j==précédenti);

Sinon si ((diri&&j==précédenti)|| (!diri&&j!=précédenti)) Alors concavi=k;

Sinon nbreqi=2;

Si (chefi<i) Alors étati=attente;

Sinon

si (conci==i||k==conci) Alors étati = terminé;

envoyer_à(suivanti,(conf,i));

Finsi

Finsi Sinon étati=attente; Si (j==précédenti) Alors envoyer_à(suivanti,(req,k));

Sinon envoyer_à(précédenti,(req,k));

Finsi Finsi Fin

La confirmation du site élu fait un tour de l'anneau.

sur_réception_de(j,(conf,k))

Début

Si (i!=k) Alors envoyer_à(suivanti,(conf,k));

étati=terminé;

Finsi ;

Fin

Chapiter 7 MÉMOIRE VIRTUELLE RÉPARTIE

7.1 Introduction

Problématique

L'idée derriere la memoire virtuelle répartie est de fournir au programmeur l'illusion de travailler avec des processus partageant un espace mémoire unique.

Quelque suppositions:

- Nous supposons que cet espace est structuré en objets (ou variables partagées)
- Nous supposons que ces objets sont statiques (c'est à dire connus à l'initialisation de l'application)
- La structure des objets est sans importance pour les algorithmes. Aussi nous ne préciserons pas le type des objets de la mémoire répartie
- Le programmeur dispose des seules opérations de lecture et d'écriture.

Ces operations engendrent deux problemes:

- La sémantique des accès concurrents à la mémoire doit être formellement définie. En effet, en raison de l'activité parallèle des processus, plusieurs exécutions d'un même programme sont possibles.
- Une fois cette sémantique définie, il faut construire un "protocole" qui garantit que cette sémantique est préservée. Il s'agit en fait d'un algorithme réparti associé au service d'accès à la mémoire virtuelle répartie.

Le service que nous réaliserons a une interface constituée de deux opérations :

- **lire(x)** qui renvoie le contenu de l'objet x .
- **ecrire(x,v)** qui met à jour l'objet x avec la valeur v.

Deux sémantiques sont définies et mises en oeuvre dans la suite du chapitre :

1. La linéarisabilité
2. La consistance séquentielle (algorithme de propagation des ecritures).

7.2 La linéarisabilité [Her90]

Dans la sémantique linéarisabilité

- Les algorithmes ne s'appliquent qu'aux objets propriétaires
- Chaque objet est modifié par un seul processus et peut être lu par tous les processus.

Ce cas particulier se rencontre assez fréquemment dans les technologies logicielles (CORBA, JAVA RMI,...). Dans le cas de la consistance séquentielle, nous nous limitons à un protocole s'appliquant aux objets propriétaires.

Dans l'exemple représenté à la figure 7.1, on note $r_i(x):v$ une instance d'appel à lire(x) qui a renvoyé la valeur v et $w_i(x,v)$ une instance d'appel à écrire(x,v). L'indice nous sert à identifier de manière unique l'instance de l'appel.

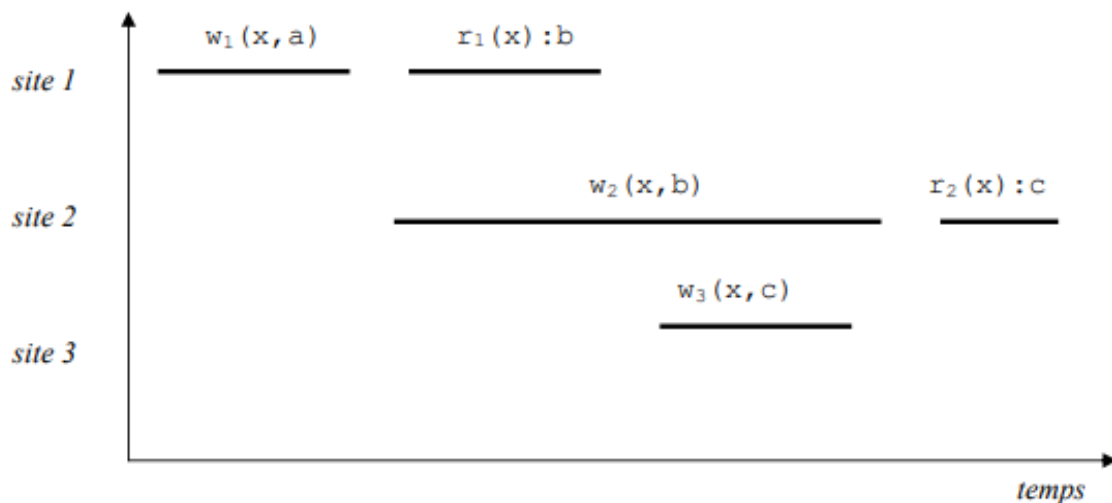


Figure 7.1 Une exécution linéarisable [Her90]

Figure 7.1 : Une exécution linéarisable Afin d'étudier à quelles conditions l'exécution représentée ci-dessus pourrait être une exécution "acceptable", nous introduisons une définition préalable qui prend en compte le fait que les actions ne sont pas instantanées.

Définition (précédence temporelle)

Soient deux opérations sur la mémoire répartie $op1$ et $op2$. On dit que $op1$ précède temporellement $op2$ si et seulement si la fin de l'opération $op1$ précède le début de l'opération $op2$. On notera cette précédence temporelle par $op1 \prec_t op2$.

Appelons deux opérations, des opérations concurrentes si aucune d'entre elles ne précède temporellement l'autre.

Définition linéarisabilité

Une exécution répartie est linéarisable si et seulement si il existe une relation d'ordre total sur les opérations notée $<$ qui étend la précédence temporelle et qui vérifie : Pour toute lecture $ri(x):v$, il existe une écriture $wj(x,v) < ri(x):v$ et toute autre écriture $wk(x,v')$ avec $v' \neq v$ vérifie :

- soit $wk(x,v') < wj(x,v)$
- soit $ri(x):v < wk(x,v')$ Autrement dit, une exécution est linéarisable, si on peut ordonner l'ensemble des opérations en respectant la précédence temporelle et de telle sorte qu'une lecture renvoie la dernière valeur écrite selon cet ordre.

L'exécution est de la figure 7.1 est linéarisable et équivalente à l'exécution séquentielle : $w1(x,a);w2(x,b);r1(x):b;w3(x,c);r2(x):c$

Les différents algorithmes pour assurer la linéarisabilité

1. Linéarisabilité par exclusion mutuelle
2. Lecture à la demande
3. Linéarisabilité avec gestionnaire statique
4. Linéarisabilité avec gestionnaire dynamique

7.2.1 Linéarisabilité par exclusion mutuelle

Principe de l'algorithme

Avant de présenter le premier protocole qui garantit la linéarisabilité de l'exécution, nous soulignons d'abord une propriété essentielle de cette sémantique qui facilite le développement de protocoles associés. Supposons que nous ayons un protocole qui garantit que, relativement aux opérations sur chaque objet, une linéarisation est possible, alors ce protocole garantit la linéarisation globale. En effet, si on pose $<_x$ la relation d'ordre total relative aux opérations sur un objet x obtenue par le protocole, on construit une relation d'ordre total sur toutes les opérations comme suit :

- on définit une relation d'ordre total entre tous les objets $x_1, \dots, x_n$
- appelons E , l'ensemble des opérations, on sélectionne E_0 l'ensemble des éléments minimaux de $(E, <_t)$ qu'on ordonne selon deux critères :

- l'objet sur lequel porte l'opération (d'abord les opérations sur x_1 , puis sur x_2 , etc.)
 - pour trier les opérations sur un objet x , on suit l'ordre total $<_x$
- on sélectionne ensuite les éléments minimaux de $(E \setminus E_0, <_t)$ et on réitère le procédé. On construit ainsi par récurrence une relation d'ordre total (car toute opération n'a qu'un nombre fini de prédécesseurs pour $<_t$). Afin d'obtenir de la linéarisation, il suffit de se focaliser sur la gestion de chaque objet de manière indépendante.

Le premier algorithme garantit la linéarisabilité de manière triviale en assurant l'exclusion mutuelle entre les opérations sur un même objet. Ainsi la relation $<_t$ est une relation d'ordre totale sur les opérations de chaque objet et la propriété précédente nous assure la linéarisabilité. N'importe quel algorithme d'exclusion mutuelle convient (voir le chapitre 4).

Nous organisons les sites en anneau, et nous utilisons l'algorithme à jetons avec un jeton qui circule par objet et contient son identité et sa valeur.

Description Variables du site i

- $suivanti$: constante contenant l'identité du site successeur de i sur l'anneau.
- $vali$: variable contenant une valeur transmise par un jeton
- $attendui$: variable contenant l'identité de l'objet dont attend le jeton. Cette variable est initialisée à NULL indiquant qu'on n'attend pas de jeton

Algorithme du site i

Pour lire la valeur d'un objet, on attend le jeton et on renvoie la valeur après avoir envoyé le jeton au suivant sur l'anneau.

lire(x)

Début

```
attendui=x;

Attendre(attendui==NULL);

envoyer_à(suivanti,(x,vali));

renvoyer(vali);
```

Fin

Pour modifier un objet,, on attend le jeton et on envoie le jeton au suivant sur l'anneau après avoir modifié sa valeur.

ecrire(x,v)

Début

attendui=x;

Attendre(attendui==NULL);

envoyer_à(suivanti,(x,v));

Fin

A la réception d'un jeton, si l'objet associé n'était pas attendu, on envoie le jeton au suivant sur l'anneau sinon on indique que le jeton est arrivé et la valeur de cet objet.

sur_réception_de(j,(x,v))

Début

Si (attendui != x) Alors envoyer_à(suivanti,(x,v));

Sinon vali = v;

attendui=NULL;

Finsi

Fi

7.2.2 Lecture à la demande

Principe et réalisation de l'algorithme

Dans cet algorithme, nous faisons les hypotheses suivantes :

- Les objets ne sont modifiés que par un seul site appelé le propriétaire de l'objet.
- Lorsqu'un site appelle la primitive lire() sur un objet distant, il envoie une demande de valeur au site propriétaire qui à la réception lui envoie la valeur demandée par contre la modification de l'objet est purement local par le propriétaire de l'objet. Cet algorithme est présenté afin de servir de base à une version auto-stabilisante (voir chapitre 8).

Variables du site i

- vali : variable temporaire qui contient la valeur demandée lors d'un lire() distant.
 - xi : variable associée à l'objet partagée x. Le site i a autant de variables de ce type que d'objets dont il est propriétaire.
- étati : état du service à valeurs dans (repos, attente) initialisé à repos.
 - propi[Objets] : constante qui contient l'identité du propriétaire de chaque objet.

Algorithme du site i

A la lecture d'un objet, soit on est le propriétaire et on renvoie immédiatement la valeur, soit on envoie une demande au propriétaire et on renvoie la valeur fournie par le propriétaire.

lire(x)

Début

Si propi[x]==i Alors renvoyer(xi);

Sinon étati=attente;

envoyer_à(propri[x],(req,x));

Attendre(étati==repos);

renvoyer(vali);

Finsi

Fin

La modification d'un objet est purement locale.

ecrire(x,v)

Début

xi = v;

Fin

la réception d'une requête, on renvoie la valeur demandée.

sur_réception_de(j,(req,x))

Début

envoyer_à(j,(acq,xi));

Fin

A l'acquittement, on enregistre la valeur et on modifie l'état du service.

sur_réception_de(j,(acq,v))

Début

vali = v;

étati=repos;

Fin

Inconvenient de l'algorithme lecture à la demande:

Cet algorithme interdit la lecture simultanée de ce fait la Linéarisabilité avec gestionnaires statiques et dynamique resout ce probleme et donne lieu a la gestion des lectures concurrentes.

7.2.3 Linéarisabilité avec gestionnaires statiques [Li89]

Principe de l'algorithme

On constate que l'algorithme précédent interdit des lectures simultanées . Nous allons donc mettre en œuvre *une gestion des lectures concurrentes*.

Hypotheses:

-Nous supposons qu'un site peut s'envoyer un message à lui-même..

-Les objets sont mobiles et éventuellement dupliqués (cas des lectures simultanées). Il faut donc un moyen pour un site de repérer l'objet.

- Nous supposons qu'à chaque objet est associé un gestionnaire statique par objet connu de tous. Son identité peut se calculer par exemple à l'aide d'une fonction de hachage appliquée à l'identité de l'objet.

- Nous supposons que chaque site dispose d'une table qui lui indique pour chaque objet son gestionnaire (gesti). Les informations qu'un gestionnaire possède sur un objet x sont contenues dans la cellule $\text{infosi}[x]$. Elle comporte d'abord l'identité du site qui acquies l'objet le plus anciennement appelé par le propriétaire ($\text{infosi}[x].\text{prop}$), puis l'ensemble des autres sites qui ont une copie de l'objet ($\text{infosi}[x].\text{ontcopie}$).

Enfin puisque des requêtes concurrentes peuvent parvenir au gestionnaire, celui dispose d'une file qui contient les requêtes à traiter dont la requête en cours ($\text{infosi}[x].\text{àtraiter}$). Chaque site dispose d'une mémoire virtuelle locale qui peut potentiellement contenir l'ensemble des objets du système ($\text{memi}[x].\text{val}$). Cependant chaque site doit connaître le statut de la cellule relatif à chaque objet : objet invalide, accès en lecture seule ou accès en lecture-écriture ($\text{memi}[x].\text{statut}$). Le lecteur aura compris que la linéarisabilité est assurée par le recours au gestionnaire (rappelons qu'il suffit que les accès à chaque objet soient linéarisables indépendamment des accès aux autres objets).

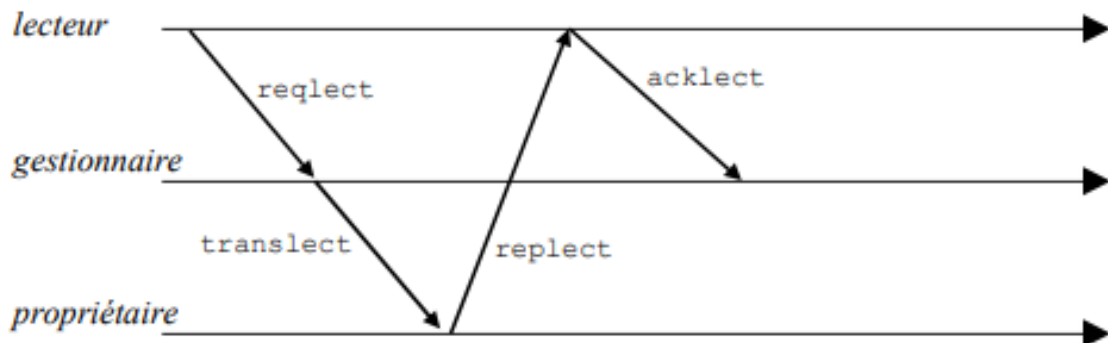


Figure 7.2 Flux des messages lors d'une requete de lecture [Li89]

le flux des messages engendrés par une lecture et par une écriture lorsque le site ne dispose pas du droit correspondant:

Un site qui désire obtenir le droit de lire pour devenir un site lecteur envoie une requête de lecture (*reqlect*) au gestionnaire. Si celui-ci ne traite pas une autre requête sur cet objet, il transmet la requête au propriétaire qui modifie son accès à l'objet (lecture seule) et renvoie au lecteur l'objet (*replect*). A la réception de l'objet, celui-ci indique au gestionnaire qu'il dispose de la copie

(acklect). Après mise à jour des informations, le gestionnaire peut éventuellement traiter une autre requête sur cet objet.

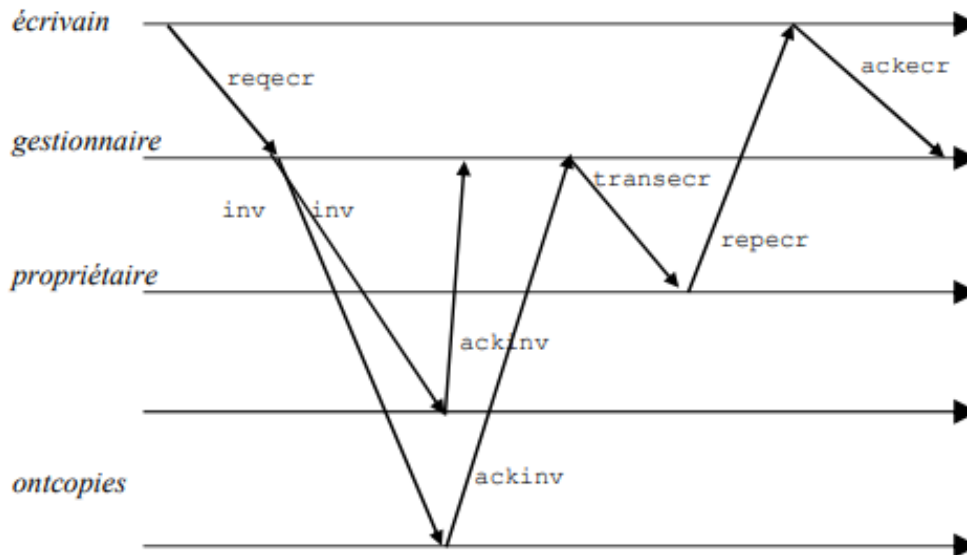


Figure 7.3 Flux des messages lors d'une requête d'écriture [Li89]

Un site qui désire obtenir le droit d'écrire pour devenir un site écrivain envoie une requête d'écriture (reqecr) au gestionnaire. Si celui-ci ne traite pas une autre requête sur cet objet, il envoie des invalidations aux sites qui possèdent la copie (inv) et attend leur acquittement (ackinv). Puis il transmet la requête au propriétaire qui invalide son accès à l'objet et renvoie à l'écrivain le droit d'accès (repecr). A la réception de celui-ci, le demandeur indique au gestionnaire qu'il dispose du droit d'écrire (ackecr). Après mise à jour des informations, le gestionnaire peut éventuellement traiter une autre requête sur cet objet.

Description Variables du site *i*

- `gesti[Objets]` : constante qui contient l'identité du gestionnaire de chaque objet.
- `Objetsi` : ensemble constant des objets gérés par *i*.
- `infos[Objetsi]` de `prop` : propriétaire courant de l'objet initialement *i*

`ontcopie` : ensembles des autres sites qui ont une copie de l'objet initialement vide à traiter : file de couples initialement vide (primitives associées : `enfiler()`, `défiler()`, `tête()`)

- `memi[Objets]` de `val` : valeur de l'objet initialisé à la valeur initiale de l'objet s'il est affecté à `i`
statut : (`invalide`,`lect`,`ecr`) statut de l'objet, `ecr` pour un objet affecté, `invalide` pour un objet non affecté.
- `tempi` : variable temporaire d'identité de site
- `requi` : variable temporaire de type de requête

Algorithme du site `i`

En cas de lecture d'un objet invalide, on émet une requête au gestionnaire de l'objet et on attend que l'objet soit disponible en lecture.

lire(x)

Début

Si (`memi[x].statut==invalide`) Alors `envoyer_à(gesti[x],(reqlect,x));`
`Attendre(memi[x].statut==lect);`

Finsi

`renvoyer(memi[x].val);`

Fin

A la réception d'une requête de lecture, si une requête n'est pas en cours on transfère la requête au propriétaire courant de l'objet. Dans tous les cas, on met à jour la file des requêtes à traiter.

sur_réception_de(j,(reqlect,x))

Début

Si `infosi[x].àtraiter==vide` Alors `envoyer_à(infosi[x].prop,(translect,j,x));`

Finsi

`enfiler(infosi[x].àtraiter,);`

Fin

A la réception d'un transfert d'une requête de lecture, le propriétaire de l'objet envoie une réponse au demandeur et met à jour le statut en lecture.

sur_réception_de(j,(translect,k,x))

Début

```
memi[x].statut=lect ;
```

```
envoyer_à(k,(relect,x,memi[x].val));
```

Fin

A la réception de la réponse de lecture, la valeur et le statut de cet objet sont mis à jour et un acquittement est envoyé au gestionnaire.

sur_réception_de(j,(relect,x,v))

Début

```
memi[x].val=v;
```

```
memi[x].statut=lect;
```

```
envoyer_à(gesti[x],(acklect,x));
```

Fin

A la réception de l'acquittement de lecture le nouveau lecteur est ajouté aux possesseurs d'une copie. Puis après avoir défilé la requête courante on traite la requête suivante s'il y en a une. à l'aide de la primitive finrequête.

sur_réception_de(j,(acklect,x))

Début

```
infos[x].ontcopie = infos[x].ontcopie  $\cup$  {j}; finrequête(x);
```

Fin

finrequête(x)

Début

défiler(infos[x].àtraiter);

Si infos[x].àtraiter!=vide Alors <temp_i,req_i > =tête(infos[x].àtraiter);Si req_i==reqlect Alors envoyer_à(infos[x].prop,(translect,temp_i,x));Sinon traiterécriture(temp_i,x);

Finsi

Finsi

Fin

Pour modifier un objet dont on ne possède pas le droit en écriture, on envoie une requête d'écriture et on attend le droit d'écriture. Finalement on écrit la valeur prévue.

ecrire(x,v)

Début

Si (memi[x].statut!=ecr) Alors envoyer_à(gesti[x],(reqecr,x));

Attendre(memi[x].statut==ecr);

Finsi

memi[x].val=v;

Fin

A la réception d'une requête d'écriture, si une requête n'est pas en cours on appelle la primitive traiterécriture(). Dans tous les cas, on met à jour la file des requêtes à traiter.

sur_réception_de(j,(reqecr,x))

Début

Si infos[x].àtraiter==vide Alors traiterécriture(j,x);

Finsi

```
enfiler(infos[x].àtraiter,);
```

Fin

Le traitement d'une écriture consiste d'abord à envoyer à un message d'invalidation à tous les possesseurs d'une copie excepté éventuellement le demandeur. S'il n'y en a pas, on transfère directement la requête d'écriture au propriétaire.

traiterécriture(j,x)

Début

```
Si infos[x].ontcopie\{j} != ∅ Alors
```

```
Pour tempi ∈ infos[x].ontcopie\{j} faire
```

```
envoyer_à(tempi,(inv,x));
```

```
Finpour
```

```
Sinon envoyer_à(infos[x].prop,(transecr,j,x));
```

Finsi

Fin

A la réception d'une invalidation, l'objet est invalidé et un acquittement est envoyé au gestionnaire.

sur_réception_de(j,(inv,x))

Début

```
memi[x].statut=invalidé;
```

```
envoyer_à(j,(ackinv,x));
```

Fin

A la réception d'un acquittement d'invalidation l'émetteur est extrait des possesseurs de copies. S'il n'y a plus de possesseurs autre qu'éventuellement le demandeur, la requête est transférée au propriétaire.

sur_réception_de(j,(ackinv,x))

Début

`<tempi,requi>=tête(infos[x].àtraiter);``infos[x].ontcopie = infos[x].ontcopie\{j};``Si infos[x].ontcopie\{tempi} == Ø Alors envoyer_à(infos[x].prop,(transecr,tempi,x));`

Finsi

Fin

A la réception d'un transfert d'une requête d'écriture, le propriétaire de l'objet envoie une réponse au demandeur et met à jour le statut à invalide.

sur_réception_de(j,(transecr,k,x))

Début

`memi[x].statut=invalid;``envoyer_à(k,(repecr,x));`

Fin

A la réception de la réponse d'écriture, le statut de cet objet est mis à jour et un acquittement est envoyé au gestionnaire.

sur_réception_de(j,(repecr,x))

Début

`memi[x].statut=ecr;``envoyer_à(gesti[x],(ackecr,x));`

Fin

A la réception de l'acquiescement d'écriture, le demandeur est le nouveau propriétaire de l'objet et personne n'en possède une copie. Puis après avoir défilé la requête courante, on traite la requête suivante s'il y en a une.

sur_réception_de(j,(ackecr,x))

Début

```
infos[i].ontcopie = ∅;
```

```
infos[i].prop = j;
```

```
finrequête(x);
```

Fin

7.2.4 Linéarisabilité avec gestionnaires dynamiques [Li89]

Une alternative à l'utilisation de gestionnaires statiques consiste à identifier le propriétaire et le gestionnaire *qui change au cours du temps*.

L'idée consiste à ce que chaque site ait une variable contenant l'identité supposée du gestionnaire. Cette identité est initialement correcte puis lorsqu'elle change, l'ancien propriétaire modifie de manière adéquate sa variable. Ainsi par indirection, on arrive toujours au propriétaire réel. Autrement dit, l'ensemble des liens induits par ces variables forme un arbre dont la racine est le propriétaire courant. Les variables sont similaires à celles de l'algorithme précédent (Linéarisabilité avec gestionnaires statiques) excepté que tout site peut être gestionnaire de tout objet et que l'identité du propriétaire n'a plus besoin d'être stockée.

Description Variables du site i

- gestprobi[Objets] : identité du gestionnaire probable de chaque objet initialisé au propriétaire initial de chaque objet.
- infos[i][Objets] de
 - ontcopie : ensembles des autres sites qui ont une copie de l'objet initiale vide
 - àtraiter : file de couples initialement vide (primitives associées : enfiler(), défiler(), tête())

- `memi[Objets]` de

- `val` : valeur de l'objet initialisé à la valeur initiale de l'objet s'il est initialement possédé par `i`

- `statut` : (`invalide`,`lect`,`ecr`) statut de l'objet, `ecr` pour un objet affecté, `invalide` pour un objet non affecté. On suppose que les trois valeurs sont ordonnées.

- `tempi` : variable temporaire d'identité de site

- `requi` : variable temporaire de type de requête

Algorithme du site `i`

En cas de lecture d'un objet invalide, on émet une requête au gestionnaire probable de l'objet et on attend que l'objet soit disponible en lecture.

lire(`x`)

Début

Si (`memi[x].statut==invalide`) Alors envoyer_à(`gestprobi[x]`,(`i`,`relect`,`x`));

Attendre(`memi[x].statut==lect`);

Finsi

renvoyer(`memi[x].val`);

Fin

A la réception d'une requête de lecture, si une requête n'est pas en cours, alors soit on n'est pas le gestionnaire et on transfère la requête au gestionnaire probable de l'objet, soit on est le gestionnaire. Dans ce cas, on renvoie la copie de l'objet et on met à jour le champ `ont_copie` et le statut à `lect`. Sinon on met à jour la file des requêtes à traiter.

sur_réception_de(`j`,(`k`,`relect`,`x`))

Début

Si `infos[i].àtraiter==vide` Alors Si `gestprobi[x]!=i` Alors envoyer_à(`gestprobi[x]`,(`k`,`relect`,`x`));

Sinon envoyer_à(`k`,(`relect`,`x`,`memi[x].val`));

`memi[x].statut=lect` ;

`infos[i].ontcopie = infos[i].ontcopie ∪ {k}`;

Finsi

Sinon enfiler(infos[i].àtraiter,);

Finsi

Fin

A la réception de la réponse de lecture, la valeur, le statut et le gestionnaire probable de cet objet sont mis à jour.

sur_réception_de(j,(replect,x,v))

Début

memi[x].val=v;

memi[x].statut=lect;

gestprobi[x]=j;

Fin.

Pour modifier un objet dont on ne possède pas le droit en écriture, on envoie une requête d'écriture et on attend le droit d'écriture. Après l'obtention de ce droit, on écrit la valeur prévue. Si on était déjà propriétaire d'autres requêtes qui ont pu être mises en attente, on traite la première d'entre elles.

ecrire(x,v)

Début

Si (memi[x].statut!=ecr) Alors envoyer_à(gestprobi[x],(i,reqecr,x));
Attendre(memi[x].statut==ecr);

Finsi

memi[x].val=v;

Si infosi[x].àtraiter!=vide Alors <temp_i, req_i > =tête(infosi[x].àtraiter);

défiler(infosi[x].àtraiter);

Si req_i==replect Alors envoyer_à(temp_i, (replect,x,memi[x].val));

```

memi[x].statut=lect ;

infos[i[x].ontcopie]=infos[i[x].ontcopie ∪ {tempi};

Sinon traiterécriture(tempi,x);

Finsi

Finsi

Fin

```

A la réception d'une requête d'écriture, si une requête n'est pas en cours soit on transmet au gestionnaire probable (si différent du site) soit on appelle la primitive traiterécriture(). Sinon on met à jour la file des requêtes à traiter.

sur_réception_de(j,(k,reqecr,x))

```

Début

Si infos[i[x].àtraiter]==vide Alors Si gestprob i[x]≠i Alors envoyer_à(gestprobi[x],(k,reqecr,x));
Sinon traiterécriture(k,x);

Finsi

Sinon

enfiler(infos[i[x].àtraiter,);

Finsi

Fin

```

Le traitement d'une écriture consiste d'abord à envoyer un message d'invalidation à tous les possesseurs d'une copie excepté éventuellement le demandeur. S'il n'y en a pas, on change de propriétaire car la requête émane nécessairement d'un autre site.

traiterécriture(j,x)

Début

Si infosi[x].ontcopie\{j} != $\emptyset$ Alors enfiler(infosi[x].àtraiter,);

Pour tempi $\in$ infosi[x].ontcopie\{j} faire

Envoyer_à(tempi,(inv,x));

Finpour

Sinon changeprop(j,x);

Finsi

Fin

A la réception d'une invalidation, l'objet est invalidé et un acquittement est envoyé au gestionnaire.

sur_réception_de(j,(inv,x))

Début

memi[x].statut=invalidé;

envoyer_à(j,(ackinv,x));

Fin

A la réception d'un acquittement d'invalidation, l'émetteur est extrait des possesseurs de copies. S'il n'y a plus de possesseur autre que le demandeur, la requête d'écriture peut être satisfaite. Dans le cas d'une requête locale, il suffit de mettre à jour le statut. Dans le cas d'une requête distante, on change de propriétaire.

sur_réception_de(j,(ackinv,x))

Début

<tempi,reqi> =tête(infos[x].àtraiter);

infos[x].ontcopie = infos[x].ontcopie\{j};

Si infos[x].ontcopie\{tempi} == $\emptyset$ Alors Si tempi!=i Alors défiler(infos[x].àtraiter);
 changeprop(tempi,x);

Sinon memi[x].statut=ecr;

Finsi

Finsi

Fin

En cas de changement de propriétaire, on transmet la réponse au demandeur, on met à jour son statut puis on lui retransmet les requêtes mises en attente.

changeprop(j,x)

Début

envoyer_à(j,(repecr,x));

memi[x].statut=invalidé;

gestprobi[x]=j;

Tant que infos[x].àtraiter!=vide faire

<tempi,reqi> =tête(infos[x].àtraiter);

défiler(infos[x].àtraiter);

envoyer_à(j,(tempi,reqi,x)); Fintant que

Fin

A la réception de la réponse de l'écriture, le statut de cet objet est mis à jour ainsi que les informations de gestion..

sur_réception_de(j,(repecr,x))

Début

memi[x].statut=ecr; infosi[x].ontcopie=∅;

infosi[x].àtraiter=vide;

Fin

7.3 La consistance séquentielle [Att91] [Miz92]

Nous faisons un aperçu concernant les différents algorithmes associés à la linéarisabilité, nous remarquons que :

- L'exclusion mutuelle entre les opérations (e.g. par un jeton) a une conception simple mais est d'une grande inefficacité
- La lecture à la demande ne s'applique qu'aux objets propriétaires. Elle autorise le parallélisme des lectures mais chaque lecture requiert deux envois de message consécutifs.
- Les solutions avec gestionnaire (statique et dynamique) permettent le parallélisme des lectures mais la gestion des écritures est relativement coûteuse. D'autre part, la linéarisabilité dépend de la vitesse relative d'exécution sur chacun des sites.

Nous cherchons donc à définir une sémantique de même nature pour les accès à la mémoire virtuelle c'est l'objectif de la consistance séquentielle.. Nous restreignons d'abord à la précedence temporelle aux opérations exécutées sur un même site (appelée précedence locale).

Définition (précedence locale)

Soient deux opérations sur la mémoire répartie op1 et op2. On dit que op1 précède localement op2 si et seulement si les deux opérations sont exécutées sur le même site et $op1 \prec_t op2$

Définition consistance séquentielle

Une exécution répartie est séquentiellement consistante si et seulement si il existe une relation d'ordre total sur les opérations notée " $<$ " qui étend la précédence locale et qui vérifie :

Pour toute lecture $r_i(x):v$, il existe une écriture $w_j(x,v) < r_i(x):v$ qui vérifie :

1. toute autre écriture $w_k(x,v')$ avec $v' \neq v$ vérifie :
 - soit $w_k(x,v') < w_j(x,v)$
 - soit $r_i(x):v < w_k(x,v')$

2. NOT $r_i(x):v <_t w_j(x,v)$

7.3.1 Propagation des écritures

Hypotheses:

- Nous supposons que les objets ne sont modifiés que par leur "propriétaire".
- Chaque site commence son exécution par une écriture.

L'objectif de cet algorithme est d'anticiper les lectures distantes par diffusion de chaque modification (l'opération d'écriture) d'un objet partagé par le propriétaire. Cependant il faut prendre des précautions pour obtenir une exécution répartie qui soit séquentiellement consistante. Le squelette de la séquentialisation repose sur un ordre total entre les écritures obtenu en comparant lexicographiquement les doublons (heure, identité du site) associés aux écritures dont chaque site gère une horloge logique (voir le chapitre 3) pour dater les écritures sur les objets partagés.

Conformément au fonctionnement de ces horloges, celles-ci s'incrémentent à chaque nouvelle écriture et d'autre part leur gestion garantit qu'une écriture distante a été effectué.

Réalisation de l'algorithme

Variables du site i

- $vali[Objets]$: tableau de valeurs indicé par les objets. Ces valeurs serviront comme résultat d'un appel à lire(). Chaque cellule est initialisée à une valeur par défaut commune à l'ensemble des sites mais qui est spécifique à chaque objet.
- $deri[Objets]$: tableau de valeurs indicé par les objets. Ces valeurs sont les dernières valeurs connues de l'objet correspondant. Ce tableau est initialisé de la même façon que le tableau précédent($vali[Objets]$). Il sert qu'aux objets "distants".

- h_i : heure locale de i initialisée à 0.
- $hori[1..N]$: tableau d'heures indicé par les sites initialisé à $hori[j]=0$ pour $j \neq i$ et $hori[i]=-1$. Dans le cas d'un site différent de i , la cellule fournit un minorant de l'heure de la prochaine écriture. Dans le cas du site i , la cellule fournit l'heure de la dernière écriture.
- $prop_i[Objets]$: constante qui contient l'identité du propriétaire de chaque objet.

Algorithme du site i

A la lecture d'un objet, soit on est le propriétaire et on renvoie immédiatement la valeur, soit on attend d'être sûr que les prochaines écritures du propriétaire de l'objet "suivent" la dernière écriture sur i .

lire(x)

Début

Si $(prop_i[x] \neq i)$ Alors Attendre($(hori[prop_i[x]], prop_i[x]) > (hori[i], i)$);

Finsi

renvoyer($val_i[x]$);

Fin

A la modification d'un objet, on met aussi à jour les variables dont on n'est pas propriétaire au vu des dernières valeurs reçues. On enregistre l'heure de l'écriture, on informe les autres sites de la modification et on met à jour son horloge logique.

ecrire(x,v)

Début

$val_i[x] = v$;

Pour tout y tel que $prop_i[y] \neq i$ faire

$val_i[y] = der_i[y]$;

Finpour;

$hori[i] = h_i$;

diffuser(maj, h_i, x, v);

hi++;

Fin.

A la réception d'une modification, on mémorise la valeur et on vérifie qu'elle peut être renvoyée en cas de lecture. Puis on met à jour le tableau des heures d'écritures.

sur_réception_de(j,(maj,h,x,v))

Début

deri[x]= v;

Si ((h,j)< (hori[i],i)) Alors vali[x]=v;

Finsi

hori[j]= h+1;

hi = max(hi,h+1);

envoyer_à(j,(acq,hi);

Fin

L'acquittement sert uniquement à mettre à jour le tableau des heures des opérations.

sur_réception_de(j,(acq,h))

Début

hori[j]= h;

Fin

Chapiter 8 AUTO STABILISATION

8.1 Introduction

problématique

Afin de cerner le concept d'algorithme auto-stabilisant, nous allons étudié le comportement d'un composant actif de réseaux, le commutateur.

Le concentrateur et le commutateur sont dotés de ports , connectés soit à des machines soit à d'autres composants de réseaux. Lorsque le concentrateur reçoit une trame m à destination de la machine B désignée par son adresse MAC , il retransmet la trame sur tous ses autres ports. Cette façon de procéder est simple mais inefficace car la bande passante des ports est consommée inutilement et de plus dans le cas du protocole Ethernet, le risque de collision est augmenté. De son côté, le commutateur ne retransmet la trame que sur le port à partir duquel il "pense" pouvoir atteindre la machine B mais de quelle façon connaît-il ce port ?

La réponse réside dans le mécanisme d'apprentissage intégré dans le commutateur. Ce dernier est doté d'une table associative (notée Table) dans laquelle chaque cellule comporte deux champs : une adresse de machine et un numéro de port. Cette table est initialement vide. Lorsqu'une trame arrive sur un port, le commutateur met à jour la table en ajoutant l'association entre l'adresse de l'émetteur et le port de réception (et en supprimant éventuellement une association erronée). Puis il recherche une association pour le destinataire et envoie la trame sur le port trouvé en cas de succès. En cas d'échec, il diffuse la trame sur tous les ports (excepté le port de réception).

Le commutateur localise toutes les machines sur le réseau et les trames ne sont plus diffusées. Qu'arrive-t-il si la table ne reflète plus la topologie du réseau ? Ceci peut survenir à la suite d'une perturbation temporaire rendant le mécanisme d'apprentissage du commutateur impuissant ou encore plus simplement en raison d'une modification des connexions par un administrateur réseau.

La propriété fondamentale d'un algorithme auto-stabilisant appelée "configuration", un état global de l'algorithme et de son environnement.

Sans changement ultérieur de l'environnement et partant d'une configuration quelconque, un algorithme auto-stabilisant atteint en un nombre fini d'actions une configuration cohérente, c'est à dire une configuration où les fonctionnalités de l'algorithme seront garanties dans le futur de son exécution.

On distingue deux types de tâche pour les algorithmes auto-stabilisants :

- Les tâches statiques : dans une configuration correcte, l'algorithme ne modifie plus les données à maintenir. Le maintien d'une table routage est une tâche statique.
- Les tâches dynamiques : à partir d'une configuration correcte, un service est assuré de manière attendue. La gestion de l'exclusion mutuelle entre sites est une tâche dynamique.

Nous présentons la construction d'un arbre de plus courts chemins qui peut être utilisé par un algorithme de routage. Puis nous montrons comment émuler une mémoire virtuelle répartie avec variables propriétaires. Enfin, nous appuyant sur l'algorithme précédent, on montrera comment assurer l'exclusion mutuelle entre des sites organisés en anneau. [Dol00]

8.2 Routage auto-stabilisant [Dol89]

Principe et réalisation de l'algorithme

Nous désirons construire une table de routage au dessus du graphe de communication.. Nous appellerons i_0 le site à atteindre. L'algorithme a pour objectif de construire un arbre de plus courts chemins vers la destination. Pour ce faire, chaque site i maintient deux variables significatives :

- un tableau $dist_i$ indicé par i et ses voisins qui indique la distance supposée à i_0
- $père_i$ le voisin à emprunter pour y parvenir.

L'ensemble des voisins est représenté par $voisins_i$.. Chaque site exécute une boucle infinie et à chaque tour de boucle :

1. Il calcule en fonction des distances supposées de ses voisins, sa distance minimale à i_0 et simultanément son meilleur père.
2. Il envoie à ses voisins sa distance.
3. Il arme un time-out (information permanente)

Variables du site i

- $voisins_i$: constante contenant l'ensemble des voisins i dans le graphe de communication.
- $time_out_i$: constante contenant la valeur d'un délai de suspension.
- B_i : constante contenant un majorant strict du diamètre du réseau.

- $disti[voisinsi \cup \{i\}]$: tableau des distances à $i0$.
- $pèrei$: identité du prochain noeud sur la route.
- $tempi$: variable temporaire.

Algorithme du site i

router()

Début

Tant que VRAI faire

Si ($i=i0$) Alors $disti[i]=0$;

$pèrei=i0$;

Sinon $disti[i]=Bi$;

Pour tout $tempi \in voisinsi$ faire

Si $disti[tempi] < disti[i]-1$ Alors $pèrei=tempi$;

$disti[i]=disti[tempi]+1$;

Finsi

Finpour

Finsi

Pour tout $tempi \in voisinsi$ faire envoyer_à($tempi, disti[i]$);

Finpour

Armer($time_outi$);

Attendre();

Fintant que

Fin

sur_réception_de(j,d)

Début

disti[j]=d;

Fin.

Exemple d'exécution de l'algorithme du plus court chemin (Dijkstra)

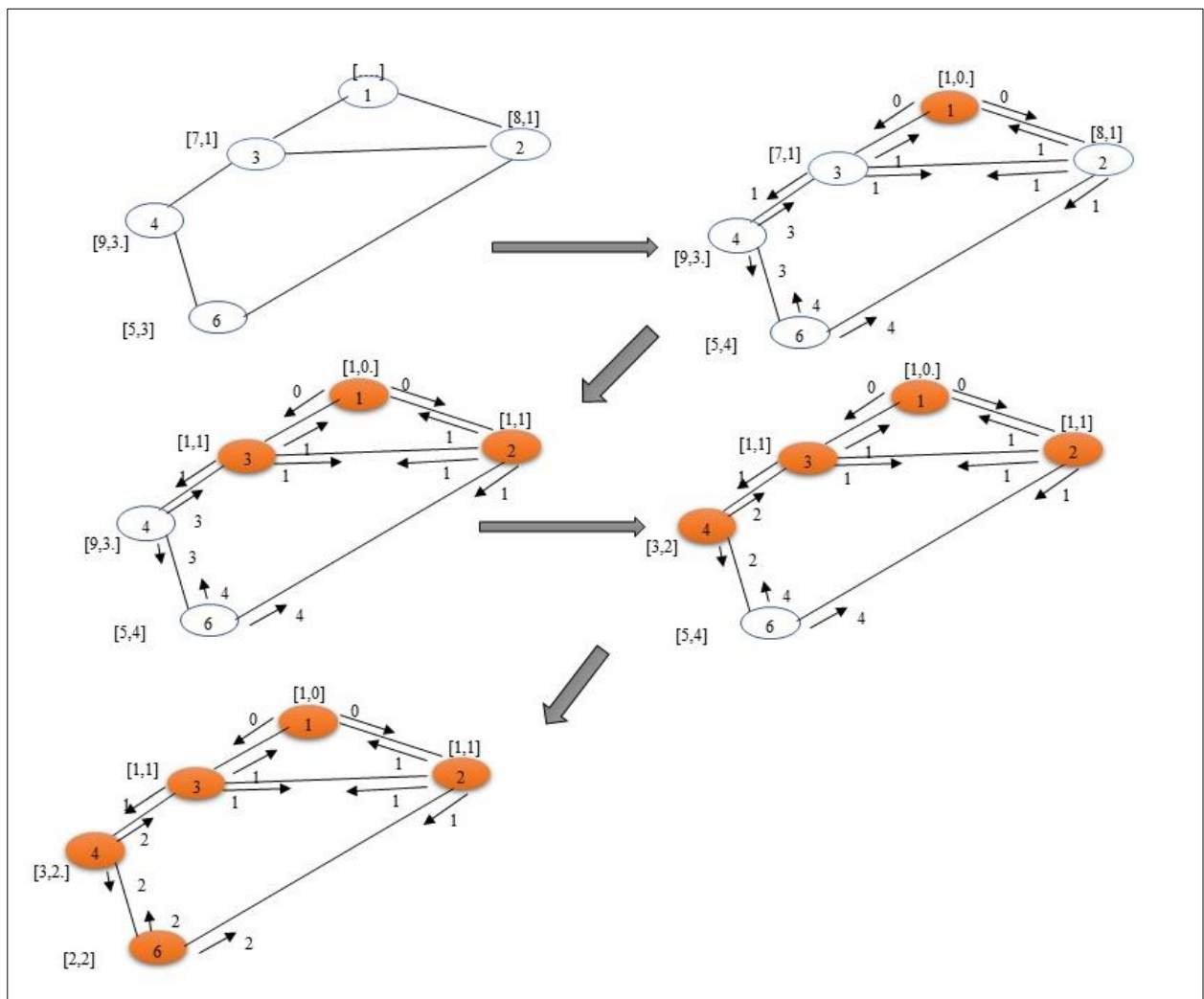


Figure 8.1 Une exécution du routage auto-stabilisant
(algorithme Dijkstra)

La figure 8.1 décrit une exécution possible de cet algorithme (le site 1 est la destination).

Nous faisons les hypothèses suivantes sur la configuration initiale :

- Le compteur ordinal du site 1 est en début de boucle.
- Le compteur ordinal des autres sites est positionné avant l'envoi des messages.
- Les valeurs initiales du père et de la distance supposée sont telles qu'indiquées sur le premier schéma.
- Initialement le réseau ne contient pas de messages . De plus, nous supposons que dans la suite de l'exécution, un message envoyé par un site arrive à destination avant l'expiration du time-out sur le site destinaire . Les voisins sont examinés par ordre croissant dans la boucle du choix du père.

On remarque les different sites se stabilisent de proche en proche en partant de la destination qui est le site 1. Chaque site décrit le prochain site pour atteindre le site de destination (site 1) avec la distance minimale correspondante qui resulte le plus court chemin pour atteindre le site destinataire.

8.3 Gestion de mémoire virtuelle répartie [Dol97]

Principe de l'algorithme

Dans le chapitre precedent qui concerne les different algorithms de la gestion de mémoire virtuelle répartie ,les messages qui circulent sur le réseau n'empêche pas celui-ci de se stabiliser.

Cependant la conception de la plupart des algorithmes auto-stabilisants se fait en supposant que les processus partagent une mémoire virtuelle répartie et plus précisément une gestion de variables propriétaires (voir le chapitre 7).

Nous supposons que chaque canal bidirectionnel contient un nombre de messages strictement inférieur à une borne B.

Dans l'algorithme precedent du lecture à la demande, le lecteur envoie une requête au propriétaire et attend la valeur en retour, il se peut que le lecteur attende une réponse que le propriétaire ne lui repondra jamais à sa requete.

La solution s'obtient par la modifications de l'algorithme de lecture à la demande et de le rendre auto stabilisant.

La première modification de l'algorithme de lecture à la demande consiste à faire circuler un jeton qui "porte" à la fois la requête et la réponse. Le lecteur attend un premier passage du jeton qui correspond à l'émission de sa requête et un deuxième passage du jeton pour recevoir la réponse. Ceci ne résout pas encore le problème puisque le jeton peut être absent de la configuration initiale. Aussi à l'aide d'un time-out, si le propriétaire ne fais pas passer de jeton, il le réémet et on aura plusieurs "jetons" circulant simultanément. Cette solution introduit un nouveau problème. Comment le lecteur va-t-il reconnaître que le message qui arrive est le jeton ?

Comme solution, les messages sont numérotés par un compteur qui s'incrémente modulo B . Chacun des sites a son propre compteur.

Variables du site distant i

- étati : état du service. Cette variable prend une valeur parmi (repos, prem_att, sec_att).
- hi : compteur du lecteur
- vali : valeur à renvoyer à la lecture

Algorithme du site distant i

La lecture consiste à se mettre en première attente (prem_att) et attendre de repasser a l'état repos pour renvoyer la valeur.

lire(x)

Début

étati=prem_att;

Attendre(étati==repos);

renvoyer(vali);

Fin.

A la réception d'un message, on teste d'abord s'il s'agit du jeton. Dans ce cas, on met à jour son compteur, puis si une lecture est en première attente (prem_att), elle passe en seconde attente (sec_att) et si elle est en seconde attente, elle passe au repos en récupérant la valeur courante de l'objet. Dans tous les cas le message est renvoyé.

sur_réception_de(j,(v,h))

Début

Si ($hi \neq h$) Alors $hi = h$;

Si ($\text{étati} == \text{prem_att}$) Alors $\text{étati} = \text{sec_att}$;

Sinon si ($\text{étati} == \text{sec_att}$) Alors $\text{vali} = v$;

$\text{étati} = \text{repos}$; Finsi

Finsi

envoyer_à(j,(acq,h));

Fin

Variables du site propriétaire j

- time_outj : constante contenant la valeur d'un délai de suspension.
- hj : compteur du propriétaire
- xj : valeur de l'objet x

Algorithme du site propriétaire j

L'écriture est purement locale.

ecrire(x,v)

Début

$xj = v$;

Fin

On envoie la valeur au site distant si un time-out expire.

régénérer()

Début

Tant que VRAI faire envoyer_à(i,(xj,hj));

Armer(time_outj);

Attendre();

Fintant que

Fin

A la réception d'un message qui porte le compteur (i.e. reconnu comme un jeton), on incrémente celui-ci et on le renvoie. Puis on envoie la valeur et le compteur et on réarme le time-out pour éviter des envois en plus.

sur_réception_de(i,(acq,h))

Début

Si (hj==h) Alors hj =(hj+1) %B;

envoyer_à(i,(xj,hj));

Armer(time_outj);

Finsi

Fin

8.4 Exclusion mutuelle [Dij73]

Présentation

L'algorithme précédent assure implicitement une exclusion mutuelle entre le propriétaire et le lecteur. On désire généraliser ce principe pour obtenir l'exclusion mutuelle entre un ensemble de sites.

C'est l'objectif de l'algorithme de Dijkstra qui a de plus la particularité d'être l'algorithme d'où a émergé le concept d'auto stabilisation.

Tout d'abord nous organisons les N sites en un anneau "unidirectionnel" avec un site distingué i_0 (l'équivalent du propriétaire). Chaque site est doté d'un compteur.

Nous faisons l'hypothèse que chaque site peut lire la valeur du compteur du site précédent sur l'anneau.

La possibilité d'entrer en section critique est matérialisée de la façon suivante :

- Le site distingué peut entrer en section critique, lorsque son compteur est égal au compteur précédent.
- Un autre site peut entrer en section critique lorsque son compteur est différent du compteur précédent. La sortie de section critique se définit ainsi :
- Le site distingué (site propriétaire i_0) incrémente son compteur modulo $N+1$.
- Un autre site recopie la valeur du compteur précédent dans son compteur. Evidemment, si l'application n'est pas intéressée par la section critique, le service effectue la sortie de section critique immédiatement.

Spécification de l'algorithme

Lorsque l'application d'un site veut exécuter une fonction en section critique, elle appelle la primitive $SC(f)$ où f est la fonction à exécuter.

Variables du site i

- ci : compteur du site i.
- cpreci : compteur du site précédent i sur l'anneau. Il n'est accédé qu'en lecture par le site i.
- fi : fonction à exécuter en section critique
- time_outi : constante contenant la valeur d'un délai de suspension.

Algorithme du site i

Le processus qui fait circuler le droit d'entrer en section critique, teste d'abord la possibilité d'entrée en section critique. Si tel est le cas, il appelle la fonction prévue et remet la fonction à NULL après exécution. Il fait ensuite circuler le droit d'entrer à la section critique. Nous faisons ici l'hypothèse que le mécanisme d'appel de fonction applicative ne peut pas interrompre le processus de service et que ce mécanisme teste si l'adresse est nulle.

circuler()

Début

Tant que VRAI faire

Armer(time_outi);

Attendre();

Si (i==i0)&&(ci==cpreci) || (i!=i0)&&(ci!=cpreci) Alors

SC(f);

fi=NULL;

Si (i==i0) Alors ci =(ci+1) %(N+1);

Sinon ci = cpreci;

Finsi Finsi

Fin tant que

Fin

Pour exécuter une fonction f en section critique , un processus positionne la variable de fonction de manière appropriée et attend l'indication d'exécution.

SC(f)

Début

$fi=f;$

Attendre($fi==NULL$);

Fin

Les problèmes rencontrés lors de l'exécution de cet algorithme:

- Nous remarquons d'abord que la variable fi sera nécessairement soit positionnée à NULL soit à la fonction à exécuter après l'obtention du premier droit. Seul le premier appel à $fi()$ peut donc être incorrect.
- L'hypothèse que chaque site peut lire la valeur du compteur du site précédent sur l'anneau pose un problème lors de la lecture à distance du compteur c_{prec} du site précédent .

Amélioration et Extension de l'algorithme

Nous allons maintenant lever l'abstraction qui consiste à supposer qu'un site peut lire le compteur de son prédécesseur.

- c_{prec} sera maintenant une copie distante de ce compteur qui donne lieu à une lecture distante qui sera maintenu par l'algorithme de mémoire virtuelle répartie.

Variables du site i

Tout d'abord chaque site joue à la fois le rôle du propriétaire pour son compteur et celui du lecteur pour le compteur précédent. Ceci explique que la variable hi soit "dédoublée". On note $H=2.N-1$.

- ci : compteur du site i .
- c_{prec} : copie distante du compteur du site précédent i sur l'anneau.

- f_i : fonction à exécuter en section critique
- h_i : compteur associé à c_i .
- étati : état du service de lecture. Cette variable prend une valeur parmi (repos , prem_att , sec_att).
- h_{preci} : compteur associé à c_{preci}
- vali : valeur à renvoyer à la lecture
- time_outi : constante contenant la valeur d'un délai de suspension.
- suivanti : constante contenant l'identité du suivant sur l'anneau.

Algorithme du site i

La première partie de l'algorithme est presque identique à l'algorithme précédent à la différence près que c_{preci} donne lieu à une lecture distante.

circuler()

Début

Tant que VRAI faire

$c_{\text{preci}} = \text{lire}();$

Si $(i == i_0) \&\& (c_i == c_{\text{preci}}) \parallel (i != i_0) \&\& (c_i != c_{\text{preci}})$ Alors SC(f);

$f_i = \text{NULL};$

Si $(i == i_0)$ Alors $c_i = (c_i + 1) \% H;$

Sinon $c_i = c_{\text{preci}};$

Finsi

Finsi

Fin tant que

Fin

SC(f)

Début

fi=f;

Attendre(fi==NULL);

Fin

La deuxième partie consiste à faire jouer les deux rôles de la lecture distante à un même site selon la variable. La fonction lire() est inchangée.

lire()

Début

étati=prem_att;

Attendre(étati==repos);

renvoyer(vali);

Fin

régénérer()

Début

Tant que VRAI faire envoyer_à(suivanti,(ci,hi));

Armer(time_outi);

Attendre();

Fin tant que

Fin

Cette primitive correspond au lecteur distant. Aussi elle manipule hpreci.

sur_réception_de(j,(v,h))

Début

Si (hpreci!=h) Alors hpreci=h;

Si (étati == prem_att) Alors étati=sec_att;

Sinon si (étati == sec_att) Alors vali=v;

étati=repos;

Finsi

Finsi

envoyer_à(j,(acq,h));

Fin

Cette primitive correspond au propriétaire. Aussi elle manipule hi.

sur_réception_de(j,(acq,h))

Début

Si (hi==h) Alors hi = hi+1 %B;

envoyer_à(suivanti,(ci,hi));

Armer(time_outi);

Finsi

Fin

Nous pouvons identifier l'instruction "cpreci=read();" à l'instruction "cpreci=cj" d'une exécution séquentielle de l'algorithme pour j prédecesseur de i sur l'annau. Nous concluons que cet algorithme a pu généraliser le principe de l'exclusion mutuelle entre l'ensemble de sites.

Les références bibliographiques

- [Ray91] M. Raynal "La communication et le temps dans les réseaux et les systèmes répartis" Collection Direction des Etudes et des Recherches d'EDF n°75. Hermès. 1991 ISSN 0399- 4198
- [Ray92] M. Raynal "Synchronisation et état global dans les systèmes répartis" Collection Direction des Etudes et des Recherches d'EDF n°79. Hermès. 1992 ISSN 0399-4198
- [Tan94] A. Tanenbaum "Systèmes d'exploitation. Systèmes centralisés. Systèmes distribués" Troisième Edition Dunod – Prentice Hall. ISBN 2-10-004554-7. 1994
- [Tel00] G. Tel "Introduction to Distributed Algorithms" Second Edition Cambridge University Press. ISBN 0-521-79483-8. 2000
- [Car83] O.S.F. Carvalho, G. Roucairol "On mutual exclusion in computer networks" Communication of ACM vol 26,2 février 1983 pp 146-147.
- [Lam78] L. Lamport "Time, clocks, and the ordering of events in a distributed system" Communications of the ACM 21,7 juillet 1978 pp 558-565.
- [Ric81] G. Ricart, A.K. Agrawala "An optimal algorithm for mutual exclusion in computer networks" Communication of ACM vol 24, janvier 1981 pp. 9-17.
- [Cha79] E.J.-H. Chang, R. Roberts "An improved algorithm for decentralized extrema finding in circular arrangements of processes" Communication of ACM vol 22 (1979) pp 281-283.
- [Dol82] D. Dolev, M. Klawe, M. Rodeh " An $O(n \log n)$ unidirectional distributed algorithm for extrema-finding in a circle." Journal of algorithms 3 (1982) pp 245-260.
- [Fra82] W.R. Franklin "On an improved algorithm for decentralized extrema finding in circular configurations of processors" Communications of the ACM 25,5 (1982) pp 336-337.
- [Att91] H. Attiya, J. Welch "Sequential consistency versus linearizability" Proceedings du troisième ACM symposium on parallel algorithms and architectures (juillet 1991) pp 304-315.
- [Bag89] R.L. Bagrodia "Synchronisation of asynchronous processes in CSP" ACM Toplas, vol 11,4 (Oct. 1989), pp. 585-597
- [Boc79] G.V. Bochmann "Architecture of distributed computer" Springer-Verlag, LNCS 77 (1979)

- [Lam78] L. Lamport "Time, clocks, and the ordering of events in a distributed system" Communications of ACM 21 (1978), 558-564
- [Her90] M. Herlihy, J. Wing "Linearizability : a correctness conditions for concurrent objects" ACM transactions on programming languages and systems. Vol 12,3 (1990) pp 463- 492.
- [Li89] K. Li, P. Hudak "Memory coherence in shared virtual memory systems" ACM transactions on Computer Systems Vol. 7,4 (1989) pp 321-359.
- [Miz92] M. Mizuno, M. Raynal, G. Singh, M. Neilsen "Communication efficient distributed shared memories" Rapport de recherche IRISA n° 691. Décembre 1992.
- [Ray93] M. Raynal, M. Mizuno "How to find his way in the jungle of consistency criteria for distributed object memories" Rapport de recherche INRIA n° 1962. Juillet 1993.
- [Dij73] E.W. Dijkstra "Self-stabilization in spite of distributed control" EWD391 SpringerVerlag (1973) pp 41-46.
- [Dol89] S. Dolev, A. Israeli, S. Moran "Self-stabilization of dynamic systems" Proceedings of the MCC Workshop on Self-stabilizing systems, MCC Technical Report N° STP-379-89 (1989)
- [Dol97] S. Dolev, A. Israeli, S. Moran "Resource bounds for self-stabilizing message driven protocols" SIAM Journal of Computing 26, pp 273-290 (1997)
- [Dol00] S. Dolev "Self-Stabilization" MIT press (2000)
- [Gou91] M.G. Gouda, N. Multari "Stabilizing communication protocols" IEEE Transactions on computers, 40, pp 448-458 (1991)