Première partie : Algorithmique répartie

1. Introduction :

1.1 Définition d'une architecture répartie :

Une architecture répartie est un ensemble de processeurs communiquent entre eux via un dispositif de communications (bus, réseau, etc.)

1.2 Définition d'un système réparti :

Un système réparti se compose d'un ensemble de processus et d'un ensemble de services assurant la communication entre ces processus. Un système réparti fonctionne au-dessus d'une architecture répartie.

1.3 Définition d'un calcul réparti :

Un calcul réparti est un calcul parallèle réalisé par un ensemble de processus qui utilisent plusieurs processeurs (un (ou plusieurs) processus par processeurs).

La communication entre ces processus peut être réalisée :

- par mémoire partagée, ou
- par échange de messages.

Un calcul réparti se fait au dessus d'un système réparti

1.4 Modèles temporels d'un système distribué :

Il existe deux modèles :

- **Synchrone** : il existe des bornes connues sur les vitesses relatives des processus, sur les délais de transmission de messages et sur la dérive des horloges.
- **Asynchrone :** aucune bornes.

1.5 Défaillances :

Il existe deux sortes de défaillances :

1.5.1 Matérielles :

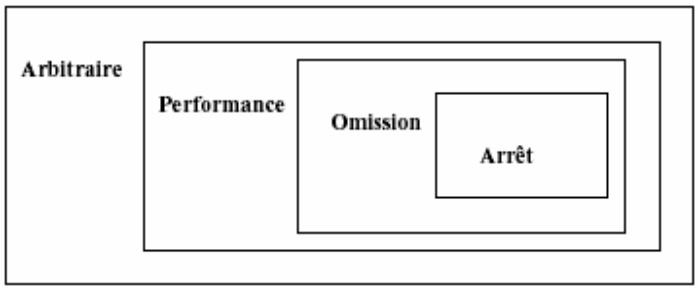
- Crash d'un processeur
- Défaillance d'un canal de communication : il existe plusieurs type
  - la destruction du canal ;
  - la perte de message ;
  - la duplication de message ;
  - la corruption de message.

- Ecrasement d'une mémoire
- etc.

1.5.2 Logicielles

Il existe 4 types de défaillances :

- **La défaillance par arrêt définitif :** Dans ce type de défaillances, un processus s'arrête prématurément et ne fait rien à partir de ce point et il ne peut pas reprendre son exécution. La terminologie panne franche (ou crash dans la littérature anglo-saxonne) est aussi utilisée pour décrire un arrêt définitif.
- **La défaillance par omission :** Dans ce type de défaillance, un processus défaillant peut omettre certaines actions. Ces actions concernent l'envoi et la réception de messages
- **Les défaillances de performance :** Dans ce type de défaillance, un processus défaillant ne respect pas les délais d'exécution des tâches.
- **La défaillance arbitraire (byzantine) :** Dans ce type de défaillance, un processus défaillant peut se comporter arbitrairement. Un processus byzantin peut, par exemple, corrompre des messages ou encore ne pas exécuter volontairement des parties entières de l'algorithme.



Classification de défaillances de processus

1.6 Avantages d'un système distribué asynchrone

- Meilleure tolérance aux défaillances (décentralisation des décisions).
- Meilleure efficacité
  - répartition du travail (parallélisme)
  - autonomie relative des processus (asynchronisme)

Ce qui implique un couplage faible : moins de risques de blocages mutuels

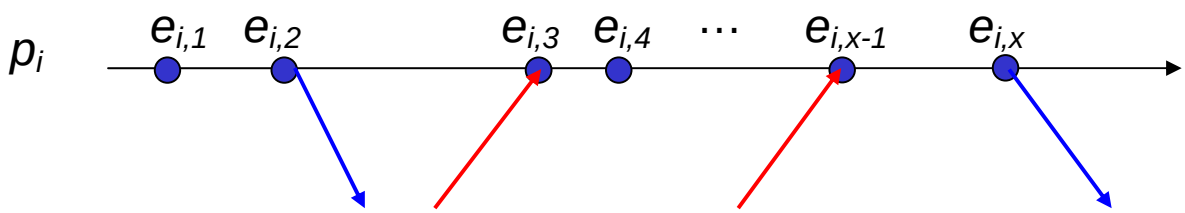
1.7 Inconvénients d'un système distribué asynchrone

- Contrôle des processus plus difficile, à cause :
  - de l'asynchronisme : évaluations instantanées impossibles (expressions réparties sur plusieurs processus)
  - des défaillances : maintien de la cohérence du calcul difficile
  - de la conjonction des deux : comment distinguer entre un processus

2- Calculs repartis asynchrones fiables :

Nous considérons dans ce qui suit le modèle du système suivant :

- Ensemble de processus $p_1, p_2, \ldots, p_n$
- Séquence d'évènements locaux par processus :
  - histoire de p_i : $[e_{i,1}; e_{i,2}; \ldots ; e_{i,x}; \ldots [$
  - $h_{i,x} = [e_{i,1}; e_{i,2}; \ldots ; e_{i,x}]$
  - $h_{i,x+1} = h_{i,x} . [e_{i,x+1}]$
- Trois types d'évènements :
  - Internes (mise à jour de l'état local du processus)
  - Emissions de valeurs (mise à jour du médium de communication)
  - Réceptions de valeurs (mise à jour du médium de communication et de l'état local du récepteur)

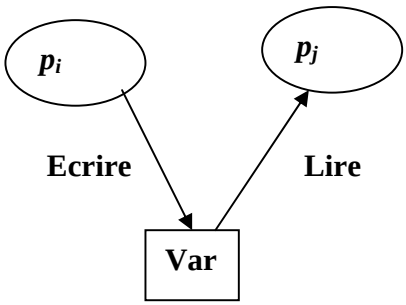


2.1 Communications

p_i transmet une valeur *v à* p_j au moyen d'un canal $\Rightarrow$communication entre p_j et p_i

Pour transmettre la valeur *v* il existe deux sortes de communication :

- **Mémoire partagée :** valeurs stockées par variables accessibles par les processus $\Rightarrow$Un canal est une variable partagée.
- **Messages :** valeurs transportées par messages véhiculées par des canaux de communication.



Variable partagée

2.1.1 Communications par messages

- Chaque message a un *identifiant unique m*
- Un canal est une ligne de transmission orientée : $c_{i,j}$ est canal transmettant les messages de p_i (émetteur) vers p_j (récepteur)
- Un état est un ensemble de messages

2.1.2 Evènements de communication

Il existe deux types d'évènement :

- Emission : (sur un processus p_i)
  - évènement *émission de m vers* p_j
  - action **envoyer** (m, p_j)
  - effet " dépose " m dans le canal $c_{i,j}$

- Réception: (sur un processus p_j)
  - évènement *réception de m (depuis* p_i)

- action **délivrer** (*m*)
- effet ” retire ” *m* du canal $c_{i,j}$; met à jour les variables de p_j

2.1.3 Fiabilité des canaux

Un canal est dit fiable ssi :

- ❖ pas de pertes : si émission de *m* vers p_j a lieu sur p_i alors réception de *m* aura sûrement lieu sur p_j.
- ❖ pas de duplication : si réception de *m* (depuis p_i) a lieu sur p_j alors émission de *m* vers p_j a sûrement eu lieu (sur p_i)

3. Ordonnancement des évènements :

- o Sur chaque processus : Ordre total sur évènements locaux

$$e_{i,x} \xrightarrow{l} e_{i,y} \Leftrightarrow x \leq y$$

- o Relation de communication :

$$e_{i,x} \xrightarrow{com} e_{i,y} \Leftrightarrow ((e_{i,x} = send(m, p_j)) \wedge (e_{j,y} = receive(m, p_i))$$

3.1 Causalité

La relation de causalité a été définit par Lamport (1978) : $\xrightarrow{hb} \equiv (\xrightarrow{l} \bigcup \xrightarrow{com})^*$

$$e_{i,x} \xrightarrow{hb} e_{j,y} \Leftrightarrow \begin{cases} i = j \wedge x \leq y & (\xrightarrow{l}) \text{ ou} \\ e_{i,x} \xrightarrow{com} e_{j,y} & (\xrightarrow{com}) \text{ ou} \\ \forall e_{k,z} : e_{i,x} \xrightarrow{hb} e_{k,z} \wedge e_{k,z} \xrightarrow{hb} e_{j,y} & (*) \end{cases}$$

La relation de causalité est un ordre partiel

- o réflexive, transitive
- o antisymétrique (c’est-à-dire *acyclique*) par définition d’un calcul réparti

Un circuit ” non trivial ” violerait le principe de causalité : un évènement devrait se produire *avant sa cause*.

Théorème : tout calcul réparti est un ordre partiel

3.2 Estampillage linéaire

- But : obtenir un ordre total sur les évènements
- Moyen : estampiller les évènements, de sorte que :

$$e \xrightarrow{hb} e' \Rightarrow h(e) \leq h(e') \qquad \text{(Strictement si } e \neq e' \text{)}$$

4. Horloges logique :

4.1 Horloges de Lamport :

Les horloges de Lamport sont définit comme suit :

- Chaque processus p_i a une horloge logique h_i (horloge locale)
- R1 : $e_{i,x}$ et $e_{i,x+1}$ internes ou émission : $h(e_{i,x+1}) = 1+h(e_{i,x})$
- R2 : tout m émis par pi transporte $m.h = h_i(send(m))$
- R3 : lors de réception de m par p_j : $h_j(réception(m)) = 1+\max(h_j,m.h)$

Est-ce que les Horloges de Lamport $\Rightarrow$ ordre total ?

Théorème :

$$e \xrightarrow{hb} e' \Rightarrow h(e) \leq h(e') \qquad \text{(Strict si } e \neq e' \text{)}$$

Proposition : $e \longrightarrow e' \equiv h(e) \leq h(e')$ est pré-ordre total

Pas d'antisymétrie : $h(e) = h(e') \not\Rightarrow e = e'$

En fait $h(e) = h(e') \Rightarrow e // e'$ mais $e // e' \not\Rightarrow h(e) = h(e')$

Donc, pour avoir un ordre total sur les événement, on ajoute un deuxième critère secondaire aux horloges :

$estamp(e) = (h, i)$

h = estampille de e (valeur d'horloge), i = identité de p_i

$$e(h,i) \prec e'(h', j) \equiv \begin{cases} h < h' & \text{ou} \\ h = h' & \text{et } i < j \end{cases}$$

Interprétation :

- $h(e) = v$ équivaut à : Il y a exactement v- 1 événements le long du plus long chemin causal se terminant en e
- La relation d'ordre total n'est pas équivalente à la causalité:

  $e \neq e' \wedge e \xrightarrow{hb} e' \Rightarrow e \prec e'$, mais la réciproque est fausse

4.2 Horloges vectorielles de Fidge et Mattern :

Le but d'utiliser ces horloges est de repérer les évènements selon l'ordre partiel de causalité.

On peut faire ça avec l'estampillage des évènements, de sorte que :

$$e \xrightarrow{hb} e' \Leftrightarrow h(e) \leq h(e')$$

Définition :

- Chaque processus p_i a un vecteur d'horloges H_i.
- Règle 1 : Lorsque p_i produit un évènement: $m.H := H_i[i] + 1$ (compteur des évènements de p_i).
- Règle 2 : Tout m émis par p_i transporte : $m.H = H_i(send(m))$
- Règle 3 : Lors de réception de m par p_j : $\forall k : H_j[k] = \max(H_j[k], m.H[k])$.

Les horloges vectorielles sont équivalentes à la causalité

Théorème :

$$e \neq e' \wedge e \xrightarrow{hb} e' \Leftrightarrow H(e) < H(e') \qquad \text{(Strict si } e \neq e'\text{)}$$

5- Problèmes de synchronisation :

La synchronisation est l'ensemble de règles et de mécanismes pour assurer et contrôler une évolution correcte des processus.

On peut classifier les problèmes de synchronisation en deux grandes classes :

- **Problèmes de Compétition :** chaque processus "ignore" a priori l'existence des autres (inutiles pour lui).

  Par exemple :

  - Exclusion mutuelle
  - Allocation de ressources

- **Problèmes de Coopération :** chaque processus a besoin de coopérer avec les autres pour la réalisation de sa tâche.

  Par exemple :

  - Rendez-vous généralisé
  - Maintien d'un invariant global
  - Construction d'un temps virtuel global

7- Exclusion mutuelle

L'exclusion mutuelle est l'accès à une ressource partagée, par un seul processus à la fois.

Les solutions suivantes peuvent être utilisées :

- Obtention d'un verrou d'écriture sur un fichier partagé,
- Mise à jour cohérente de copies multiples.

7.1- Formulation :

- Chaque P_i a une variable $etat_i \in \{dehors, demandeur, dedans\}$
- Chaque P_i peut évoquer acquérir, libérer

7.2- Structure d'un protocole d'exclusion mutuelle

Tout protocole d'exclusion mutuelle a la structure suivante :

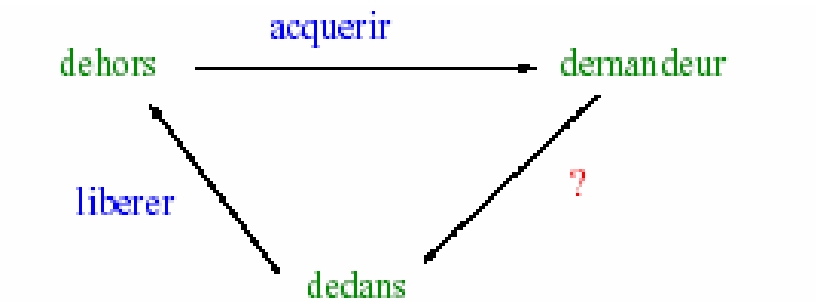
Pour tout processus P_i

Prélude $\{etati = dehors\}$ acquérir ; $\{etati = demandeur\}$ attendre-privilege ; $\{etat_i = dedans\}$

< Section Critique >

Postlude $\{etati = dedans\}$ libérer ; $\{etati = dehors\}$

7.3- Exclusion mutuelle : le problème



Problème : comment passer de l'état demandeur à l'état dedans ?

Les transitions acquérir et libérer sont activées par le processus lui-même.

La transition demandeur $\rightarrow$ dedans est contrôlée par le protocole d'exclusion mutuelle

7.4- Spécification:

La spécification d'une solution est caractérisée par les deux propriétés suivantes :

- **Sûreté :** au plus un processus dans l'état dedans
- **Vivacité :** tout demandeur devient dedans au bout d'un temps fini (ni inter blocage ni famine)

7.5- Classes de solutions :

Il existe deux grandes classes :

- **Protocoles à Permissions**

  Obtention du privilège ≡ obtenir des autres un nombre suffisant de permissions

  Sûreté : permissions exclusives

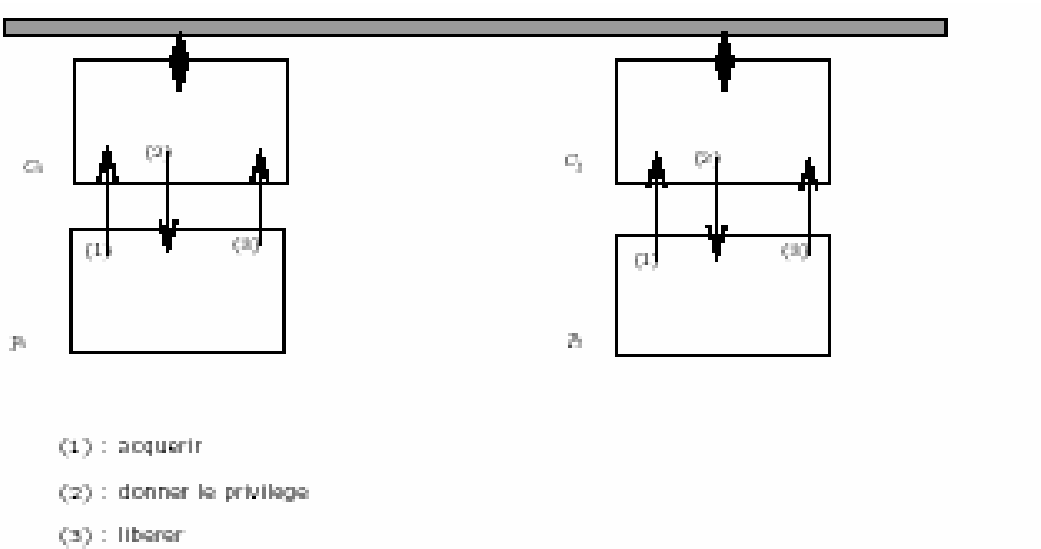
- **Protocoles à jeton**

  Obtention du privilège = possession du jeton

  Sûreté : unicité du jeton

7.6- Les contrôleurs :

Le protocole d'exclusion mutuelle est assuré par les contrôleurs qui coopèrent entre eux.



8- Protocoles à Permissions

A tout contrôleur *Ci* est associe un ensemble de contrôleurs R_i

Acquérir : $\forall c \in R_i$: envoyer requête à c

Obtention du privilège : $\forall c \in R_i$: permission reçue

Q- Comment assurer la sûreté ?

R- Propriétés souhaitables des ensembles R_i :

Exemples :

$R_i = \{i\}$ ne convient pas !

$R_i = \{1,2,...,n\}$ convient

Il existe de type de protocoles :

- Permissions individuelles: un contrôleur peut donner sa permission à plusieurs requêtes simultanément. “En ce qui **me** concerne, vous pouvez entrer en SC”
  Propriété invariante : $\forall$ i $\forall$ j : $C_j \in R_i \vee C_i \in R_j$
- Permissions d’arbitre Un contrôleur ne donne sa permission qu’à une seule requête `a la fois. “En ce qui concerne tous les processus que j’arbitre, vous pouvez entrer en SC”
  Propriété invariante : $\forall i \forall j : R_i \cap R_j \neq \phi$

Pour garantir la vivacité : deux solutions

- Ordre total sur les requêtes (par estampillage de Lamport).

Requêtes servies dans cet ordre.

- Graphe de priorité (ordre partiel) $C_i \xrightarrow{prio} C_j$. Graphe dynamique, toujours acyclique

8.1- Protocoles à permissions individuelles:

A- Protocole de Ricart et Agrawala (1981):

- $\forall i \ R_i = \{1,2,...,n\} - \{i\}$
- Gestion des priorités

$last_i$=date de la dernière op. acquérir de p_i;

Lors de réception de *requête* (*h, j*) :

priorite$_i$:= (*etat*$_i$ = *dehors*) $\wedge$ (*lasti, i*) < (*h, j*)

Si *priorite*$_i$ alors mettre la requête dans une file

sinon envoyer permission ($_i$) à C_j

Lorsque p_i invoque libérer: envoyer permission à toutes les requêtes de la file

Texte du protocole :

var $etat_i$: $(dehors, demandeur, dedans)$ init $dehors$
h_i : entier init 0
$last_i$: entier init 0
$prioritaire_i$: booleen init faux
$nbatt_i$: entier init 0
$differe_i$: ensemble de id_proc init $\emptyset$
R_i : ensemble de id_proc init $\{1, 2, \ldots, n\} - \{i\}$

```
proc acquerir
      debut
          etat_i:= demandeur;
          h_i := h_i + 1 ; last_i := h_i ; (abrégé en last_i := h_i + 1)
          nbatt_i := card(R_i) ;
          pourtout j de R_i
                    envoyer requete(last_i, i) a C_j
          fpourtout ;
          attendre nbatt_i = 0 ;
          etat_i:=dedans
      fin
```

```
proc liberer
debut
      pourtout j de differe_i
                envoyer permission a C_j
      fpourtout ;
      differe_i := ∅;
      etat_i := dehors
fin
```

```
lors de reception requete(k,j)
        debut
            h_i := max(h, k) ;
            priorite_i := (etat_i ≠ dehors) ∧ (last_i, i) < (k, j);
            si priorite_i
                        alors differe_i := differe_i ∪ {j}
                        sinon envoyer permission a C_j
            fsi
        fin

lors de reception permission de j
        debut
            nbatt_i := nbatt_i + 1
        fin
```

B- Protocole de Carvalho-Roucairol (1983)

Idée : il est inutile de demander une permission déjà obtenue et non rendue. Donc, il est plus intéressant si l'évolution des ensembles R_i soit dynamique :

- lorsque *Ci* reçoit *permission*(*j*), il supprime *j* de *Ri*
- lorsque *Ci* envoie *permission*(*i*) à *Cj*, il ajoute *j* à *Ri*

La relation ***i ∈ Rj* ouexcl *i ∈ Ri*** assure la sureté.

Texte du protocole :

Seules les modifications par rapport à Ricart Agrawala sont notées
La variable $nbatt_i$ disparait
Initialisation des R_i t.q. $i \in R_j$ OUEXCL $j \in R_i$

```
proc acquerir
    debut
        etat_i := demandeur;
        last_i := h_i + 1;
        pourtout j de R_i
                envoyer requete(last_i, i) a C_j
        fpourtout ;
        attendre R_i = ∅ ;
        etat_i := dedans
    fin
```

```
proc liberer
    debut
        pourtout j de differe_i
                envoyer permission a C_j
        fpourtout ;
        R_i := differe_i;
        differe_i := ∅;
        etat_i := dehors
    fin

lors de reception permission de j
    debut
        R_i := R_i − {j}
    fin
```

```
lors de reception requete(k,j)
debut
    h_i := max(h, k) ;
    priorite_i := (etat_i = dedans)∨
                  ((etat_i = dem) ∧ (last_i, i) < (k, j));
    si priorite_i
      alors differe_i := differe_i ∪ {j}
      sinon envoyer permission a C_j
            R_i := R_i ∪ {j}
            si etat_i = dem alors
              envoyer requete(last_i, i) a C_j
            fsi
    fsi
fin
```