TABLE DES MATIÈRES

TABLE DES MATIÈRES

CHAPITRE 1

Principes et mécanismes de base des systèmes distribués

1.1 Introduction

1.1.1 Rôles d'un système distribué

Le terme de système distribué désigne une collection de processus coopérant dans le but de réaliser une application spécifique. Les fonctions réalisées par ces différents processus peuvent être de différentes natures

- soit de calcul
- soit de base : exclusion mutuelle
- soit de contrôle (liées à la distribution) : détection de la terminaison, cohérence,...

Ainsi, les systèmes d'exploitation distribués ne calculent pas de résultats (au sens propre du terme) aux termes desquels ils sont terminés mais ils rendent des services $\Longrightarrow$ ils offrent des moyens aux applications réparties et permettent :

- d'assurer la communication et le partage d'informations entre des applications ;
- d'assurer l'exécution parallèle de programmes sur des processeurs différents ;
- et de partager des ressources physiques et logiques sur un ensemble d'utilisateur.

d'où la nécessité de mécanismes :

- de désignation, de transferts et de partages d'informations ;
- de contrôle de la cohérence de ces informations ;
- de diffusion d'informations ;
- de synchronisation, de placement de processus et de détection de leur terminaison ;
- de contrôle global du système et d'élection ;
- d'ordonnancement de processus ;
- d'exclusion mutuelle ;

- de reprise sur panne ;
- de transaction ;
- d'administration...

$\Longrightarrow$ plusieurs modèles dont les deux principaux sont le modèle serveur/clients et le modèle à objet.
Il est à noter que l'architecture sous-jacente est une donnée fondamentale. En effet en fonction du fait qu'elle soit **fortement** couplée ou **faiblement** couplée la problématique n'est pas la même.

- Dans le premier cas, la communication est faite via une mémoire commune : les problèmes sont plus simples à résoudre. Ainsi, l'exclusion mutuelle peut être simplement résolu par un blocage du bus (ensemble des lignes de communication connectant les différents composants d'un ordinateur. Exemple : clé usb) comme en mono processeur.
- Dans le deuxième cas, les communications se font via des messages avec tous les problèmes liés aux réseaux (perte d'informations, ...)

Il est à noter aussi que architecture parallèle ne veut pas dire système distribué. Ainsi une machine bi-processeurs peut être Solaris ou Linux bien que ni l'un ni l'autre ne soit un système d'exploitation distribué : ce sont des systèmes d'exploitation symétriques.

1.1.2 Caractéristiques des systèmes distribués

Dans la suite, nous considérons des systèmes distribués faiblement couplés $\Longrightarrow$ les problèmes de contrôle ne peuvent être réglés par des primitives de synchronisation des accès à une mémoire $\Longrightarrow$ nécessité d'utiliser des mécanismes de communication.

1.1.3 État global

Une des caractéristiques essentielles d'un tel système distribué est l'absence d'**état global** perceptible à un instant donné par un observateur.

1. les processus ne "connaissent" que les événements qu'ils ont générés ;
2. les processus ne connaissent que les messages qu'ils ont reçus ou envoyés $\Longrightarrow$ ils ne connaissent pas "l'état" du réseau (ce qui y circule) ;
3. il n'y a pas de relation naturelle d'ordre strict sur des événements ayant lieu sur des machines différentes.

$\Longrightarrow$ l'état des processus à un instant donné n'est pas suffisant.

1.1.4 Distribution des données

Deux cas :

- distribution inhérentes aux problèmes : le concepteur se voit imposé, par exemple une distribution géographique (disques de Bases de données sur plusieurs sites, ...).
- distribution inhérente à la mise en oeuvre : les utilisateurs veulent pouvoir travailler sur des sites distants.

d'où

- duplication : une donnée peut se trouver sur plusieurs sites $\Longrightarrow$ problème de cohérence (respect de la causalité) ;

– partitionnement : une grande BD peut être couplée en plusieurs sous-BD sur des sites différents $\Longrightarrow$ problème de désignation (comment retrouver une donnée de la base : il faut d'abord savoir sur quel site elle se trouve).

Bien évidemment, les deux peuvent se faire en même temps.

1.1.5 Contrôle

On dira qu'il y a *contrôle distribué* lorsqu'il n'y a pas de relation hiérarchique **statique** entre les processeurs : il n'y a pas de processus qui joue le rôle particulier à priori. Ainsi, il n'y a pas de maître qui assure en permanence le contrôle global.
Pourquoi ?

– tolérance aux fautes faible voire nulle : si ce processus s'arrête $\Longrightarrow$ fin du système ;
– goulot d'étranglement (point d'un système limitant les performances globales et pouvant avoir un effet sur les temps de traitement et de réponse).

Il se peut que pour certaines fonctions du système, il y ait nécessité d'un maître : on utilisera alors un mécanisme d'élection.

1.2 Éléments de base des systèmes distribués

1.2.1 Protocole

Un **protocole** défini le comportement d'un processus vis-à-vis d'autres processus : on va s'intéresser principalement (et presque uniquement) aux réactions du processus à la réalisation d'événements : réception d'un message, réalisation d'une condition, émission d'un message etc.
Par exemple, on peut définir (succinctement) la réservation d'une imprimante par le protocole suivant : Pour un processus P_i

- Etat initial : Libre
- Pour utiliser l'imprimante, passer à l'état Demande
  – Demande
    émettre la demande vers tous les autres processus (***)
    mettre un compteur à zéro (*)
    passer à l'état Attente
  – Attente
    a la réception d'un message "OK" (**)
    incrémenter compteur
    si compteur==N, passer à l'état Utilisation

  – Utilisation
    Lancer impression (*)
    Sur réception de "Demande" venant de P_j, mémoriser j. (*)

A la fin d'impression, passer à l'état Libération
– Libération
Envoyer "OK" à tous les P_j tels que j ait été mémorisé
effacer les j mémoriser
passer à l'état Libre
– Libre
sur réception de "Demande" venant de P_j, envoyer "OK" à P_j.
(*) : On ne s'intéresse que très peu à ce qui se passe en local. En effet, on suppose que localement, on dispose de ressources logicielles pour réaliser les opérations. La seule contrainte est que le processus dispose de toutes les données nécessaires aux calculs.
(**) : On considère que l'envoi et la réception de message est réalisé par la système d'exploitation.
(***) : nous verrons plus tard ce que cela veut vraiment dire.

1.2.2 Processus

Notion déjà vue dans les systèmes classiques, ne change pas. Néanmoins, afin de pouvoir participer à plusieurs protocoles simultanément, il est nécessaire que soit "associé" à chaque processus de calcul un ensemble de processus de contrôle. Ainsi, un processus de calcul doit pouvoir être "géré" dans les exclusions mutuelles , doit pouvoir accéder aux données partagées...
Il doit donc répondre à tout moment à des requêtes concernant différents protocoles en cours.
Un **site** est l'ensemble formé :
– du (ou des) processus de calcul et
– des processus assurant les protocoles de contrôles internes au système d'exploitation local et ceux spécialement mis en place pour l'application.
De fait, il nous arrivera de "confondre" site et processus : on parlera indifféremment de S_i et P_i en considérant que chaque processus de calcul est inclus dans un site différent.

1.3 Liaison

1.3.1 Propriétés structurelles

Caractérisent la topologie du maillage des liaisons logiques de communication appelé "graphe de communication".
– maillage en anneau : processus ne connait directement que son suivant (anneau unidirectionnel) ou ces deux voisins immédiats (anneau bi-directionnel).
$P_{i\neq N}$ ne connait que P_{i+1} et P_N ne connait que P_1.
– maillage en étoile : il existe un processus particulier
$P_{i\neq 1}$ ne connait que P_1 et P_1 connait tous les autres sites.
– maillage en arbre :
P_1 ne connait que ses fils (P_1 est la racine de l'arbre) ;
Si $P_{i\neq 1}$ a des fils, alors $P_{i\neq 1}$ ne connait que ses fils et son père ;

Si $P_{i\neq 1}$ n'a pas de fils alors : $P_{i\neq 1}$ ne connait que son père (processus ou terminal).

– maillage complet : type Internet $\Longrightarrow$ tous les processus se connaissent.

La connaissance de cette topologie est fondamentale lorsqu'on définit des algorithmes distribués. On peut être amené à rajouter une "couche" sur un réseau X pour simuler un maillage Y. Ainsi, l'anneau peut n'être que virtuel $\Longrightarrow$ il ne faut donc pas confondre réseau physique et maillage logique.

1.3.2 Propriétés comportementales

Elles sont diverses. Les plus fréquemment rencontrées sont :

H1 : la transmission sur la voie se fait sans duplication de messages ;

H2 : la transmission se fait sans altération des messages ;

H3 : pour tout couple de processus, l'ordre de réception des messages est identique à l'ordre d'émission (pas de déséquencement) ;

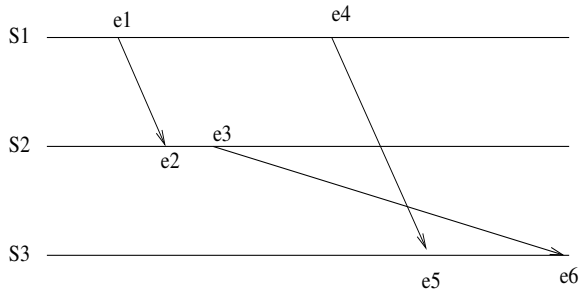
H4 : le délai d'acheminement des messages est fini : tout message envoyé est reçu au bout d'un temps fini ;

H5 : le délai d'acheminement est borné : si un message n'est pas reçu après X secondes, il est perdu.

Dans le cas où le protocole de communication garantit les hypothèses **H1**, **H2**, **H3** et **H4**, on dira que les canaux de communication sont **FIFO**.

Attention tous les canaux peuvent être **FIFO** et pourtant l"'ordre" de réception des messages sur les différents sites peut être indéterminé.

Exemple :



On ne peut déterminer l'ordre e_5 e_6 : e_6 peut très bien avoir lieu avant e_5.

Rappel : Il n'y a pas d'horloge globale $\Longrightarrow$ ce n'est pas parce que sur le dessin un événements à l'air d'avoir lieu avant un autre que c'est vrai. Il faut partir du principe que deux événements indépendants sur deux sites ne peuvent être ordonnés dans le temps grâce à une horloge "physique" globale.

ATTENTION un algorithme de système distribué est *toujours* donné (validé) en fonction d'hypothèses sur le comportement des canaux. Ainsi, fréquemment, les algorithmes sont donnés pour des canaux **FIFO**. Les faire tourner sur des canaux non **FIFO**, ne garantit plus leur fonctionnement correcte. Donc, lorsque vous définissez un algorithme distribué, il faudra *toujours* exprimer *clairement* les hypothèses de comportement des canaux (en gros, moins il y aura

d'hypothèses, "meilleur" sera votre algorithme !... (s'il fonctionne) !).

1.3.3 Trafic

Toutes propriétés identiques et pour une même fonction réalisée, un algorithme distribué est d'autant plus intéressant qu'il est performant (évident). Or le plus souvent, ce qui ralentit le plus, c'est le réseau. Il faut donc étudier précisément quels sont les coûts réseau (nombre de messages échangés principalement) engendrés par l'algorithme que vous êtes en train de définir.

1.3.4 Robustesse d'un protocole par rapport à un réseau

Soit *p* la probabilité de perte d'un message et *e* le nombre d'échanges nécessaires pour réaliser le protocole.
a) Calculons le taux d'échecs d'un protocole en fonction de *p* et de *e*.
Posons X le nombre de pertes lors d'exécution du protocole.
La probabilité $P(X = k)$ suit une loi binomiale $B(k, p)$, d'où

$$P(X = k) = \binom{k}{e} p^k (1-p)^{e-k}$$

.
Calculons la probabilité p_t pour que le protocole ne "perde" pas de message. Cela correspond au cas où $X = 0$.
Or $p_t = P(X = 0) = \binom{0}{e} p^0 (1-p)^{e-0} = (1-p)^e$.
Exemple : pour $p = \frac{1}{1000}$ et $e = 1000$, la probabilité qu'aucun message ne se perde est de $(0.999)^{1000} = 0.368$. Le protocole se plante environ 2 fois sur 3. Avec $p = \frac{1}{100}$ et $e = 1000$, la probabilité qu'aucun message ne se perde est de $(0.99)^{1000} = 0.00004$: le protocole ne marche jamais ! !
b) Voyons combien de fois il faut relancer le protocole pour qu'il y ait $x\%$ de chance de se terminer sans perte. Soit T le nombre de tentatives. Soit p_t la probabilité qu'aucun message ne se perde lors d'une tentative (par exemple, pour $p = \frac{1}{1000}$ et $e = 1000$, $p_t = 0.368$). D'où, si on veut que le protocole se termine correctement avec une probabilité d'au moins $x\%$, il faut que le probabilité qu'il se plante aux T tentatives soit inférieure à$(1 - x)$. Or la probabilité que le protocole se plante à une tentative est égale à$(1-p_t)^T < (1-x)$ d'ou $exp(T.log(p_t)) < (1-x)$ d'où $T.ln(1 - p_t) < ln(1 - x)$ d'où $T > \frac{ln(1-x)}{ln(p_t)}$.
Exemple : pour $x = 95\%$ et $p_t = 0.368$, on obtient $-0.195T < -1.301$ d'où $T > 6.6$. D'où avec 7 tentatives, le protocole à95% de chances de réussir (en fait 96%). Pour une fiabilité à99%, on a $ln(1 - 0.99) = -4.6052$ d'où $T = 10$.

1.4 Techniques d'implémentation des protocoles

Trois grande famille de protocoles :

– calcul diffusant ;
– jeton circulant ;
– estampillage.

1.4.1 Calcul diffusant

Les processus peuvent être connectés de manière quelconques. Initialement un seul processus (dit processus racine) peut émettre des messages et ensuite tout autre processus ne peut en émettre que s'il en a reçu lui-même (les demandes de calcul vont se diffuser).
On utilise ce type de calcul principalement lorsque le graphe de communication entre les processus a la structure d'arbre. Le principe de calcul est alors le suivant :
– Init : le processus **racine** émet un message de demande de calcul vers chacun de ses successeurs dans l'arbre
– Lorsqu'un processus reçoit un message de demande de calcul, il le ré-expédie vers chacun de ses successeurs
  – s'il en a
  – sinon il fait son calcul et retourne la réponse vers son père
– Lorsqu'un processus reçoit une réponse d'un de ses fils, il la traite ou la mémorise suivant le type de calcul. S'il a reçu une réponse de tous ses fils, il fait son calcul et retourne la réponse vers son père.
– Lorsque la racine a reçu toutes les réponses, il fait son calcul : le calcul diffusant est terminé.

1.4.2 Jeton circulant

La technique consiste à faire circuler un privilège dans un ensemble de processus connectés en réseau (virtuel ou non).
Exemple : modifions [1] le protocole du 1.2.1.

- Etat initial : Libre, jeton_present=Faux sauf pour P_0, util=Faux.
- Pour utiliser l'imprimante, passer à l'état Demande
  – Demande
    Si jeton_present==Vrai alors
    util=vrai
    passer à l'état Utilisation
    Sinon
    émettre la demande vers tous les autres processus
    passer à l'état Attente
  – Attente
    a la réception du message "Jeton"
    jeton_present=Vrai
    util=Vrai

1. Attention. Ce protocole n'est pas complet : il faudrait le "corriger" pour pouvoir réellement l'utiliser

passer àl'état Utilisation
- Utilisation
  Lancer impression
  Sur réception de "Demande" venant de P_j, mémoriser j.
  A la fin d'impression, passer à l'état Libération
- Libération
  util=Faux
  Si il existe un j mémorisé alors
  jeton_present=Faux
  envoyer "Jeton" à P_j
  passer à l'état Libre
- Libre Sur réception de "Demande" venant de P_j :
  Si jeton_present==Vrai alors
  jeton_present=Faux
  envoyer "Jeton" àP_j

1.4.3 Estampillage

Un des problèmes principaux des systèmes distribués est l'absence d'horloge réelle globale alors que l'ordre dans lequel surviennent les événements est primordial dans bien des cas. Ainsi :
- pour une entrée n dans une section critique, on ne peut, par exemple, prendre comme critère d'autorisation la date de la demande : une machine dont la date à1 heure d'avance sur les autres machines sera fortement avantagée ;
- de même, la date d'arrivée de réception d'un message peut du coup être inférieure à la date d'émission !
- lorsqu'une date est modifiée sur deux sites, laquelle de ces modifications est le plus récente et donc la dernière ?
- ...

En fait, comme on le verra, dans presque tous les cas, si on peut ordonner les événements liés à un problème, celui-ci est résolu ! ! Les techniques basées sur l'estampillage vont donc chercher, grâce à des échanges de messages à dater et ordonner ces événements.
On est donc amené à définir des **horloges logiques** qui vont permettre de **dater** tous les événements se produisant dans le système et ce selon une relation d'ordre total ou partiel.

1.4.3.1 Horloges logiques

Le principe est le suivant :
- si, sur un même site, l'événement e_i précède (a lieu physiquement avant) e_j alors la date de e_i doit être inférieure à celle de e_j : $D(e_i) < D(e_j)$;
- lors de l'envoi d'un message *m*, la date de l'événement $e_{emission(m)}$ correspondant à l'envoi doit être inférieure à celle de l'événement $e_{reception(m)}$ correspondant à la réception de ce message $D(e_{emission(m)}) < D(e_{reception(m)})$;

Algorithme de Lamport (1978)

Sur chaque site S_i, on trouve une variable entière H_i dite horloge locale. La date locale d'un événement E est notée $d(E)$.

$\Longrightarrow$ Pour chaque événement E ne correspondant pas ni à l'envoi, ni à la réception d'un message, S_i incrémente H_i et date cet événement par $d(E) = H_i$

$\Longrightarrow$ Lors de l'émission d'un message M par S_i, S_i incrémente H_i, estampille le message M par (H_i, i) et date l'émission par $d(E) = H_i$

$\Longrightarrow$ Lors de la réception d'un message estampillé (H_j, j) par S_i, $S - i$ recale son horloge de la manière suivante :

$$\text{si } H_i < H_j \text{ alors } H_i = H_j + 1 \text{ sinon } H_i = H_i + 1$$

Puis S_i date l'événement E de réception par $d(E) = H_i$

La date globale $D(E)$ d'un événement E est alors $(d(E), i)$ où i est le numéro du site où a eu lieu l'événement et $d(E)$ sa date locale sur ce site.

On peut aisément montrer que la relation **précède** (notée $\rightarrow$) sur les dates globales définie par :

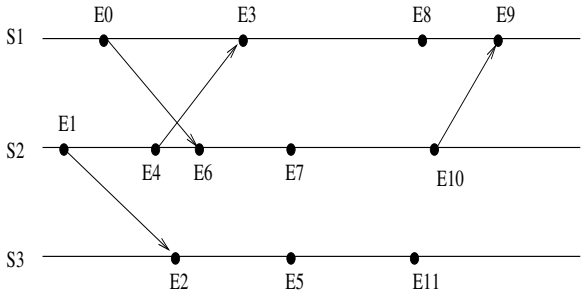
$$((d(e_1), i_1) \Longrightarrow (d(e_2), i_2))$$

$$\Updownarrow$$

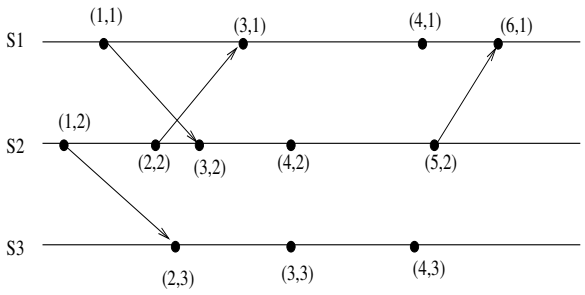
$$\begin{cases} d(e_1) < d(e_2) \\ ou \\ (d(e_1) = d(e_2) \wedge (i_1 < i_2) \end{cases}$$

est bien un ordre total strict (A $\Longrightarrow$ B est vrai, si la date globale de A< date globale de B).

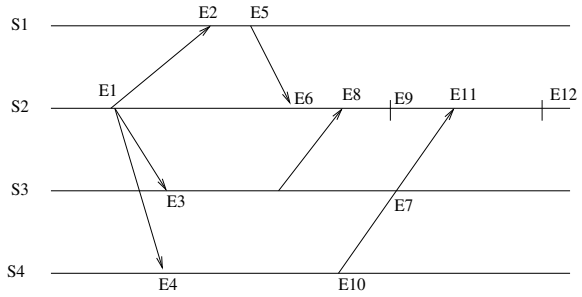
Le datage des événements dans l'exemple suivant, dans un réseau FIFO :



où les flèches représentent des envois de message, nous donne :



Problème La relation $\Longrightarrow$ ordonne bien les événements mais elle perd la causalité c'est-‡-dire l'information qu'un ou plusieurs événements ont **potentiellement** été la cause d'un autre est bien mise en évidence mais par contre le fait que deux événements sont indépendants (ou *concurrents*) : $e_i||e_j \Leftrightarrow \neg(e_j \rightarrow e_i)$ et $\neg(e_i \rightarrow e_j)$ est totalement perdu. Ainsi l'exemple suivant :



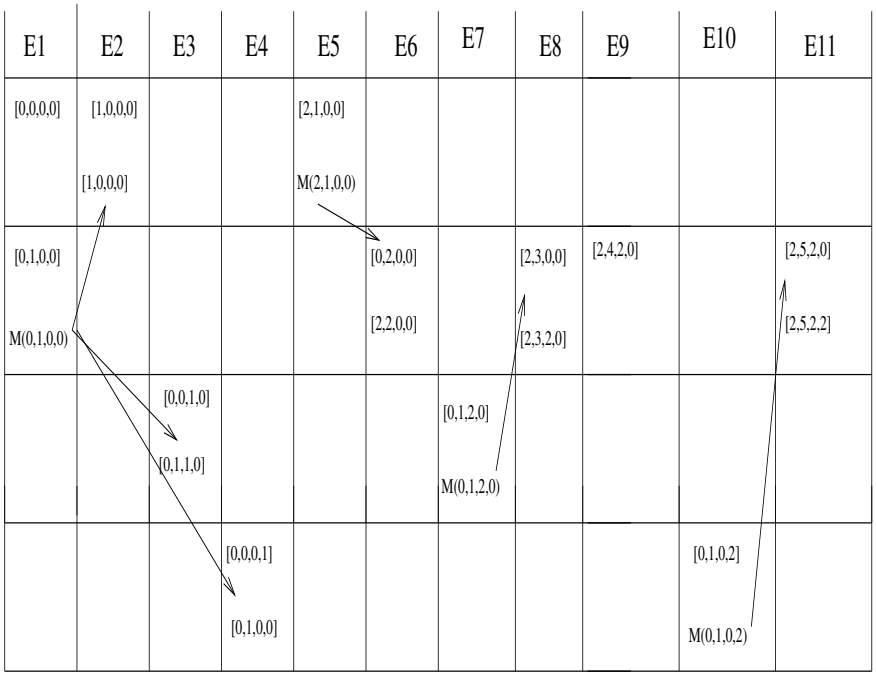
l'algorithme précèdent nous donne : $D(E_1) = (1,2)\, D(E_2) = (2,1)\, D(E_3) = (2,3)$
$D(E_4) = (2,4)\, D(E_5) = (3,1)\, D(E_6) = (4,2)\, D(E_7) = (3,3)\, D(E_8) = (5,2)\, D(E_9) = (6,2)\, D(E_{10}) = (3,4)\, D(E_{11}) = (7,2)\, D(E_{12}) = (8,2)$
et on voit que : $E_2 \Longrightarrow E_3 \Longrightarrow E_4$ mais rien ne nous dit qu'ils sont en fait indépendants.
De même

$$\begin{cases} E_2 \Longrightarrow E_9 \\ E_3 \Longrightarrow E_9 \\ E_4 \Longrightarrow E_9 \end{cases}$$

d'où le nouvel algorithme dit des **horloges vectorielles**.

1.4.4 Horloges vectorielles

Sur chaque site $S_i, i = 1, \cdots, n$, on définit une horloge vectorielle comme un vecteur $V_i[1 \cdots n]$ initialisé à 0. A chaque événement E, $V_i[i]$ est incrémenté.
– si l'événement E correspond à l'envoi d'un message M par un site S_i, M est estampillé par V_m= valeur de l'horloge V_i au moment de l'envoie
– si l'événement E correspond à la réception d'un message (M, V_m) par S_i, celui-ci exécute : $V_i[k] = max(V_i[k], V_m[k])$ pour $k = 1, \cdots, n$.
Dans tous les cas, E est, ensuite daté par V_i.
Dans l'exemple précédent cela nous donne :



1.4.5 Horloges vectorielles et relation de causalité

On remarque que pour un événement E, sur un site S_i de date V_E, $V_E[i]$ est le nombre d'événements sur S_i ayant précédé E (E compris) et que $\sum_{k=0}^{n} V_E[k]$ est le nombre total d'événements ayant précédé E. Pour deux horloges vectorielles V_1 et V_2, on définit la relation d'ordre suivante :

$$\begin{cases} V_1 \geq V_2 & \Leftrightarrow & V_1[i] \geq V_2[i] \\ V_1 \leq V_2 & \Leftrightarrow & V_1[i] \leq V_2[i] \\ V_1 || V_2 & \Leftrightarrow & \neg(V_1 \geq V_2) \wedge (V_1 \leq V_2) \end{cases}$$

La dernière "condition" peut se ré-écrire en $V_1||V_2 \Leftrightarrow \exists k_1, k_2, (V[k_1] \leq V[k_2]) \wedge (V[k_1] > V[k_2])$ Pour deux événements E_1 et E_2 : $E_1 \longrightarrow E_2 \Leftrightarrow V_{E_1}||V_{E_2}$. Dans l'exemple : $E_1 \longrightarrow E_2$? $\dashrightarrow V_{E_1} = [0,1,0,0] \leq V_{E_2} = [1,1,0,0]$ d'où E_1 est bien une cause de E_2.
$E_2 \longrightarrow E_3$? $\dashrightarrow V_{E_2} = [1,1,0,0]$ et $V_{E_3} = [0,1,1,0]$ d'où $V_{E_2}[2] \leq V_{E_3}[2]$ mais $V_{E_2}[1] > V_{E_3}[1]$ d'où $V_{E_2}||V_{E_3}$ d'où eE_2 et E_2 sont indépendants.

1.5 Conclusion

La conception d'algorithmes distribués nécessite :
– de donner les propriétés structurelles et comportementales des liaisons
– de choisir la technique à utiliser (jeton, ...)
– à donner le code

– à le valider (promela, NS2, etc.)

CHAPITRE 2

Partage et gestion de données distribuées

2.1 Introduction

Comment faire pour qu'une information soit vue par plusieurs sites comme si elle était présente localement sur chacun des sites et manipulée par des processus s'exécutant sur ce site SACHANT qu'éventuellement une copie de cette information se trouve sur chacun des sites.
D'où deux problèmes principaux :

- nommage et disponibilité de la donnée
- cohérence de celle-ci

2.2 Désignation

2.2.1 Nommage

Le nom d'un objet est formé d'un part, de son nom proprement dit et d'autre part, de la voie d'accès à cet objet $\Longrightarrow$ dépend du niveau d'abstraction.
Ainsi, par exemple, en UNIX, un fichier a 4 "noms" :

- nom symbolique local ou global
- descripteur dans une application
- descripteur en mémoire lors de l'utilisation
- inode (structure de donnée contenant des informations concernant les fichiers stockés dans un serveur. Par exemple Unix/Linux).

Un **nom interne** est une information désignant des objets et destinée à l'usage interne du système :

- pour une machine : adresse internet ou ethernet, ...
- pour des unités d'informations : inodes, numéros de pages, ...
- pour des moyens de communication : des numéros de ports, ...

La **désignation ou nommage** est la fonction qui permet :

- d'associer des noms aux objets, c'est-à-dire d'associer un nom symbolique à un nom interne interprétable par les couches basses du système $\Longrightarrow$ phase de création du nom
- d'interpréter un nom pour retrouver l'objet, c'est-à-dire de retrouver son nom interne $\Longrightarrow$ phase d'utilisation.

Ce nom doit :

- désigner l'objet de manière unique
- permettre de retrouver l'objet
- être difficile à contrefaire

Exemple : pour l'unicité :

- on peut ajouter le numéro de série de la machine à tout nom
- sur le réseau internet, on rajoute le l'adresse internet pour la sécurité, on simule les capacités par l'utilisation de fonctions de cryptage des champs

MAIS problème lors du déplacement (migration) de l'objet.

2.2.2 Migration et nom interne

Si un objet ne migre que très rarement on peut inclure dans le nom une information sur la localisation physique de l'objet. Par contre, pour les objets "mobiles", les noms doivent être "invariants" et donc ne pas dépendre de la localisation de l'objet.
Si l'objet est peu mobile :

- solution "simple" : prévenir les utilisateurs (problème : qui sont-ils ?)
- utilisation de liens de poursuite à partir de son site d'origine :
  - utilisation de liens de poursuite à partir de son site d'origine
  - problème de nombre de sites à visiter avant de le retrouver
  - problème de la rupture de la chaîne en cas de panne ou d'arrêt d'un site
  - problème de la durée de vie et la validité de ces liens

Si l'objet est très mobile :

- soit utilisation d'une base de données centralisée sur un serveur
  - problème du goulet d'étranglement
  - problème en cas de panne du serveur
  - problème de la validité de la base : si un objet bouge plus vite que la base n'est mise à jour
- soit utilisation de la diffusion
  - problème de la saturation du réseau
  - problème de l'accessibilité en diffusion

2.2.3 Recherche de noms

Comment trouver un nom interne à partir d'un nom symbolique ?

- manuellement
- utilisation de fichiers :/etc/hosts ;/etc/passwd...mais alors problème de la connaissance"initiale"et de l'évolution sur chaque machine

- serveur de désignation (NIS [1], NIS+, ...)
  tous les fichiers précédents sur les noms sont “remplacés” par des fichiers sur un seul site. **Exemple :** tous les utilisateurs sont déclarés sur une machine (exple : ada) qui stocke leur identification (nom, home, etc..) et leur mot de passe. Les machines clientes peuvent l'interroger pour accepter ou non une connexion. Ainsi, si le problème de la connaissance initiale n'est pas résolue, la mise à jour des informations est simplifiée MAIS encore, faut-il savoir que les données doivent être mise à jour : par exple, si une machine aux USA change d'adresse, elle ne va pas prévenir tous les serveur NIS sur terre ! !
- serveur de noms (DNS) : utilisé principalement pour les adresses internet.
  Tous les couples (nom symbolique, adresse internet) d'un réseau (exple : ugb.sn) sont connues par une machine serveur (exple : isis -> 130.79.200.1). Cette machine peut être interrogée par toutes les machines du monde. Ainsi si, je cherche à connaître l'adresse internet de la machine dollar.big-money.com, j'émets une requête vers le serveur du mon réseau (mail.ugb.sn) qui ne connaît pas cette adresse. Il transmet donc au serveur du réseau .sn (à St-Louis). Celui-ci transmet la demande au serveur du réseau .com qui lui la transmet au serveur du réseau big-money.com. (Il peut y avoir une étape de plus .sn -> NIC -> .com). Enfin, ce serveur donne l'adresse qui refait le chemin inverse => il suffit donc que chaque machine du réseau soit déclaré au serveur de son réseau pour que son adresse soit connue du monde entier.

2.2.4 Désignation symbolique et système de fichiers

Le problème principal dans l'union de deux sous-systèmes est que chacun a déjà son propre espace de noms.

Exemple : comment unifier deux arborescences UNIX ?

- création d'une super racine virtuelle (cas de UNIX United).
  Pour un utilisateur connecté sur M2, le fichier A2 a pour nom A2. Par contre pour un utilisateur connecté sur M1, ce fichier a pour nom /../M2/A2 => le nom dépend de la machine distante
- montage logique (cas de NFS [2])

NFS est un ensemble de logiciels permettant de partager des fichiers sur un réseau hétérogène (Unix, PC, Mac...) sur un modèle serveur/client. Toute machine peut être client et celles disposant d'un disque peuvent être en plus serveurs. Il est construit sur deux protocoles XDR et RPC et utilise UDP. Il permet une désignation transparente à l'utilisateur.

Principes de fonctionnement : Une machine désirant mettre son disque (ou une partie de son disque) à la disposition de clients exporte une partie de son arborescence qui peut alors être montée comme un répertoire local par les clients. On parle alors de répertoire distant. Ainsi si on monte le répertoire /D1/A2 d'une machine M2 sur le répertoire /A2 d'une machine M1 alors, le fichier f1 se nomme /A2/f1 sur M1 et /D1/A2/f1 sur M2. Mais dans les deux cas, l'utilisateur ne connaît pas le nom "réel" et la localisation du fichier f1. (ce pourrait être le répertoire A2 de M1 qui est monté sur /D1/A2 de M2).

1. Network Information System : Protocole client/serveur permettant la centralisation d'informations sur un réseau Unix
2. Protocole permettant à un ordinateur d'accéder à des fichiers via un réseau

Attention :
NFS est un système à serveurs sans état. Un serveur NFS ne conserve pas d'information sur l'état de ses clients => une panne d'un client n'a pas d'effet sur le serveur (par contre, un client est mis en attente lors d'une panne du serveur) car toute l'information sur la position, sur l'état des fichier (ouvert, mode ...), etc. , est dans les clients (les requêtes sont auto-suffisantes) alors que dans un système à états tel que AFS [3], des tables permettent de connaître les fichiers ouverts et par qui, la position dans ces fichiers pour chacun des clients, etc. **Avantages :**

Serveurs sans états (ou mémoire)	Serveurs à états (ou mémoire)
tolérance aux pannes	messages plus petits
Open et close facultatifs	meilleures performances
pas d'espace sur le serveur pour tables	lecture anticipée possible
nombre de fichiers ouverts illimié	idempotence plus simple
pas de problème en cas de panne client	possibilité de vérouiller un fichier

Le fait que NFS soit sans état a aussi pour conséquence que lorsqu'un serveur modifie un fichier, il ne peut prévenir ses clients. Comme chaque client comporte un cache-disque, il se pose donc une problème de cohérence et de mise à jour.
$\Longrightarrow$ Dans NFS, chaque écriture est transmise immédiatement au serveur. Un client vérifie périodiquement son cache : si une information y est depuis plus de 3 secondes, il considère qu'elle n'est plus valable. Cela a pour conséquence, une augmentation du nombre de requêtes, de mise à jour, ..., d'où une augmentation du trafic réseau, d'où une limitation du nombre de clients pouvant être servis (max de 10 à 20). Et en plus le cohérence n'est pas assurée.
Réalisation :
Le noyau doit être configuré pour pouvoir participer au partage de fichiers (tous, ou presque tous les systèmes le sont par défaut) : il y a modification du système des inodes. Les fonctions noyau concernées par des opérations d'E/S sur les fichiers décodent différemment les inodes en fonction du type du fichier. Cette information leur sont fournies grâce à l'appel de la fonction fss (file system switch). On parle alors de VINOD (virtual inod).
Administration et démons :
Le serveur

Commande	Fichiers paramètres par défaut	Commentaires
exportfs, share	/etc/exports, /etc/dfs/dfstab	exporte un répertoire
showmount		visualise les montages
nfsd		démon pour les requetes NFS
mountd		démon pour les montages NFS

Le client

3. Andrew File System : système de fichiers distribué arborescent qui permet à un ensemble de machines reliées au réseau de partager des fichiers de façon cohérente

Commande	Fichiers paramètres par défaut	Commentaires
mount	/etc/fstab	montages des répertoires distant
biod		démon assurant la gestion du cache local

Contrôle des accès :

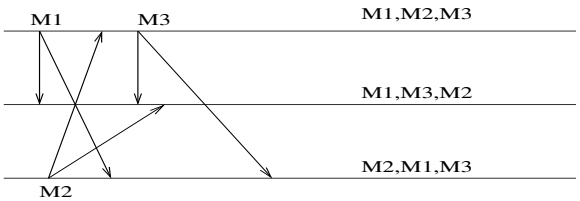
- le contrôle de la correspondance entre l'utilisateur et le propriétaire se fait par l'uid mais NFS n'unifie pas les uid sur les différentes machines. De plus, si un uid ne correspond à rien sur une machine distante, NFS lui affecte l'uid de l'utilisateur nobody. Idem pour root.
- en NFS, à chaque accès à un fichier, les droits sont vérifiés. Néanmoins, le propriétaire conserve les droits initiaux même s'il les modifie.
- Le serveur NFS autorise les lectures sur un fichier binaire exécutable même si le droit de lecture n'est pas donne ? : il ne sait pas si le client lit le fichier ou s'il est en train de le charger.

2.3 Cohérence (consistance) de données dupliquées

2.3.1 Définition

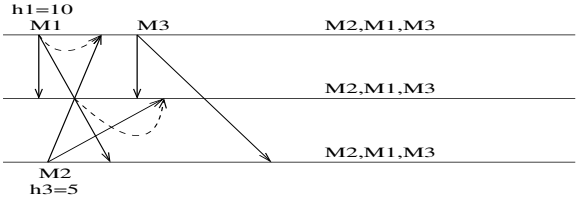
La cohérence est un problème surtout lorsque l'utilisateur ou l'application font des copies explicites d'informations ou lorsque le système ou l'application font des copies "cachées" de fichiers d'informations par utilisation de caches.
On définit trois niveaux de cohérence :

- *cohérence faible :* la mise à jour de toute copie doit avoir lieu au bout d'un temps fini.
  - l'ordre des mise à jour d'une information n'est pas nécessairement celui de ses modifications
  - une copie peut être temporairement incohérente



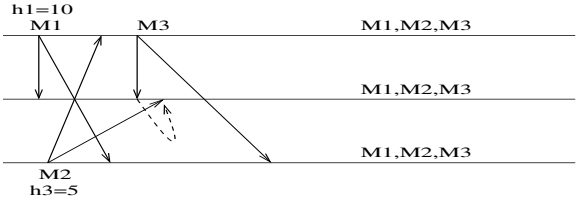
L'ordre de prise en compte des modifications Mi n'est pas la même sur tous les sites. Par contre, à un moment, la donnée doit être mise à jour : dépend de l'application. Il est évident que les opérations Mi doivent être "commutatives".

- *cohérence forte :* toute consultation doit refléter le résultat de toutes les modifications antérieures une copie lorsqu'on la consulte, est toujours à jour.
  Ainsi dans l'exemple précédent, S1 va d'abord s'assurer que sa modification est faite APRES la modification M2. Puis seulement, il l'envoie.



De même, S2 fera les modifications dans l'ordre en "retardant" éventuellement les messages.

- *cohérence causale :* si deux événements qui causent des modifications peuvent être globalement ordonnés alors les modifications correspondantes doivent être faites dans le même ordre. Ainsi, dans l'exemple précédent, les modifications M2 et M1 sont indépendantes donc peuvent être traitées dans n'importe quel ordre. Par contre, M2 est cause de M3 , elle doit donc être faite avant.



D'où, S2 retarde M3. Deux aspects à étudier : maintien de la cohérence et détection de l'incohérence

2.3.2 Caches et cohérence

On trouve deux "types" de caches : des caches mémoire (qui contiennent des données image de données de la mémoire centrale) et des caches fichiers (qui contiennent des fichiers ou des blocs de fichiers image de fichiers disque) Suivant ces types, les protocoles de maintenance de la cohérence sont différents. Il existe trois approches permettant de vérifier et valider des données dans un cache :

- vérification déclenchée par le client (cas des systèmes de fichiers répartis) : le client déclenche un contrôle soit à chaque accès à une donnée, soit à l'ouverture des fichiers uniquement ou enfin, périodiquement, à intervalle de temps fixe.
- vérification déclenchée par le serveur : le serveur enregistre pour chaque client les (parties de) fichiers que les clients mettent en caches. Quand il détecte une incohérence potentielle, il réagit.
- vérification par invalidation et espionnage (cas des caches mémoire) : dans ce cas , l'architecture la plus utilisée (et en fait, la seule vraiment réalisable) est celle fortement couplée avec un bus unique : Chaque fois qu'une donnée est écrite dans un cache, elle est aussi écrite en mémoire centrale (cache du type write-through cache). Les lectures réussies n'engendrent pas de trafic. Tous les caches surveillent constamment le bus. Chaque fois qu'un cache détecte une écriture à une adresse mémoire dont il possède le contenu, il agit (par exemple, suivant le protocole MESI). On parle alors de cache espion (snoopy cache). Ceci garantit la cohérence des caches.

Principales solutions :

- le site primaire est utilisé : il donne ou non l'autorisation de modification d'une donnée et assure la diffusion si nécessaire
- le site modificateur demande d'abord à tous les autres sites l'autorisation de modifier
- un jeton circule : il permet l'utilisation et modification de la donnée. Il contient de plus, la liste des modifications successives.

Ces trois solutions s'apparentent à une forme d'exclusion mutuelle. Elles garantissent un cohérence forte de la donnée. Mais, elles sont lourdes à mettre en place et interdissent en fait des modifications "simultanées". Or, dans certains cas, des modifications d'une donnée (ajout de données dans un fichier, ...) peuvent être faites indépendamment sur plusieurs sites, la seule contrainte est que l'ordre dans lequel ces modifications sont prises en compte soit le même sur tous les sites. D'où l'idée de dater toutes les modifications de façon globale (horloge logique ou vectorielle) et de les réaliser sur tous les sites dans cet ordre.

2.3.3 Maintenir la cohérence

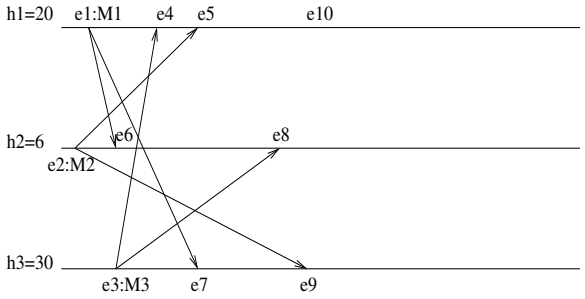
Un algorithme de maintien de la cohérence "à la demande"

Idée : Lorsqu'une modification est faite sur un site, les autres sites ne sont pas nécessairement obligés d'en tenir compte immédiatement. On va donc s'arranger pour que tous les sites soient au courant des modifications, mais que chacun décide lui-même du moment opportun de les traiter : soit parce qu'il a du temps, soit parce qu'il a besoin de connaître l'état de la donnée pour continuer, ou autre raison. On espère ainsi limiter le nombre de message dans le système.

Principe : Pour remettre à jour une donnée, un site doit réaliser toutes les modifications antérieures dues aux autres sites. On va donc maintenir un datage des événements de modifications.

Un algorithme de maintien de la cohérence "à la demande"

On suppose que le réseau est FIFO. Sur l'exemple suivant



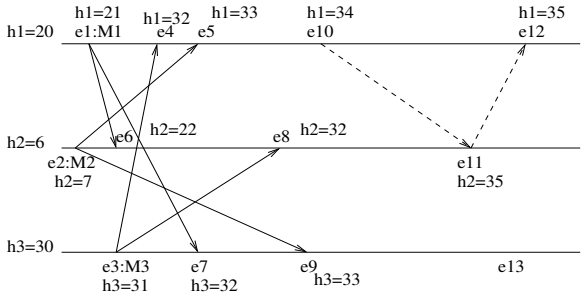
Un algorithme de maintien de la cohérence "à la demande"

Remarque : Sans algorithme, S1 traiterai M1 puis M2 puis M3 ; S2 traiterai M2 puis M1 puis M3 ; S3 traiterai M3 puis M1 puis M2

Supposons qu'en e10, S1 veuille prendre en compte sa demande de modification M1.

En e4, grâce au message M3, il sait qu'il n'a pas de demande de modification provenant de S3 de date inférieure à la date de M1car date(M3) = (31, 3). Il recale son horloge h1 = (31,s 22) + 1 = 32. Par contre en e5, il sait que date(M2) = (7, 2) < date(M1). Il recale son horloge h1 = (7, 32) + 1 = 33. Il doit traiter M2 en premier. Il ne peut toujours pas traiter M1 car rien ne lui dit que S2 n'a pas refait

une demande en (8,2). S1 va donc faire une “enquête”. En e10, S1 envoie E1(33, 1) à S2. Puis h1 + +.



En e11, S2 va recaler son horloge h2 = max(date(E1) = 33, h2 = 32) + 1= 34 et répondre par un acquittement A1 daté par (34, 2) puis incrémenter son horloge h2 = 35.
En e13, S3 devra faire une enquête auprès de S1 et S2.

L’algorithme :

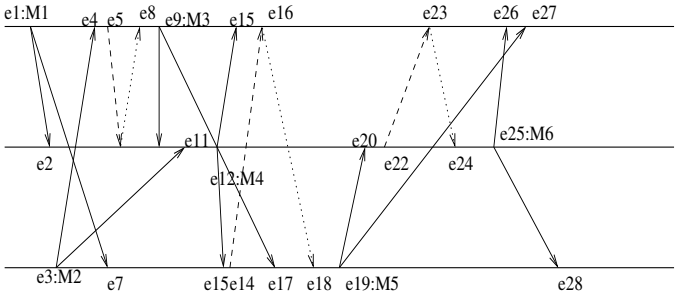
Chaque site Si dispose d’un tableau F tel que F[j] contient la liste des messages reçus par Si en provenance de Sj.
Lorsque Si veut faire modification, il la date et l’émet en diffusion.
Puis, lorsque Si veut prendre en compte les modifications, il doit exécuter toutes les modifications reçues et ce suivant leur date de modification. Pour cela, il parcours les listes contenues dans F tant qu’elles contiennent des modifications de date inférieure à la date de la modification qu’il veut prendre en compte OU qu’elles soient vides. Mais lorsqu’une liste est vide, rien ne garantit qu’une modification n’est pas en train de se faire sur un autre site ou qu’un message annonçant un modification n’est pas en train d’arriver : le site Si doit donc d’abord s’assurer que le site correspondant à une liste vide n’a pas de modification en cours.

Demande_de_modif(Mk){
Hi = Hi + 1 ;
dater Mk par Hk=Hi
diffuser Mk(Hi, i)
inserer Mk(Hk, i) dans F[i]
}

Prise_en_compte(Mk(Hk, i)){
Faire {
Vider_liste((Hk, i))
V ={$j\ tel\ que\ F[j] = \emptyset$}
$\forall j \in V$ envoyer E(Hi,i) à Sj
Attendre au moins un acquittement ou autre message de chaque site $Sj \in V$
} tantque $V \neq \emptyset$
executer Mk }

Reception Enquete(h, j)
{ Hi =max(h,Hi)+1
envoyer A(Hi,i) à Sj
}

Vider_liste(Mk(Hk , i)){
Faire {
$W = \{j/estampille\ de\ tete(F[j] < (Hk, i)\}$
Si Type de tete(F[j]) = Modif {
executer la modification
}
Enlever (tete(F [j])) }
tantque $W \neq \emptyset$
}
Reception Modif(M,h,j) {
Hi =max(h,Hi)+1
inserer M(h,j) dans F[j] }
Reception Acquittement(h,j) {
Hi =max(h,Hi)+1
inserer A(h,j) dans F[j]}
Un exemple complet :



Voyons ce que cela donne (A VERIFIER) :

	e1	e2	e3	e4	e5	e6	e7
H1	1			2			
F[1]	M1(1,1)			M1(1,1)			
F[2]				ens vide			
F[3]				M2(1,3)			
Action	M1(1,1)				E(2,1)−−S2		
H2		2				3	
F[1]		M1(1,1)					
F[2]							
F[3]							
Action						A(3,2) S1	
H3			1				2
F[1]							M1(1,1)
F[2]							
F[3]			M2(1,3)				M2(1,3)
Action			M2(1,3)				

En e5, P1décide de prendre en compte sa modification M1. Il sait qu'il ne peut plus avoir de demande de modification inférieure à la sienne provenant de P3. Par contre, il doit enquêter pour P2.
L'algorithme :

	e8	e9	e10	e11	e12	e13
H1	4	5				
F[1]	M1(1,1)	M3(5,1)				
F[2]	A(3,2)	A(3,2)				
F[3]	M2(1,3)	M2(1,3)				
Action	Traiter M1					
H2			6	7	8	
F[1]			M1(1,1)M3(5,1)	M1(1,1)M3(5,1)	M1(1,1)M3(5,1)	
F[2]					M4(8,2)	
F[3]				M2(1,3)	M2(1,3)	
Action					M4(8,2)	
H3						9
F[1]						M1(1,1)
F[2]						M4(8,2)
F[3]						M2(1,3)
Action						Traiter M1(1,1)

En e13, alors, P3 aurait pu décider de ne pas prendre en compte la modification M1. Mais, il avait un peu de temps, l'a fait. Cohérence (consistance) de données dupliquées Maintenir la cohérence **L'algorithme :**
Puis, dans la foulée, il décide de prendre en compte les autres modifications : e14

	e14	e15	e16	e17	e18	e19
H1		9	10			
F[1]		M3(5,1)	M3(5,1)			
F[2]		M4(8,2)	M4(8,2)			
F[3]		M2(1,3)	M2(1,3)			
Action			A(10,1)−−−S3			
H2						
F[1]						
F[2]						
F[3]						
Action						
H3	9			11	12	13
F[1]	∅			M3(5,1)	M3(5,1), A(10,1)	M3(5,1), A(10,1)
F[2]	M4(8,2)			M4(8,2)	M4(8,2)	M4(8,2)
F[3]	M2(1,3)			M2(1,3)	∅	M5(13,3)
Action	E(9,3) −− S1			Traiter M2(1,3)		M5(13,3)

	e20	e22	e23	e24
H1			16	
F[1]			M3(5,1)	
F[2]			M4(8,2)	
F[3]			M2(1,3)	
Action			A(16,1)−−S2	
H2	14	15		17
F[1]	M1(1,1), M3(5,1)	∅		A(16,1)
F[2]	M4(8,2)	M4(8,2)		M4(8,2)
F[3]	M2(1,3), M5(13,3)	M5(13,3)		M5(13,3)
Action	Traiter M1, M2, M3	E(15,2) −−− S1		Traiter M4(8,2)
H3				
F[1]				
F[2]				
F[3]				
Action				

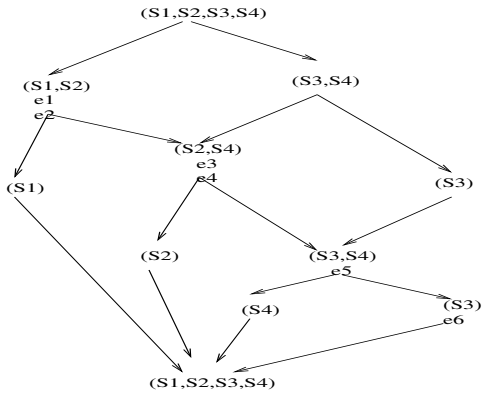
	e25	e26	e27	e28	e...
H1		20	21		
F[1]		M3(5,1)	M3(5,1)		∅
F[2]		M4(8,2), M6(19,2)	M4(8,2), M6(19,2)		∅
F[3]		M2(1,3)	M2(1,3), M5(13,3)		∅
Action					Traiter M2,M3,M4...
H2	19				
F[1]	A(16,1)				
F[2]	M6(19,2)				
F[3]	A(16,1)				
Action	M6(19,2)				Traiter M6
H3				20	
F[1]				M3(5,1), A(16,1)	A(16,1)
F[2]				M4(8,2), M6(19,2)	
F[3]				M5(13,3)	
Action					Traiter M2,M3,M4...

2.3.4 Détection de l'incohérence mutuelle

Position du problème : On considère un réseau constitué de plusieurs sites dont chacun dispose d'une copie d'une information. Deux cas se présentent :

- il n'y a aucun protocole de maintien de la cohérence => il n'y a tout simplement rien à espérer : pour voir si des copies d'une donnée sont incohérentes, il suffit (gasp) de comparer toutes les copies.
- Tous les sites suivent un même protocole qui assure la cohérence des copies il peut quand même y avoir incohérence des copies . Ainsi, dans le cas où le réseau s'est coupé en N sous-réseaux déconnectés. Chaque sous réseau peut maintenir la cohérence "locale". Mais que ce passe-t-il lorsque les sous-réseaux sont reconnectés ?

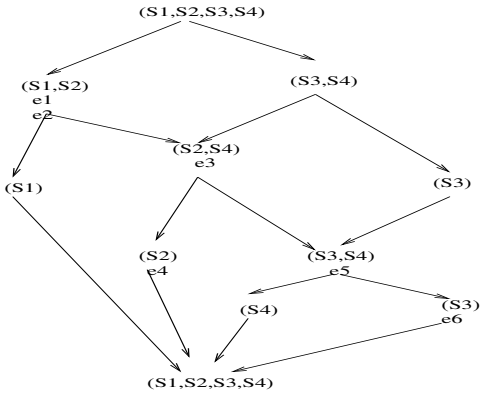
On peut représenter par un graphe l'évolution du réseau. Par exemple, un réseau à 4 sites S1, S2, S3 et S4 et l'évolution suivante :



Problèmes :

- les copies des sous-réseaux qui se reconfigurent pour n'en former plus qu'un sont-elles identiques ou non ?
- dans le cas où elles ne le sont pas, les copies d'un sous-réseau (et qui donc ont la même valeur) sont-elles "plus à jour" que les copies de l'autre sous-réseau : en d'autres termes, une convergence des deux ensembles de copies est-elle possible ou non $\Longrightarrow$ dans ce dernier cas, il y a incohérence.

Principe de l'algorithme de Parker pour la détection de l'incohérence :
Détecter l'identité de deux ensembles de copies et leur incompatibilité éventuelle nécessite de mémoriser l'histoire des ces copies sur chaque site -> on associe à chaque copie, un vecteur qui mémorise cette histoire.
Initialement V = [S1 : 0, S2 : 0 , S3 : 0, S4 : 0]
Ce vecteur indique pour chaque site, le nombre de modifications qu'a effectué le site. Prenons l'exemple des 4 sites précédents. Supposons que S1 ait modifié 1 fois la donnée (événement e1), S2 deux fois (événements e2 et e4), S3 une fois (événement e6) et S4 deux fois (événement e3 et e5).
Chaque fois que grâce à l'algorithme de maintien de cohérence, Si prend en compte une modification de la donnée due à Sk, Si incrémente $V_i[k]$.
Sur chaque site Si,on devrait avoir Vi =[S1 :1,S2 :2,S3 :1,S4 :4]
Mais si le réseau se coupe : des sites ne vont plus recevoir les informations et donc les modifications à apporter à la donnée. Ainsi, par exemple, si dès le début, S2 est coupé des autres sites alors, lorsqu'une modification aura lieu sur S2 , alors ni S1, ni S3 , ni S4 ne seront prévenus.
Dans notre exemple, supposons que, lors de la reconfiguration du réseau, on aura
V1 =[S1 :1,S2 :1,S3 :0,S4 :0] V2=[S1 :1,S2 :2,S3 :0,S4 :1]
V3 =[S1 :1,S2 :2,S3 :1,S4 :2] V4=[S1 :1,S2 :2,S3 :0,S4 :2]
On voit que V3 > V1, V3 > V2 et V3 > V4 : V3 "regroupe" les modifications qu'ont subies les copies $\Longrightarrow$ la copie associée à V3 est plus à jour : on la prend comme donnée finale. Que ce passe-t-il si e4 a lieu après la coupure de (S2, S3, S4) en (S2) ; (S3, S4) ?



On aura :
V1 =[S1 :1,S2 :1,S3 :0,S4 :0] V3 =[S1 :1,S2 :1,S3 :1,S4 :2]
V2=[S1 :1,S2 :2,S3 :0,S4 :1] V4 =[S1 :1,S2 :1,S3 :0,S4 :2]
On voit que V2 > V1 les histoires sur S1et S2sont compatibles. Idem pour S3 et S4 car V3 > V4. Par V2 et V3 sont incompatibles $\Longrightarrow$ les copies associées sont issues d'histoires différentes => il y a donc incohérence.
SI on ne trouve pas de Vi supérieur à tous les autres : PERDU $\Longrightarrow$ on peut plus reconstruire la donnée ! !

CHAPITRE 3

Diffusion

3.1 Introduction

Problématique Etant donné N sites et un réseau de communication, comment assurer une *diffusion fiable* où :

- tous les destinataires, non en pause, reçoivent les mêmes messages ;
- si l'émetteur d'un message tombe en panne pendant une diffusion, alors tous les destinataires reçoivent le message OU aucun destinataire ne reçoit le message.

A priori :

- il n'y a pas d'ordre supposé sur la réception des messages par les destinataires ;
- il n'y a pas de relation entre l'ordre d'émission et l'ordre de réception.

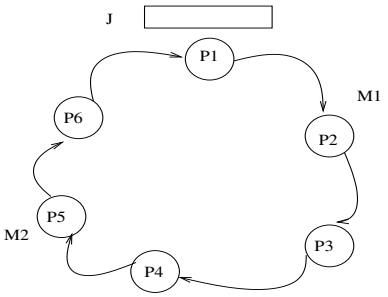
Mais Si l'ordre de délivrance est identique pour tous les récepteurs (*propriété d'uniformité*) OU si l'ordre de délivrance est identique à l'ordre causal d'émission alors la diffusion est dite *atomique*.
ATTENTION La date de délivrance d'un message peut être différente de celle de sa réception : le processus de contrôle de la diffusion ne transfert un message reçu précédemment que si ce message est le prochain devant être utilisé par le processus de calcul suivant l'algorithme utilisé. On suppose que le processus de contrôle peut stocker des messages.

3.2 Cas d'un anneau uni-directionnel

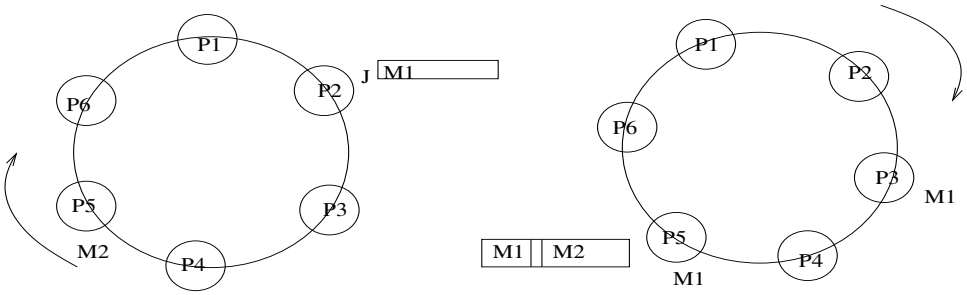
Un jeton tourne dans l'anneau.
$\Longrightarrow$ A la réception du jeton par un site S_i

- S_i traite les messages dans l'ordre donné par le jeton
- Tant que le message de tête correspond à un message émis par S_i, S_i le supprime du jeton
- Tant que S_i a un message M à émettre, il rajoute (M,i) en queue du jeton
- S_i émet le jeton vers son suivant sur l'anneau.

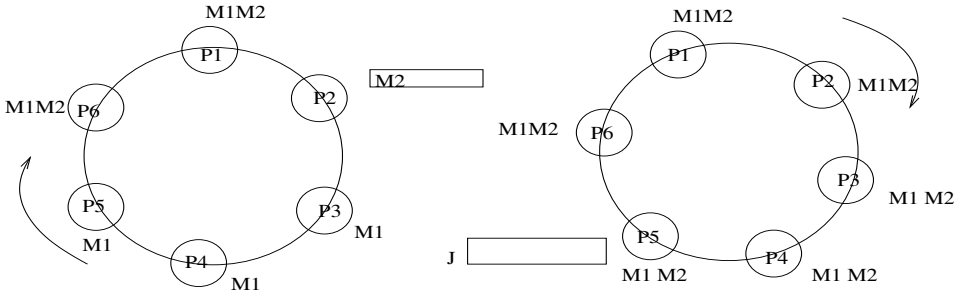
Exemple : sur l'anneau suivant, P_2 et P_5 veulent émettre en diffusion et le jeton J est en P_1.



Lorsque P_2 reçoit le jeton, il y met M1 et le transmet (figure de gauche). Lorsque le jeton passe en P_3 et P_4, ceux-ci récupèrent et traitent les messages du jeton (*en italique*). Enfin lorsque le jeton arrive en P_5, celui-ci récupère et traite les messages du jeton, rajoute M2 en queue et transmet le jeton (figure de droite).



Lorsque le jeton passe en P_6 et P_1, ceux-ci récupèrent et traitent les messages du jeton. Enfin lorsque le jeton arrive en P_2, celui-ci récupère et traite les messages du jeton, enlève M1 et transmet le jeton (figure de gauche). Lorsque le jeton passe en P_3 et P_4, ceux-ci récupèrent et traitent les messages du jeton. Enfin lorsque le jeton arrive en P_5, celui-ci récupère et traite les messages du jeton, enlève M2 et transmet le jeton (figure de droite).



3.3 Protocole utilisant un serveur

Kaashoeck 1989

Les émetteurs transmettent leurs messages au serveur qui les numérote et assure la diffusion. Ce

protocole utilise très souvent des fenêtres d'anticipation.
Le serveur S dispose d'un tampon d'émission qui lui permet de stocker T messages, d'une variable A qui donne le numéro du dernier message reçu à diffuser et d'une variable d'acquittement V qui correspond au dernier message acquitté par TOUS les destinataires. Au tampon est associé le tableau ACK de $T \times N$ bits.
Chaque site S_i dispose d'une variable R_i qui donne le numéro du dernier message reçu et acquitté.
$\Longrightarrow$ Lorsque S reçoit de S_i un message M à diffuser :

- soit le tampon n'est pas plein $A - V < T$:
  - il numérote le message par $d_M = A + 1$ et envoie (M, d_M, i)à tous les destinataires
  - il incrémente A
  - il stocke le message dans la case $c = d_M mod\ T$ du tampon T
  - il metà0 tous les cases $ACK[c][x], 1 \leq x \leq N$.
- soit le tampon est plein : il prévient l'émetteur et refuse le message.

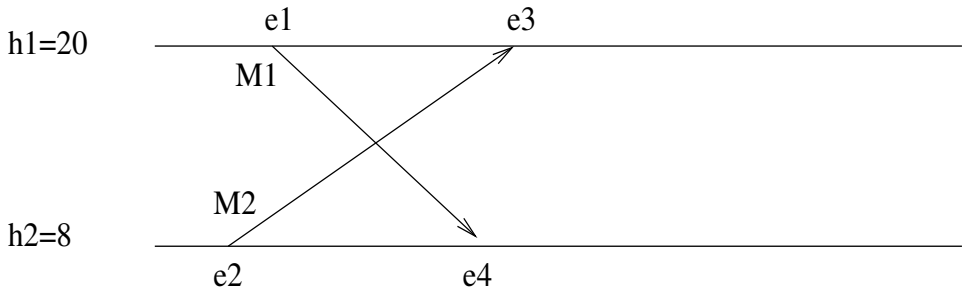
$\Longrightarrow$ Après l'envoi du message à tous les sites, S arme une time-out. A l'expiration de celui-ci, S remet le message vers les sites qui ne l'ont pas encore acquitté et réarme le time-out si nécessaire.
$\Longrightarrow$ Lorsqu'un destinataire S_k reçoit un message (M,d,i) :

- si $d > R_k + 1$, alors le destinataire stocke (si ce n'est pas déjà fait) ce message et demande au serveur la ré-émission (vers lui) des messages $(M, x, \cdots)$ tel que $d > x > R_k+1$ et $(M, x, \cdots)$ n'est pas stocké par le destinataire.
- si $d < R + 1$ alors le destinataire émet l'acquittement du message (A, d, k) vers le serveur
- si $d = R_k + 1$ alors le destinataire émet l'acquittement du message (A, d, k) vers le serveur, incrémente Rk et "utilise" M.

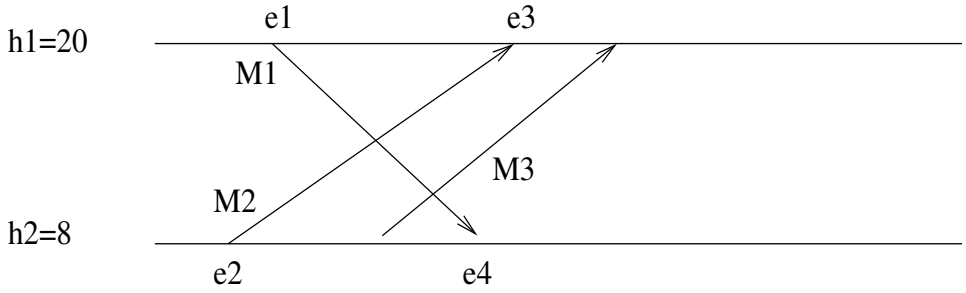
Si après une incrémentation de R_k, le destinataire trouve $(M, R_k + 1, \cdots)$ dans son stock, il émet l'acquittement du message $(A, R_k + 1, k)$ vers le serveur, "utilise" M et incrémente R_k. $\Longrightarrow$ Lorsque S reçoit un acquittement (A, d, k) du message M (qui est stocké dans la case $c = dmod\ T$ du tampon), il met à1 le bit $ACK[c][k]$. Si toutes les cases du tableau $ACK[c][x], 1 \leq x \leq N$ valent 1, le message M a été acquitté par tous les destinataires : il déstocke M. Dès que $V = d - 1$ il incrémente V.
$\Longrightarrow$ Lorsque S reçoit un message de demande de ré-émission d'un message M, il le ré-émet vers le demandeur (s'il l'a encore en stock, sinon il ne fait rien).
Cet algorithme qui assure l'uniformité par le réseau ethernet.

3.4 Protocole utilisant des estampilles

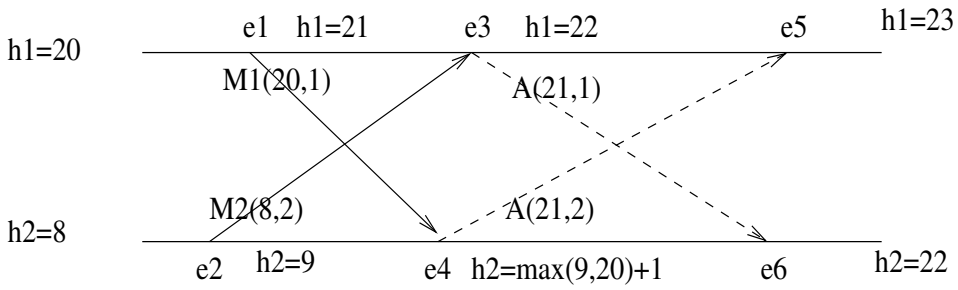
On se place dans le cas distribué sur un réseau FIFO.



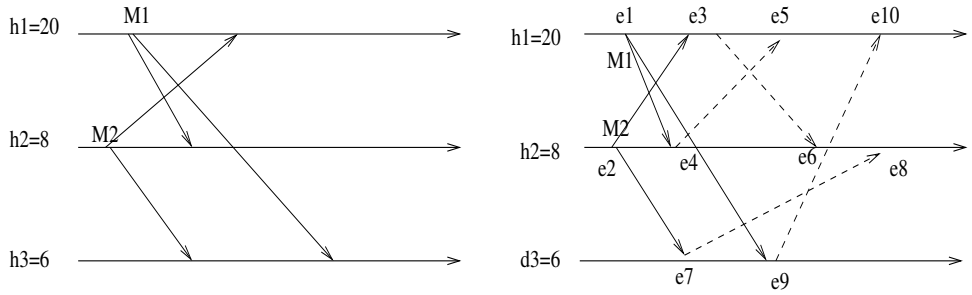
On peut utiliser les dates h_1 et h_2. Les messages en diffusion sont estampillés par la date sur le site émetteur.
en $e4$: P2 reçoit M1 daté 20. Il sait alors qu'il peut traiter M2 et M1 car il sait qu'il n'y a pas de message provenant de P1 de date inférieure à 20 car le FIFO garantit qu'il l'aurait déjà reçu.
en $e3$: P1 reçoit M2, Il sait qu'il devra traiter M2 avant M1 par contre il ne peut toujours pas traiter M1 car dans le cas suivant



rien ne dit que la date de M3 n'est pas inférieure à 20.
$\Longrightarrow$ utilisation d'acquitements datés et récalage d'horloges.

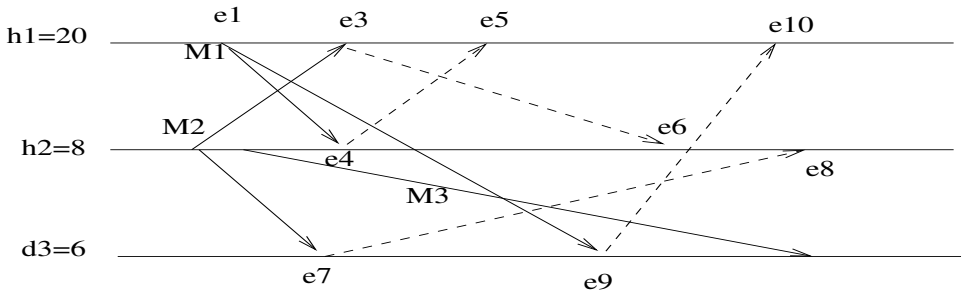


En e5, P1 sait qu'il a reçu tous les messages provenant de P2 de date inférieure à 21 (donné par l'acquittement), donc tout message venant de P2 aura donc une date supérieure à 22 donc à la date de M1, il sait qu'il peut traiter M1.
Mais que ce passe t'il dans ce cas : Réponse :



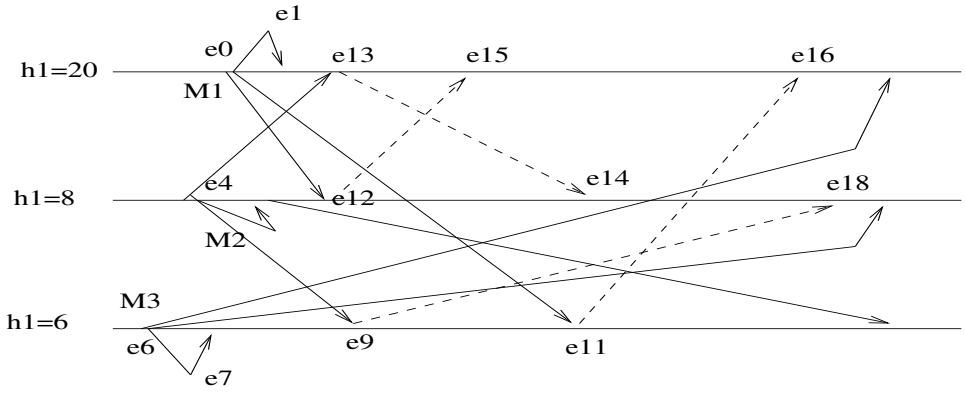
Deux problèmes :

1. P1 et P2 sont obligés d'attendre les acquittements de P3
2. P3 ne sait pas quand il peut traiter les deux messages. En effet, il peut être dans le cas suivant :



Il faut traiter M3 avant. Et on peut continuer cela longtemps. Une solution serait que les sites acquittent vers tous les autres sites et non uniquement vers l'émetteur.

Mais, cela ne résout pas le cas où **le réseau n'est pas FIFO (sauf perte de message, uniquement déséquencement)**



Pour P1 et P2, M3 n'existe pas et ils n'ont aucune raison de l'attendre !

Donc, en e18, P2 croit pouvoir utiliser M2 et M1.

Or si on conserve la date d'émission pour ordonner les message alors il faudra traiter M3 avant tous les autres messages.

D'où comment garantir que les messages seront traités dans le même ordre sur toutes les machines

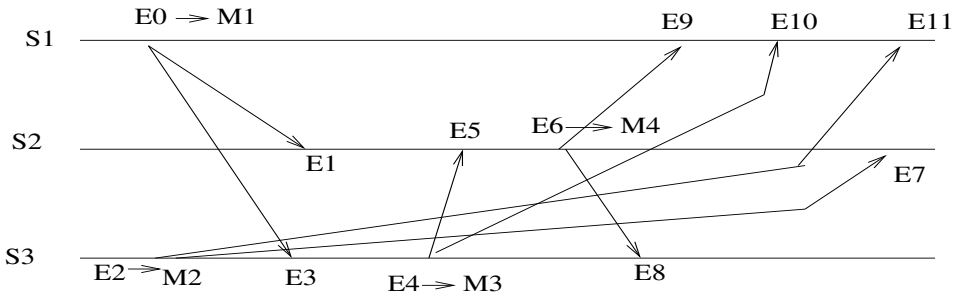
sans perdre trop de temps ?
$\Longrightarrow$ on va utiliser les dates de validation c'est-à-dire les dates où un message a été acquitté (et donc reçu) par tous les sites.

3.5 Protocole respectant l'ordre causal

Chaque site S_i entretient une horloge vectorielle $H_i[N]$. Chaque envoi de message sera daté par celle-ci : la relation de précédence causale entre les messages pourra donc être retrouvée (propriété des horloges vectorielles).

- avant de diffuser un message M, S_i traite M puis S_i exécute $H_i[i] + +$ et estampille M par $V_M = H_i$
- à la réception d'un message M envoyé par S_i et estampillé par V_M, S_j le met en attente jusqu'à ce que :
  - $V_M[i] = H_j[i] + 1$
  - et $\forall k \neq i,\ V_M[k] \leq H_j[k]$
- après traitement de M, S_j exécute : $H_j[i] + +$

Exemple :



nous donne :

TABLEAU

(*) : en effet, le message bien qu'arrivé sur S2 ne lui a pas été délivré. Donc, il n'est pas cause de M4. L'horloge vectorielle le met bien en évidence.
A noter que les trois horloges vectorielles H1, H2 et H3 ont la même valeur.
Ordre de traitement des messages : S1 : M1, M4, M2, M3 // S2 : M1, M4, M2, M3 // S3 : M2, M1, M3, M4 ce qui ne viole aucune causalité.

CHAPITRE 4

Exclusion mutuelle

4.1 Rappel

4.1.1 Propriétés des protocoles d'exclusion mutuelle

Définition : à tout instant un processus au plus se trouve en section critique (SC).
Atteignabilité : si plusieurs processus sont bloqués en attente de la S.C. alors qu'aucun processus n'est en S.C, alors l'un d'eux doit y accéder en un temps fini.
Progression : un processus en attente accède à la section critique en un temps fini.
Indépendance des parties conflictuelles et non conflictuelles : un processus hors de la S.C ou du protocole d'entrée ne doit pas influer sur le protocole d'exclusion mutuelle.
Banalisation de la solution : aucun processus ne joue le rôle privilégié.
Chaque fois que l'on proposera une solution, il faudra démontrer que ces propriétés sont bien respectées (pas toujours trivial ! !).

4.1.2 Cas des monoprocesseurs (ou mono-site)

On utilise le plus souvent des solutions du type :
- variable de condition (attente active)
- sémaphore
- moniteur
- test & set (attente active)

On celles-ci nécessitent des solution matérielles telles que soit une mémoire commune, soit un mécanisme de blocage du bus ou le masquage d'interruption $\Longrightarrow$ ceci est bien évidemment impossible sur une architecture faiblement couplée $\Longrightarrow$ utilisation de solution entièrement logicielle : protocole d'exclusion mutuelle.
Deux possibilités :

- protocole centralisé (le responsable de la ressource assure l'exclusion mutuelle). Problème : il faut qu'il y ait un tel responsable, qui en plus peut être un goulot d'étranglement
- protocole totalement décentralisé : on retrouvera ici, les deux grandes familles d'algorithmes par jeton ou estampille. Plus un algorithme particulier dans le cas où l'architecture permet le "partage" d'une mémoire (algo. de la boulangerie).

4.2 Algorithme de la boulangerie

4.2.1 Principe

Lorsque l'on entre dans une boulangerie (ça marche aussi avec une boucherie...) deux cas :

- soit il y a un système de ticket (genre Auchan) $\Longrightarrow$ ceci peut être assimilé à un serveur de ressources : pas intéressant
- soit lorsqu'on entre, on s'affecte un "numéro" en "regardant" les numéros des autres, puis on attend son tour.

L'algorithme de la boulangerie est basé sur ce principe.

4.2.2 Algorithme

Il nécessite l'utilisation de simulation d'une mémoire partagée accessible aux processus : les variables mises dans ces mémoires par un processus sont lisibles (non modifiables) par tous les autres.
Soit **num[1..N]** : un tableau de N entiers initialisés à 0
Chaque prochessus P_i maintient la case du tableau correspondant à num[i] et choix[i].
Un processus P_i doit pouvoir "lire" le tableau mais ne peut en modifier que la case **num[i]**
Pour implémenter ce tableau, on remplace l'accès direct à **num[i]**, S_i peut donc modifier **num[i]** sans problème. Lorsqu'un site S_k veut connaitre la valeur de **num[i]**, $i \neq j$, il la demandera à S_i.
On note *Entree()* la fonction de demande de la S.C, SC() la fonction d'utilisation et *Sortie()* la fonction de libération de la SC.
L'algorithme
$\triangleright Entree()$
{
// P_i est entrant :
choix[i]=vrai
$num[i] = 1 + max(num[0], \cdots, num[N-1])$ // par lecture du tableau
choix[i]=faux
// P_i en attente :
for(j=1 ;$j \neq i$; j=N-1){
{attendre que (choix[j]$\neq$ vrai)
if (num[i],i)>(num[j],j) && (num[j]$\neq$ 0)
attendre que num[j] redevienne égal à 0
}
$\triangleright Sortie()\{num[i] = 0\}$

4.2.3 Preuve

- **pas d'interblocage**
  Supposons que P_i et P_k soient en attente et que P_i était en attente avant P_k.
  Alors num[i]<num[k] $\Longrightarrow$ P_i s'exécutera avant P_k $\longrightarrow$ le protocole se comporte comme une FIFO $\longrightarrow$ un processus ne peut qu'avancer dans la liste d'attente $\longrightarrow$ il s'exécutera dans un temps fini.
  De plus, même si P_i et P_k ont exécuté au même moment leur partie entrante $\longrightarrow$ ils peuvent avoir le même numéro $\longrightarrow$ pas grave : le choix se fera sur le numéro de processus qui lui est différent pour tous les processus.
- **exclusion mutuelle**
  Montrons que : si P_i est en SC et P_k est en attente, alors P_k ne peut entrer en SC.
  Cela revient à montrer que nécessairement : (num[k],k)>(num[i],i) et num[i]$\neq$ -1 lorsque P_k le testera.
  Soit :
  - T_{a0} : l'instant où P_i écrit num[i]
  - T_{a1} : l'instant où P_i lit son choix[k] pour la dernière fois (pour j==k) $\Longrightarrow$ à ce moment, choix[k]=faux
  - T_{a2} : l'instant où P_i termine sa dernière exécution pour j==k de la seconde attente : attendre que num[j] redevienne égal à 0
    remarque 1 : $T_{a1} < T_{a2}$:
  - T_{k1} : l'instant où P_k entre dans la zone "entrant"
  - T_{k2} : l'instant où P_k écrit num[k]
  - T_{k3} : l'instant où P_k sort de la zone "entrant"
  - T_{k4} : l'instant où P_k teste (num[i],i)>(num[k],k) et num[k]$\neq$ 0
    remarque 2 : $T_{k1} < T_{k2} < T_{k3} < T_{k4}$
    remarque 3 : après T_{a0}, num[i] ne change plus. De même, après T_{k2}, num[k] ne change plus.
    remarque 4 : à T_{a1} on a obligatoirement choix[k]=faux, d'où T_{a1} : ne peut pas être compris entre T_{k1} et T_{k3}
    d'où
    - soit $T_{a1} < T_{k1}$ (P_k est entré en SC avant même que P_k soit entrant) et alors num[i]<num[k] **CQFD**
    - soit $T_{k3} < T_{a1}$ (P_i a fini son calcul de num[k] avant que P_i n'entre en SC) et alors $T_{k2} < T_{k3} < T_{a1} < T_{a2} \Longrightarrow T_{k2} < T_{a2}$ donc à T_{a1}, P_i a lu pour la dernière fois le num[k] qui a été choisi à T_{k2} d'où, comme il a passé cette instruction en T_{a2} d'où, comme il a passé cette instruction en T_{a2}, il a OBLIGATOIREMENT trouvé (num[i],i)>(num[k],k) et num[k]$\neq$ 0.
      Comme num[k] n'a pas changé depuis T_{k2} et num[i] n'a pas changé depuis T_{a0}, on a nécessairement $T_{a0} < T_{k2}$. D'où
      1. $T_{a0} < T_{k4}$

2. num[k] n'a pas changé depuis T_{k2} et num[i] n'a pas changé depuis T_{k2}
d'où en T_{k2} on avait déjà (num[i],i)>(num[k],k) et num[k]$\neq$ 0. D'où on l'aura obligatoirement en T_{k4} **CQFD**.

Inconvénient de cet algorithme :

- num[i] peut devenir très grand
- mise en place de la mémoire commune
- attente active : le site en attente doit scruter régulièrement les tableaux num et choix

4.3 Exclusion mutuelle basée sur un jeton

4.3.1 Principe

Un jeton unique circule dans le réseau. Seul le processus possédant celui-ci est autorisé à entrer en SC. • un processus qui veut entrer en SC : diffuse une requête et attend $\longrightarrow$ dès qu'il reçoit le jeton, il entre en SC

• un processus qui reçoit une requête :

mémorise cette requête puis

- s'il a le jeton et si il est en SC : terminé, rien à faire
- s'il a le jeton et s'il n'est pas en SC : il ré-exécute *Sortie()*
- s'il n'a pas le jeton : terminé, rien à faire

• un processus qui quitte la SC :

- si il a mémorisé une requête, il envoie le jeton
- sinon : terminé, rien à faire

4.3.2 Algorithme

On définit les types :

- JETON : un tableau de N d'entiers (pour une variable J de ce type, (J[i],i) correspondra à une date sur le site S_i).
- REQUETE={entier date ; entier émetteur}

Chaque processus P_i dispose :

- de deux variables du type JETON : tampon et demande
- deux variables booléennes : dedans et jeton_present initialisé à (mon_numero==1) **(4)**
- une variable entière : estampille

L'algorithme pour un P_i est le suivant : $\triangleright Entree()$

{ REQUETE requete ;
estampille++
requete.emetteur=mon_numero ; requete.date=estampille ;
demande[mon_numero]=estampille ;
diffuser(requete) /* uniquement aux autres sites */ ;
Si (jeton_present==faux) attendre(tampon) ;

dedans=vrai ;
jeton_present=vrai ;
}
$\triangleright Sortie()$
{ tampon[mon_numero]=estampille ; **(2)**
dedans=faux ;
pour j=mon_numero+1, ..n,1,...,mon_numero-1 **(3)**
{ Si (demande[j]>tampon[j]) et (jeton_present==vrai) **(1)**
{ jeton_present=faux ; envoyer(tampon,j) ;}
}
}
$\triangleright$ Lorsqu'un site P_i recoit un message il exécute :
Traiter Requete(REQUETTE requette)
{ entier k=requete.emetteur ;
demande[k]=max(demande[k], requete.date) ;
Si (jeton_present==vrai) et (dedans==faux) {*Sortie()* } }
Avec :

1. Chaque site mémorise la date de la dernière demande des autres sites dans demande[j].
   Dans le processus possédant le jeton, le tableau tampon contient les dates de la dernière possession du jeton par les autres processus donc si demande[i]<tampon[i], le processus P_i a été satisfait de sa demande, il n'y a pas lieu de lui envoyer le jeton
2. Mise à jour du tampon qui sera envoyé comme jeton : le processus y met la date locale de sa dernière possession du jeton
3. Permet de "balayer" les sites de façon circulaire : attente bornée
4. initialisation : seul le processus P_1 dispose initialement du jeton.

4.4 Preuve

Exclusion mutuelle assurée ? A tout instant, il y a au plus une des variables locales jeton_present qui vaut vrai. En effet, au départ **(4)** seul jeton_present de P_1 est vrai et ensuite seule la réception du tampn (suite à attendre_jeton) modifie cette variable. Comme cet envoi est nécessairement précédé d'un jeton_present=faux, la propriété est vraie.
Progression ? S'il n'y a pas de processus dans la SC, alors un processus en attente recevra-t-il le jeton ?
Evident : dès qu'un processus a terminé, il transmet le jeton sur simple demande ou après mémorisation.
Attente bornée ? S'il y a plusieurs processus en attente, l'ordre d'"arrivée" est-il respecté pour l'entrée en SC ?
A partir du moment où tous les messages de requêtes sont arrivés à destination (temps fini), la valeur de la j-ième entrée des tableaux demande est supérieure à celle de la j-ième entrée du jeton. Comme le processus qui tient le jeton le transmet en l'explorant de façon circulaire, un processus ne peut être

précédé que de (N-1) processus (dans l'ordre d'entrée dans la SC).
Remarques : Problème de la perte du jeton.
Coût en message : 0 ou N (N-1 pour le requête, 1 pour le jeton) par demande.

4.5 Exclusion mutuelle par liste d'attente répartie

Réseau FIFO

4.5.1 Principe

Chaque site gère une copie de la file d'attente (ou une "vision" de celle-ci) mise à jour par des messages de requêtes et de libération de la SC.
Chaque site doit :

- recevoir tous ces messages de requêtes et de libérations de tous les autres sites
- savoir les mettre *dans le bon ordre*

$\Longrightarrow$ utilisation d'horloges logiques
Chaque site S_i gère un tableau $F_i[1..N]$ à N entrées où chaque $F_i[j]$ (y compris $F_i[i]$ contient un message en provenance de S_j.
Les messages sont sous la forme : *(type du message, date du message, site)* où le type d'un message peut être : *hors_SC, requete_SC ou Ack.*
à $T_0 = 0, F_i[j] = (hors\_SC, 0, j)$ pour tout i et tout j.
▷ **Demande d'entrée en section critique** par un site S_i.

- S_i émet en diffusion le message (requete_SC,H_i,i) vers les autres sites
- ce message (requete_SC,H_i,i) est mis en tête dans la file d'attente $F_i[i]$.
- H_i++

▷ **Sorite de la section critique** par un site S_i.

- S_i émet (hors_SC,H_i,i) vers les autres sites
- H_i++

▷ **Réception d'un message** par un site S_i.

version 1
Sur un site S_i, à chaque événement de réception d'un message, l'horloge H_i est recalée puis

- si le message est du type (requete,H_j,j) : émission de (Ack,H_i,i) vers S_j
- si le message est du type (Ack,H_j,j)
  - le message est placé en tête de $F_i[j]$ si tête de $F_i[j]$ n'est pas un message du type requete_SC
  - le message est ignoré sinon

version 2

Sur un site S_i, à chaque événement de réception d'un message, l'horloge H_i est recalée puis

- si le message est du type (requete,H_j,j) : émission de (Ack,H_i,i) vers S_j si tête de $F_i[i]$ n'est pas un message du type requete_SC
- si le message est du type (Ack,H_j,j), le message est placé en tête de $F_i[j]$

Dans les deux cas un site S_i s'octroie le droit d'entrer en SC lorsque le message en tête $F_i[i]$ est du type requete_SC et que son estampille est la plus ancienne de tous les messages en tête de $F_i[j], i \neq j$.

CHAPITRE 5

Interblocages

5.1 Les processus et les ressources

L'exécution d'un processus nécessite un ensemble de ressources (mémoire principale, disques, fichiers, périphériques, etc.) qui lui sont attribuées par le système d'exploitation. L'utilisation d'une ressource passe par les étapes suivantes :

- **Demande de la ressource :** Si l'on ne peut pas satisfaire la demande il faut attendre. La demande sera mise dans une table d'attente des ressources.
- **Utilisation de la ressource :** Le processus peut utiliser la ressource.
- **Libération de la ressource :** Le processus libère la ressource demandée et allouée.

Lorsqu'un processus demande un accès exclusif à une ressource déjà allouée à un autre processus, le système d'exploitation décide de le mettre en attente jusqu'à ce que la ressource demandée devienne disponible ou lui retourner un message indiquant que la ressource n'est pas disponible : réessayer plus tard.

Les ressources peuvent être de plusieurs types :

- **Reutilisables ou disponibles** Existent-elles après son utilisation ?
- Reutilisable
  - Toutes les ressources physiques.
  - Quelques unes logiques (fichiers, mutex, verrous, etc.).
- Disponibles
  - Un processus engendre une ressource et un autre l'utilise.
  - Ressources associées à la communication et à la synchronisation comme les messages, les signaux, les sémaphores, etc.
  - **D'usage partagé :** Est-ce que la ressource peut être utilisée par plu- sieurs processus en même temps ? Les ressources partagées n'affectent pas les interblocages.
  - **Avec un ou multiples exemplaires :** Existent-ils de multiples exemplaires d'une même ressource ?

– **Preémptible ou non preémptible :** Est-ce qu'on a le droit de retirer une ressource quand elle est utilisée par un processus ?

La différence principale entre ressources préemptibles et non préemptibles est que les premières peuvent être retirées sans risque au processus qui les détient, tandis que les deuxièmes ne peuvent être retirées sans provoquer des problèmes. Comme exemples de ressources preémptibles nous avons le processeur (où le changement de contexte correspond à l'expropriation et l'état du processeur est copié à la **Table de Contrôle de Processus (BCP))** et la mémoire virtuelle (le remplacement est une expropriation et le contenu de la page doit être copié au swap). Les imprimantes et les scanners sont des exemples de ressources non preémptibles. Pour étudier le problème des interblocages, nous allons considérer uniquement les ressources non préemptibles.

5.2 Définition d'un interblocage

Des problèmes peuvent survenir, lorsque les processus obtiennent des accès exclusifs aux ressources. Par exemple, un processus P_1 détient une ressource R_1 et attend une autre ressource R_2 qui est utilisée par un autre processus P_2 ; le processus P_2 détient la ressource R_2 et attend la ressource R_1. On a une situation d'**interblocage** (deadlock en anglais) car P_1 attend P_2 et P_2 attend P_1. Les deux processus vont attendre indéfiniment comme montré sur la figure 5.1.

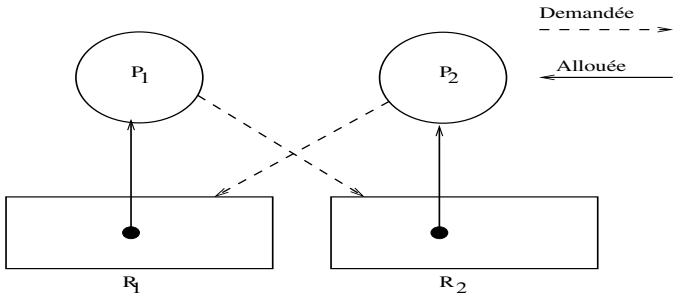


FIGURE 5.1 – Situation d'interblocage de deux processus

En général, un ensemble de processus est en interblocage si chaque processus attend la libération d'une ressource qui est allouée à un autre processus de l'ensemble. Comme tous les processus sont en attente, aucun ne pourra s'exécuter et donc libérer les ressources demandées par les autres. Ils attendront tous indéfiniment.

Exemple 1. Interblocages.

Accès aux périphériques. Supposons que deux processus A et B veulent imprimer, en utilisant la même imprimante, un fichier stocké sur une bande magnétique. La taille de ce fichier est supérieure à la capacité du disque. Chaque processus a besoin d'un accès exclusif au dérouleur et à l'imprimante simultanément. On a une situation d'interblocage si :

– Le processus A utilise l'imprimante et demande l'accès au dérouleur.
– Le processus B détient le dérouleur de bande et demande l'imprimante.

Accès à une base de données. Supposons deux processus A et B qui demandent des accès exclusifs aux enregistrements d'une base de données. On arrive à une situation d'interblocage si :

- Le processus A a verrouillé l'enregistrement R_1 et demande l'accès à l'enregistrement R_2.
- Le processus B a verrouillé l'enregistrement R_2 et demande l'accès à l'enregistrement R_1.

Circulation routière. Considérons deux routes à double sens qui se croisent, où la circulation est impossible. Un problème d'interblocage y est présent.
La résolution des interblocages constitue un point théorique très étudié. Mais, pour l'instant, il n'y a pas de solution complètement satisfaisante et la plupart des systèmes - notamment Unix - ne cherchent pas à traiter ce phénomène. Cependant, si on ne peut pas résoudre le problème dans sa généralité, on devra tenir compte de certaines techniques qui minimisent les risques. D'un autre côté, certaines tendances font de l'étude des inter- blocages un domaine très important : plus la technologie progresse, plus il y a de ressources, et un nombre grandissant de processus cohabitent dans un système.

5.2.1 Conditions nécessaires pour l'interblocage

Pour qu'une situation d'interblocage ait lieu, les quatre conditions suivantes doivent être remplies (Conditions de Coffman) :

- **L'exclusion mutuelle.** A un instant précis, une ressource est allouée à un seul processus.
- **La détention et l'attente.** Les processus qui détiennent des ressources peuvent en demander d'autres.
- **Pas de préemption.** Les ressources allouées à un processus sont libérées uniquement par le processus.
- **L'attente circulaire.** Il existe une chaîne de deux ou plus processus de telle manière que chaque processus dans la chaîne requiert une ressource allouée au processus suivant dans la chaîne.

Par exemple, dans le problème de circulation, le trafic est impossible. On observe que les quatre conditions d'interblocage sont bien remplies :

- **Exclusion mutuelle :** Seulement une voiture occupe un endroit particulier de la route à un instant donné.
- **Détention et attente :** Aucun voiture ne peut faire marche arrière.
- **Pas de préemption :** On ne permet pas à une voiture de pousser une autre voiture en dehors de la route.
- **Attente circulaire :** Chaque coin de la rue contient des voitures dont le mouvement dépend des voitures qui bloquent la prochaine intersection.

5.3 Graphe d'allocation des ressources

Le **graphe d'allocation des ressources** est un graphe biparti composé de deux types de nœuds et d'un ensemble d'arcs :

- **Les processus** qui sont représentés par des cercles.
- **Les ressources** qui sont représentées par des rectangles. Chaque rectangle contient autant de points qu'il y a d'exemplaires de la ressource représentée.
- Un arc orienté d'une ressource vers un processus signifie que la ressource est allouée au processus.

– Un arc orienté d'un processus vers une ressource signifie que le processus est bloqué en attente de la ressource.

Ce graphe indique pour chaque processus les ressources qu'il détient ainsi que celles qu'il demande.

Exemple 2.

Soient trois processus A, B et C qui utilisent trois ressources R, S et T comme illustré sur le tableau 5.1 :

A	B	C
Demande R	Demande S	Demande T
Demande S	Demande T	Demande R
Libère R	Libère S	Libère T
Libère S	Libère T	Libère R

TABLE 5.1 – Besoins de trois processus

Si les processus sont exécutés de facon séquentielle : A suivi de B suivi C, il n'y pas d'interblocage. Supposons maintenant que l'exécution des processus est gérée par un ordonnanceur du type circulaire. Si les instructions sont exécutées dans l'ordre : 1. A demande R
2. B demande S
3. C demande T
4. A demande S
5. B demande T
6. C demande R

On atteint une situation d'interblocage, fait qui est montré sur la figure 5.2.

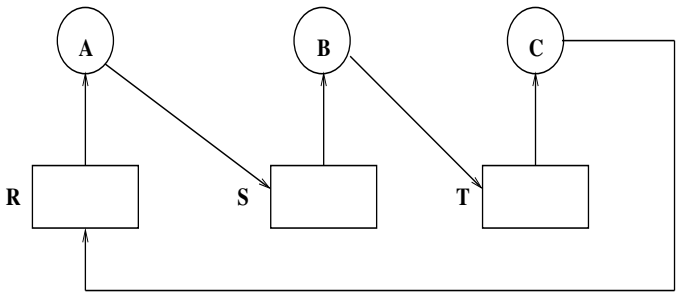


FIGURE 5.2 – Situation d'interblocage de trois processus

5.3.1 Réduction du graphe d'allocation des ressources

Un graphe réduit peut-être utilisé pour déterminer s'il existe ou non un interblocage. Pour la **réduction d'un graphe d'allocation des ressources**, les flèches associées à chaque processus et à chaque ressource doivent être vérifiées.

– Si une ressource possède seulement des flèches qui sortent (il n'y a pas des requêtes), on les efface.
– Si un processus possède seulement des flèches qui pointent vers lui, on les efface.

– Si une ressource a des flèches qui sortent, mais pour chaque flèche de requête il y a une ressource disponible dans le bloc de ressources où la flèche pointe, il faut les effacer.

Exemple 3. Considérez quatre processus $P_1, \cdots, P_n$ qui utilisent des ressources du type R_1, R_2 et R_3. Le tableau 5.2 montre l'allocation courante et le nombre maximaux d'unités de ressources nécessaires pour l'exécution des processus. Le nombre de ressources disponibles est A=[0,0,0]. Le graphe d'allocation des ressources pour l'état courant est montré sur la figure 5.3.

Processus	R_1	R_2	R_3	R_1	R_2	R_3
P_1	3	0	0	0	0	0
P_2	1	1	0	1	0	0
P_3	0	2	0	1	0	1
P_4	1	0	1	0	2	0

TABLE 5.2 – Besoins de quatre processus

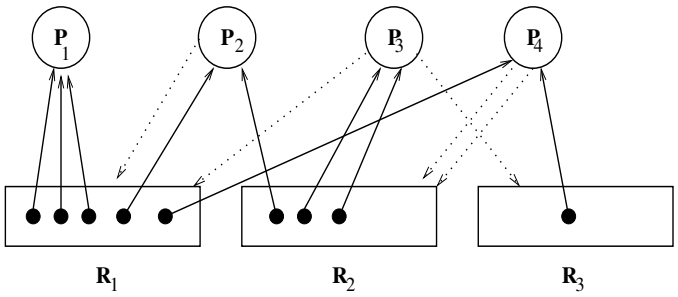


FIGURE 5.3 – Graphe d'allocation des ressources

P_1 possède seulement des flèches qui pointent vers lui, alors on les efface. Les requêtes de R_1 effectuées par P_2 peuvent donc être allouées, et P_2 aura seulement des flèches qui pointent vers lui. On les efface aussi. On efface la flèche de R_1 vers P_3 (qui a changé de direction) car R_1 n'a que des flèches sortant. Le graphe réduit final est montré sur la figure 5.4.

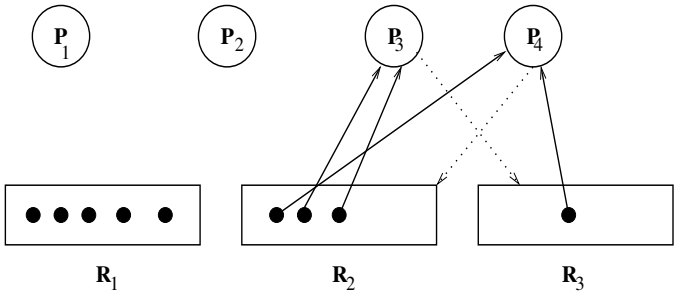


FIGURE 5.4 – Graphe réduit d'allocation

5.4 Traitement des interblocages

Comme nous l'avons mentionné, les situations d'interblocage peuvent se produire dans un système. La question qui se pose est donc : doit-il prendre en compte ce problème ou l'ignorer ?
Ignorer complètement les problèmes. On peut faire l'autruche et ignorer les possibilités d'interblocages. Cette « stratégie » est celle de la plupart des systèmes d'exploitation courants car le prix à payer pour les éviter est élevé.
Les détecter et y remédier. On tente de traiter les interblocages, en détectant les processus interbloqués et en les éliminant.
Les éviter. En allouant dynamiquement les ressources avec précaution. Le système d'exploitation peut suspendre le processus qui demande une allocation de ressource s'il constate que cette allocation peut conduire à un interblocage. Il lui attribuera la ressource lorsqu'il n'y aura plus de risque.
Les prévenir. En empêchant l'apparition de l'une des quatre conditions de leur existence.

5.5 La détection et la reprise

Dans ce cas, le système ne cherche pas à empêcher les interblocages. Il tente de les détecter et d'y remédier. Pour détecter les interblocages, il construit dynamiquement le graphe d'allocation des ressources du système qui indique les attributions et les demandes de ressources. Dans le cas des ressources à exemplaire unique, il existe un interblocage si le graphe contient au moins un cycle. Dans le cas des ressources à exemplaires multiples, il existe un interblocage si le graphe contient au moins un cycle terminal (aucun arc ne permet de le quitter). Le système vérifie s'il y a des interblocages :

- A chaque modification du graphe suite à une demande d'une ressource (coûteuse en termes de temps processeur).
- Périodiquement ou lorsque l'utilisation du processeur est inférieure à un certain seuil (la détection peut être tardive).

5.5.1 Algorithme de détection des interblocage

Soient n le nombre de processus $P[1], P[2], \cdots, P[n]$ et m le nombre de types de ressources $E[1], E[2], \cdots, E[m]$ qui existent dans un système.
L'**algorithme de détection des interblocages** utilise les matrices et vecteurs suivants :

- Matrice C des allocations courantes d'ordre ($n \times m$). L'élément C[i,j] désigne le nombre de ressources de type E[j] détenues par le processus P[i].
- Matrice R des demandes de ressources d'ordre ($n \times m$). L'élément R[i,j] est le nombre de ressources de type E[j] qui manquent au processus P[i] pour pouvoir poursuivre son exécution.
- Vecteur A d'ordre m. L'élément A[j] est le nombre de ressources de type E[j] disponibles (non attribuées).
- Vecteur E d'ordre m. L'élément E[j] est le nombre total de ressources de type j existantes dans le système.

1. Rechercher un processus P[i] non marqué dont la rangée R[i] est inférieure à A
2. Si ce processus existe, ajouter la rangée C[i] à A, marquer le processus et revenir à l'étape 1

3. Si ce processus n'existe pas, les processus non marqués sont en inter- blocage. L'algorithme se termine. On peut démontrer mathématiquement que s'il existe au moins une séquence sûre, alors il existe un nombre infini de séquences sûres. Dans les états sûrs, le système d'exploitation possède le contrôle sur les processus. Dans un état non sûr le contrôle dépends du comportement des processus.
Exemple 4. Considérons un système où nous avons trois processus en cours et qui dispose de 4 types de ressources : 4 dérouleurs de bandes, 2 scanners, 3 imprimantes et un lecteur de CD ROM. Les ressources détenues et demandées par les processus sont indique ?es par les matrices C et R.
E=[4,2,3,1] ; A=[2,1,0,0]

$$C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}, R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

Seul le processus P3 peut obtenir les ressources dont il a besoin. Il s'exécute jusqu'à la fin puis libère ses ressources, ce qui donne : A=[2,2,2,0]. Ensuite P2 peut obtenir les ressources dont il a besoin. Il s'exécute jusqu'à la fin et libère ses ressources, ce qui donne : A=[4,2,2,1]. Enfin, P1 peut obtenir les ressources dont il a besoin. Il n'y a pas d'interblocage.
Exemple 5.
Considérons un système ayant sept processus, A à G, et six ressources R à W. L'attribution des ressources est la suivante :

- A détient R et demande S ;
- B demande T ;
- C demande S ;
- D détient U et demande S et T ;
- E détient T et demande V ;
- F détient W et demande S ;
- G détient V et demande U.

Construire le graphe d'allocation des ressources ? Y a-t-il un interblocage ? Si oui, quels sont les processus concernés ?
Exemple 6.
Considérons un système ayant quatre processus, A, B. C et D, et trois types ressources R, S et T, à plusieurs exemplaires, avec 3R, 2S et 2T. L'attribution des ressources est la suivante :

- A détient une ressource de type R et demande une ressource de type S
- B détient 2 ressources de type S et demande une ressource de type R et une ressource de type T
- C détient 1 ressource de type R et demande une ressource de type S
- D détient 2 ressources de type T et demande une ressource de type R

Construire le graphe d'allocation des ressources. Y a-t-il un interblocage ? Si oui, quels sont les processus concernés ?

5.5.2 La reprise des interblocages

Lorsque le système détecte un interblocage, il doit le supprimer, ce qui se traduit généralement par la réalisation de l'une des opérations suivantes :

- Retirer temporairement une ressource à un processus pour l'attribuer à un autre.
- Restaurer un état antérieur (retour en arrière) et éviter de retomber dans la même situation.
- Supprimer un ou plusieurs processus.

5.6 L'évitement des interblocages

Dans ce cas, lorsqu'un processus demande une ressource, le système doit déterminer si l'attribution de la ressource est sûre. Si c'est le cas, il lui attribue la ressource. Sinon, la ressource n'est pas accordée. Un état est sûr si tous les processus peuvent terminer leur exécution (il existe une séquence d'allocations de ressources qui permet à tous les processus de se terminer).
Un état est non sûr si on ne peut garantir que les processus pourraient terminer leurs exécutions.
Mais comment déterminer si un état est sûr ou non sûr ? Djikstra a pro- posé en 1965 un algorithme d'ordonnancement, appelé l'Algorithme du banquier qui permet de répondre à cette question. Cet algorithme, utilise les mêmes informations que celles de l'algorithme de détection précédent : matrices A,E,C et R. Un état est caractérisé par ces quatre tableaux.

5.6.1 Algorithme du banquier

Cet algorithme (Dijistra, 1965, Habermann 1969) consiste à examiner chaque nouvelle requête pour voir si elle conduit à un état sûr. Si c'est le cas, la ressource est allouée, sinon la requête est mise en attente. L'algorithme détermine donc si un état est ou non sûr :

- 1. Trouver un processus non marqué dont la rangée de est inférieure à .
- 2. Si un tel processus n'existe pas alors l'état est non sûr (il y a interblocage). L'algorithme se termine.
- 3. Sinon, ajouter la rangée i de C à A, et marquer le processus.
- 4. Si tous les processus sont marqués alors l'état est sûr et l'algorithme se termine, sinon aller a ? l'étape 1.

L'algorithme du banquier permet bien d'éviter les interblocages mais il est peu utilisé en pratique car on ne connaît pas toujours à l'avance les besoins en ressources des processus.
Exemple 7. On considère quatre processus $P_1, \cdots, P_4$ qui utilisent des ressources du type R_1, R_2 et R_3. Le tableau 5.3 montre l'allocation courante et le nombre maximaux d'unités de ressources nécessaires pour l'exécution des processus. Le nombre de ressources disponibles est A=[0,0,0]
1. a) Construire le graphe d'allocation des ressources pour l'état courant.
2. b) L'état courant est-il sûr ?
3. c) Correspond-il à un interblocage ?

Processus	R_1	R_2	R_3	R_1	R_2	R_3
P_1	2	0	0	1	1	0
P_2	3	1	0	0	0	0
P_3	1	3	0	0	0	1
P_4	0	0	1	0	1	0

TABLE 5.3 – Besoins de quatre processus

5.7 La prévention des interblocages

Pour prévenir les interblocages, on doit éliminer une des quatre conditions nécessaires à leur apparition.

L'exclusion mutuelle. Pour éviter l'exclusion mutuelle, il est parfois possible de sérialiser les requêtes portant sur une ressource. Par exemple, pour les imprimantes, les processus « spoolent » leurs travaux dans un répertoire spécialisé et un démon d'impression les traitera, en série, l'un après l'autre.

La Détention et l'attente. Pour ce qui concerne la deuxième condition, elle pourrait être évitée si les processus demandaient leurs ressources à l'avance. Ceci est en fait très difficile à réaliser dans la pratique car l'allocation est, en général, dynamique. Empêcher cette condition serait donc particulièrement coûteux.

Pas de préemption. La troisième condition n'est pas raisonnablement traitable pour la plupart des ressources sans dégrader profondément le fonctionnement du système. On peut cependant l'envisager pour certaines ressources dont le contexte peut être sauvegardé et restauré.

L'attente circulaire. Enfin, on peut résoudre le problème de l'attente circulaire en numérotant les ressources et en n'autorisant leur demande, par un processus, que lorsqu'elles correspondent à des numéros croissants ou en accordant aux processus une seule ressource à la fois (s'il a besoin d'une autre ressource, il doit libérer la première). Par exemple :

- F(CD-ROM)=1
- F(imprimante)=2
- F(plotter)=3
- F(rubban)=4

Ainsi on garantit qu'il n'aura pas de cycles dans le graphe des ressources. On peut exiger seulement que aucun processus ne demande une ressource dont le numéro est inférieur aux ressources déjà allouées. Mais ceci n'est pas non plus la panacée, car, en général, le nombre potentiel de ressources est si important qu'il est très difficile de trouver la bonne fonction F pour les numéroter.

CHAPITRE 6

Ordonnancement et placement de processus

6.1 Introduction

6.1.1 Problématique et hypothèses

Soient un ensemble de tâches (travaux) interdépendantes et un nombre fixé de processeurs. On veut déterminer l'ordre d'exécution des tâches de façon à ce que l'exécution de l'ensemble soit le plus rapide possible sachant que l'on peut exécuter certaines tâches en parallèle.
Hypothèses applicables à un ordonnancement :

- **H1 - exécution statique :** on connaît l'ensemble de tâches, leurs durées et la structure des graphes de dépendances (c'est-à-dire l'ensemble de couples (T_i, T_j) tel que Ti doit être terminée pour que T_j puisse commencer).
- **H2 - durée invariante :** la durée d'une tâche est la même quelque soit le contexte dans lequel elle s'exécute.
- **H3 - indivisibilité :** les tâches ne sont pas pré-emptives (non morcelables).
- **H4 - communication immédiate :** il n'y a pas de délai dans les communications. T_j peut commencer dès que T_i a terminé.

Dans la suite, sauf indication contraire, les hypothèses H2, H3, H7 seront vérifiées. Hypothèses applicables à un ordonnancement :

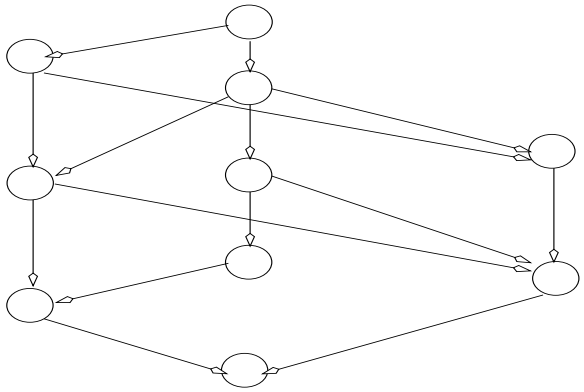
- **H5 - nombre de processeurs suffisant :** quelque soit l'ordonnancement proposé, on dispose d'assez de processeurs.
- **H6 - absence de priorités :** on ne dispose pas a priori de moyen de fixer de priorités sur les tâches.
- **H7 - ressources suffisantes :** les tâches ne sont jamais bloquées par absence de ressources (disque, mémoire, ...) et les processeurs sont suffisamment puissants pour les supporter.
- **H8 -** ...

Dans la suite, sauf indication contraire, les hypothèses H2, H3, H7 seront vérifiées.

6.1.2 Système de tâches

Soit :
- un ensemble de tâches $\{T_1, \cdots, T_n\}$ et un ensemble de durées d'exécution :$\{ex(T_1), \cdots, ex(T_n)\}$ sans communication
- et une relation de précédence ? telle que $T_i \ll T_j$ si T_i doit être terminée pour que T_j puisse commencer

On appelle *graphe de précédence* un graphe dans lequel
- les noeuds représentent les tâches ;
- deux tâches fictives T_0 dite tâche initiale, et T_{n+1} dite tâche finale, de durée nulle sont ajoutées ;
- les noeuds portent la durée de la tâche dont ils sont issus.



6.1.3 Définition d'un ordonnancement

Un ordonnancement sur p processeurs est défini comme une application Ord de $\{T_1, \cdots, T_n\}$ vers $(N, [1, \cdots, p])$ associant à chaque T_i le couple $(debut(T_i), processeur(T_i))$ où $debut(T_i)$ est la date de début de T_i et $processeur(T_i)$ le processeur qui lui est affecté telle que :
- si $T_i \ll T_j$ alors $debut(T_j) - debut(T_i) \geq ex(T_i)$
- si $processeur(T_j) = processeur(T_i)$ alors $debut(T_i) + ex(T_i) \leq debut(T_j)$ ou $debut(T_j) + ex(T_j) \leq debut(T_i)$

La condition 2 assure que deux tâches ne peuvent se dérouler en même temps sur un même processeur.
On notera plus simplement t_i la date $debut(T_i)$, appelée aussi *potentiel*.

6.1.4 Dates au tôt / au plus tard

Sur l'exemple précédent, calculons la date minimale avant qu'une tache puisse s'exécuter
$\triangleright(t_1 = 0, p_1)$ et $(t_2 = 0, p_2)$ car rien ne précède ces tâches
$\triangleright t_3 =$? Or

$$\begin{cases} t_1 \ll t3 & \Leftrightarrow & t_1 + ex(T_1) \leq t_3 \\ \text{et} & & \\ t_2 \ll t_3 & \Leftrightarrow & t_2 + ex(T_2) \leq t_3 \end{cases} \tag{6.1}$$

d'où $t_3 = max(t_1 + ex(T_1), t_2 + ex(T_2))$
$\triangleright t_8 = max(t_3 + ex(T_3), t_6 + ex(T_6)) = max(max(t_1 + ex(T_1), t_2 + ex(T_2)), t_6 + ex(T_6)) = \cdots = 10$ L'**algorithme de Bellmann** met en œuvre ce calcul :
$t_0 = 0$; marquer T_0
Tant qu'il existe des sommets non marqués faire
soit T_j un sommet non marqué dont tous les prédécesseurs T_k sont marqués [1] alors
$t_j = max\{t_k + ex(T_k)\}$
marquer T_j
finSoit
FinTantque
On remarque sur l'exemple, que T_1 peut commencer à t = 1 sans que cela modifie le temps d'exécution de l'application.
Par contre, dès que sa date de début est supérieure à 1s, l'ensemble de l'application prend du retard.
A noter que pour $T_{n+1=9}$ on obtient $t_9 = 13s$ **Chemin critique** La durée minimale de l'application est alors la valeur maximale des chemins menant de T_0 à T_{n+1}. On l'appelle *chemin critique*.
Dans l'exemple $\Rightarrow T_0, T_2, T_4, T_6, T_8, T_9$ pour 13s (qui est la date de début de T_9).
Date au plus tôt / au plus tard On appellera :

- date au plus tôt t_i pour T_i, la valeur maximale de tous les chemins menant de T_0 à T_i (La date au plus tôt t_i pour T_i est calculée de façon évidente par l'algorithme de Bellmann).
- date au plus tard di pour T_i, t_{n+1} - la valeur maximale de tous les chemins menant de T_i à T_{n+1}

En effet, la valeur d'un chemin menant de T_i à T_{n+1} est le temps que mettra au minimum la branche correspondante à s'exécuter. Donc pour que toutes les branches aient le temps de s'exécuter avant la date au plus tôt de T_{n+1} (qui est la valeur minimale d'exécution : donc si T_{n+1}ne débute pas à cet instant, l'application ne se sera pas déroulée de façon optimale) il est évident que T_i doit commencer au plus tard à t_{n+1} - le maximum de ces temps.
Dans l'exemple

i	1	2	3	4	5	6	7	8	9
Dtard;	1	0	9	4	10	6	12	10	13

On peut aisément montrer que pour les tâches T_i du chemin critique : $t_i = d_i$

6.1.5 Optimalité / minimalité d'un ordonnancement

Durée totale d'exécution d'un système de tâches La durée d'exécution totale d'exécution d'un système de taches est le temps écoulé entre le début de T_0 (tâche initiale) et la fin de T_{n+1}(tâche finale)
Durée moyenne d'exécution d'une tache dans un système de tâches La durée moyenne d'exécution d'une tache dans un système de tâches est la moyenne des temps d'exécution de chaque tâche. Le temps d'exécution d'une tâche est temps écoulé entre le moment où la tache est réalisable et sa

1. il en existe au moins un sinon c'est qu'il y a un cycle dans le graphe

fin.

Optimalité / minimalité d'un ordonnancement Un ordonnancement O est minimal si quelque soit le nombre de processeurs, il n'existe pas d'autre ordonnancement dont la durée totale d'exécution soit inférieure à celle de O.

Tout algorithme qui propose un ordonnancement tel que la durée d'exécution du système soit égal à la durée d'exécution du chemin critique est minimal.

Un ordonnancement O est optimal si pour un nombre donné de processeurs, il n'existe pas d'autre ordonnancement dont la durée totale d'exécution soit inférieure à celle de O.

En effet, il peut ne pas exister de solution minimal. Par exemple, si la somme des durées des taches divisée par le nombre de processeurs est supérieure à la durée du chemin critique, alors il ne peut pas exister de solution minimale.

Le but d'un algorithme d'ordonnancement sera donc de trouver pour tout système de taches, un ordonnancement minimal si possible , optimal sinon.

Il devra aussi chercher à minimiser le temps d'exécution moyen des taches.

6.2 Cas d'une exécution statique (H1) sans communication (H4)

6.2.1 Cas H5 : on dispose d'assez de processeurs

Si l'on dispose d'un nombre suffisant de processeurs, tout ordonnancement tel que : $\forall\, i,\ t_i \leq debut(T_i) \leq d_i$ est minimal.

Ainsi, si l'on dispose d'assez de processeurs, il suffira d'en allouer un "au hasard" à la tâche T_i dès que la date au plus tôt t_i est atteinte mais au plus tard avant que la date au plus tard di soit atteinte, pour que l'ordonnancement soit minimal.

On supprime alors souvent l'hypothèse H6 en introduisant des priorités entre les tâches et en gérant une liste des tâches exécutables (c'est-à-dire celles dont la date au plus tôt est déjà passée sans que la tâche est débutée).

6.2.1.1 Première solution dans le cas général

On applique l'algorithme précédent, puis lorsque l'on doit affecter un processeur qui se libère, on l'affecte à une tâche exécutable suivant l'ordre de priorité.

Si il n'existe pas de priorités fixées à priori, on peut prendre dans l'ordre

- les tâches dont les dates au plus tard sont les plus dépassées (ce sont donc les tâches les plus en retard)
- celles dont les dates au plus tard sont les plus proches
- et enfin celles dont les dates au plus tôt sont les plus dépassées

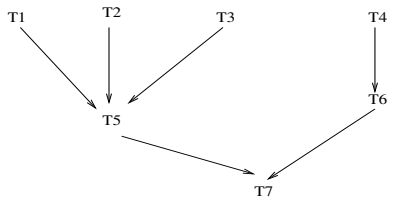
6.2.2 Cas NON-H5 : on ne dispose pas d'assez de processeurs

Malheureusement, cet algorithme ne fourni pas toujours l'ordonnancement optimal (en fait, on peut montrer que le calcul de l'ordonnancement optimal est NP-complet).

Existe-il des cas particuliers d'algorithme optimal ?

6.2.2.1 Cas d'une anti-arborescence

Le graphe est tel que chaque tâche n'a qu'un seul successeur.
Exemple :



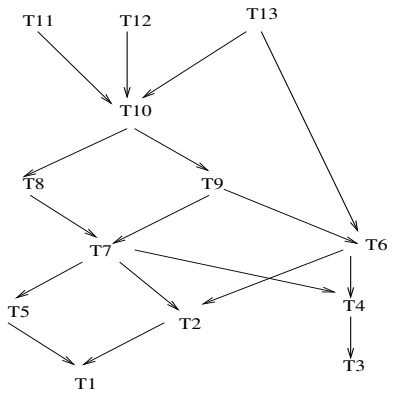
On peut montrer que dans ce cas, quelque soit le nombre de processeurs, un ordonnancement suivant la date au plus tard est optimal.
Ainsi, dans l'exemple, si chaque tâche à la même durée : T1, T2, T3, T4, T5, T6, T7 est optimal. Mais aussi T2, T4, T3, T1, T6, T5, T7.
De plus, la construction de cette liste est en O(n).

6.2.2.2 Cas d'un graphe quelconque et 2 processeurs

On classe les tâches suivant leurs dates au plus tard. Puis pour deux tâches ayant la même date au plus tard, on applique la règle suivante :
Si l'ensemble des successeurs de T_i est strictement inclus dans l'ensemble des successeurs de T_j alors T_j doit être plus prioritaire que T_i pour qu'on puisse exécuter les successeurs de T_j qui ne sont pas successeurs de T_i

6.2.2.3 Cas d'un graphe quelconque et 2 processeurs

Exemple : On suppose les tâches de durée unitaire (uniquement pour simplifier la représentation)



Ainsi, par exemple, T6 et T7 sont de même "niveau". Comme $succ(T6) \subset succ(T7)$, T7 sera plus prioritaire.
Intuitivement, en exécutant T7 avant T6, on a plus de chance de permettre la poursuite des tâches dépendantes de T7 qui ne dépendent pas de T6 : exemple T5. L'**algorithme de Coffman et Graham**, dit aussi algorithme d'étiquetage fournit une liste de priorité qui respecte la règle précédente et qui de plus tient compte des priorités des successeurs. Les durées de tâches n'ont pas d'importance.
Choisir une tâche terminale Ti : ET[Ti] = 1
Pour k=2 à N faire soit $S = \{TE1, TE2, \cdots, TEp\}$ l'ensemble de tâches étiquetables /* c'est-à-dire celles qui n'ont pas de successeurs ou celles dont tous les successeurs sont étiquetés*/
Pour chaque TEi de S faire
Calculer la liste L(TEi) = liste ordonnée décroissante des étiquettes des successeurs de TEi
fait
L'**algorithme de Coffman et Graham**
Déterminer TEm telles que L(TEm) soit inférieure ou égale à tous les L(TEi) par ordre lexicographique
ET[TEm] = k
fait
Cet algorithme fournit une liste des tâches par priorité croissante.
Il suffit alors d'ordonnancer en utilisant cette priorité et alors l'ordonnancement est optimal.

6.3 Exécution statique (H1) avec communication (Non-H4)

Hypothèse
La communication entre tâches n'a lieu qu'à la fin de la tâche émettrice pour le début de la tâche réceptrice. Ainsi, à la relation de précédence (voir "système de tâches") est associée une relation de communication : chaque arc sur le graphe de précédence sera remplacé par un arc de communication
.
On parelra alors de *graphe de communication*.

6.3.1 Communication et ordonnancement

On introduit la fonction de communication $C(T_i, T_j)$ qui donne le temps de communication entre la tâche T_i et la tâche T_j :

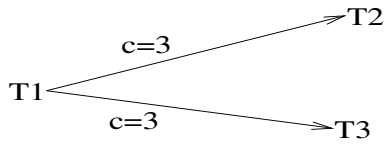
$$C(T_i, T_j) = \begin{cases} c_{i,j} \text{ si } processeur(T_i) \neq processeur(Tj) \\ 0 \text{ si } processeur(T_i) = processeur(T_j) \end{cases} \quad (6.2)$$

En fait, $C(T_i, T_j)$ vaut $c_{i,j}$ (donné par le réseau) si T_i et T_j ne sont pas sur le même processeur, 0 sinon (on considère qu'ils communiquent par mémoire partagée et que cela est "instantané"). La relation de précédence devient :

$$T_i \rightarrow T_j) \Rightarrow \begin{cases} deb(T_j) \geq deb(T_i) + ex(T_i) \text{ si } proc(T_i) = proc(T_j) \\ deb(T_j) \geq deb(T_i) + ex(T_i) + ci,j \text{ si } proc(T_i) \neq proc(T_j) \end{cases} \quad (6.3)$$

En fait, $C(T_i, T_j)$ vaut $c_{i,j}$ (donné par le réseau) si T_i et T_j ne sont pas sur le même processeur, 0 sinon (on considère qu'ils communiquent par mémoire partagée et que cela est "instantane ?"). D'où, une solution consiste à essayer de mettre sur un même processeur les tâches qui communiquent entre elle

Mais cela ne permet pas d'optimiser une situation comme par exemple, celle-ci :



En effet, a priori, on ne peut lancer T_2 et T_3 en même temps. En effet, c'est soit T_3 qui est sur le même processeur que T_1 et alors c'est T_2 doit attendre la communication 3 secondes, soit réciproquement c'est T_3 qui doit attendre.

Une solution consiste à dupliquer T_1 : sur deux processeurs différents (par exemple p_1 et p_2) on lance T_1, puis dès qu'elle se termine, on peut lancer T_2 sur un de ces deux processeurs (p_1 par exemple) et T_3 sur l'autre (p_2 dans ce cas).

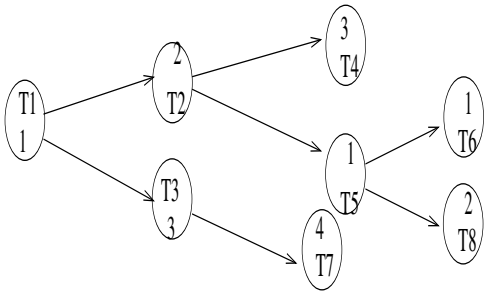
Comme il n'existe pas d'algorithme satisfaisant dans le cas où les tâches ne sont pas duplicables, nous supposerons dans la suite qu'elles le sont.

6.3.2 Cas d'un nombre de processeurs suffisant (H5)

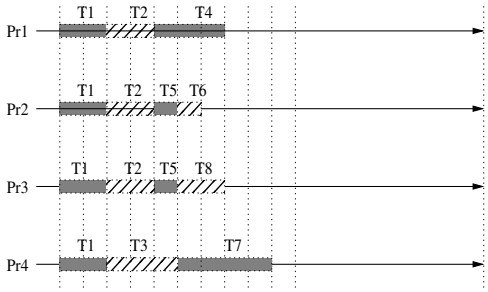
6.3.2.1 Cas d'un arbre de précédence.

Il est possible, ainsi, comme chaque tâche ne possède qu'un prédécesseur, d'associer un processeur à chaque feuille et de faire exécuter sans délai, le chemin menant de la racine à la feuille.

Exemple :



nous donne :



6.3.2.2 Cas d'un nombre de processeurs suffisant (H5)

Cas d'un graphe quelconque

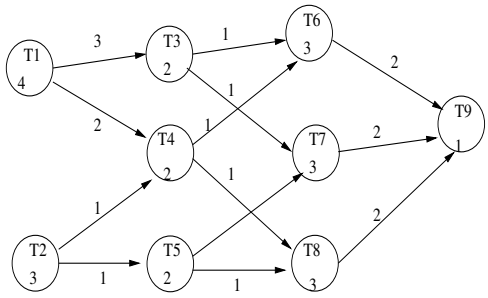
Idée : transformer le graphe en un arbre Pour chaque tâche T_i , on détermine le chemin critique menant à celle-ci. Mais, une seule tâche T_j, prédécesseur de T_i peut être mise sur le même processeur que T_i.

Question : laquelle ? *Réponse :* Celle qui retarderait le plus T_i.

D'où l'algorithme : Pour chaque tâche T_i dont les chemins critiques de tous les prédécesseurs sont déterminés

- Si T_i n'a aucun prédécesseur, on considère que T_i peut être mis sur n'importe lequel des processeurs, $s = \emptyset$
- Si T_i n'a qu'un prédécesseur T_k , on considère que T_i sera mis le même processeur que son prédécesseur et on calcule la date au plus tôt avec un temps de communication nul, s = k
- Si T_i a plus d'un prédécesseur
  - 1. on calcule tous les chemins menant à T_i en considérant que toutes les tâches sont sur un processeur différent
  - 2. soit Ts_m , la tâche prédécesseur de T_i tel que $T_0 \rightarrow \cdots \rightarrow \cdots T_{s_m} \rightarrow T_i$ soit le chemin critique
  - 3. on choisira alors de mettre T_i et T_{s_m} sur le même processeur, $s = s_m$
- 4. on peut alors recalculer le chemin critique (qui peut donc avoir diminué) qui est alors la date au plus tôt de T_i.

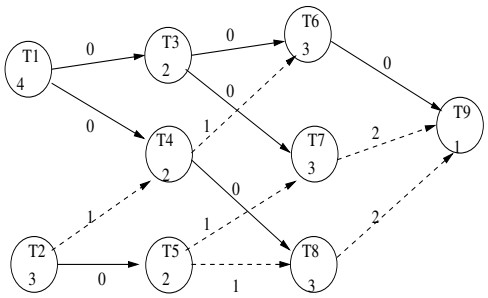
Exemple :



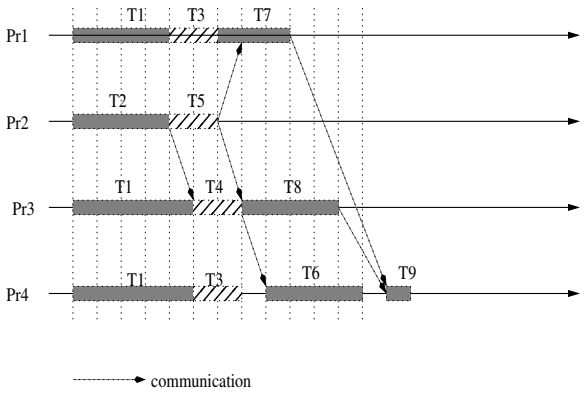
donne :

Tache	1	2	3	4	5	6	7	8	9
D_i	0	0	4	4	3	7	6	6	11
s	-	-	1	1	2	3	3	4	6

D'où l'arbre des chemins critiques (en gras)



D'où l'ordonnancement :



On peut noter que cet algorithme donne une allocation sur 4 processeurs alors que la largeur du graphe est de 3

6.3.3 Cas d'un nombre de processeurs insuffisant (Non-H5)

Pas d'algorithme satisfaisant :

- on fait l'algorithme précédent
- on éclate l'arbre obtenu en autant de sous arbres qu'il y a de processeurs
- on ordonnance sur ces processeurs en gérant des listes de tâches exécutables

6.4 Cas d'une exécution sans information préalable (NON-H1)

Il n'existe à l'heure actuelle aucun algorithme d'ordonnancement a priori satisfaisant et du plus, très souvent les processus doivent être préemptifs pour être ordonnancés. En général, l'ordonnancement se fait de façon éclatée et dynamique : chaque site gère ses processus locaux par des algorithmes classiques mono-processeur. Néanmoins, un partage des processus entre les processeurs peut être faite soit lors de la création d'un processus (il faut lui trou- ver le meilleur processeur pour son exécution) soit dynamique (on essaie d'équilibrer dynamiquement la charge des processeurs : *load balancing*)
Dans ce deuxième cas, une hypothèse supplémentaire doit être faite : les processus doivent pouvoir migrer en cours d'exécution. C'est-à-dire, le système ou le processus lui-même peuvent décider de changer le processeur affecté à un processus au cours de son exécution (en général après une reprise sur interruption) Soient les hypothèses supplémentaires suivantes :

- **HS1** quelque soit le site où il s'exécute, un processus accède toujours aux ressources (au sens physique) dont il a besoin.
- **HS2** un site est capable d'estimer sa charge (en nombre de processus et/ou en quantité de ressources utilisées ...) ;
- **HS3** un site est capable d'évaluer les besoins d'un processus.

6.4.1 Grappe de processeurs

Vaut-il mieux ordonnancer les processus sur l'ensemble des processeurs dans une seule file d'attente (approche centralisée) ou au contraire, pré-affecter les processus à des machines avec une file d'attente par machine (approche répartie) ?
Imaginons que sur une station de travail S

- les utilisateurs engendrent des demandes aléatoires de travail : soit λ le débit d'entrée de ces demandes ;
- la station est capable de traiter ces demandes avec un flux de sortie de μ.

Il est évident que si $\mu < \lambda$, la file d'attente va augmenter à l'infini. Donc, on peut admettre que $\mu < \lambda$, mais uniquement sur un court laps de temps.
On peut alors montrer que le temps moyen d'attente entre une demande et sa réponse est : $T = \dfrac{1}{\mu - \lambda}$
Par exemple, un système de fichiers peut traiter $\mu = 50$ demandes/s, alors que la file de demandes est de $\lambda = 40$ demandes/s. Le temps moyen de traitement est alors $T_i = \dfrac{1}{10}s$
Remarque lorsque λ s'approche de 0 (charge nulle), le temps moyen ne devient pas nul : en effet, si le serveur peut traiter 50 demandes/s, c'est qu'une demande dure en moyenne $\dfrac{1}{50}s$ soit 20ms. Donc, même si $\lambda = 1, T = 20ms$.
On dispose de N stations de puissance S_i équivalentes de puissance $\mu_i = \mu$. Supposons que la demande globale λ_g soit égale $\lambda_g = N.\lambda$.
$\triangleright$ Supposons que λ_g soit répartie sur les N stations : chaque station a donc $\lambda_i = \lambda = \dfrac{1}{N}\lambda_g$ demandes.
La durée moyenne de traitement d'une station est donc $T_i = \dfrac{1}{\mu_i - \lambda_i} = \dfrac{1}{\mu - \lambda}$

La moyenne des temps d'attente est donc $T = \frac{1}{N}\sum_i T_i = \frac{1}{N}.(N.\frac{1}{\mu-\lambda}) = \frac{1}{\mu-\lambda}$

▷ Que ce passe-t-il si pour la même demande $\lambda_g = N.\lambda$ on utilise un serveur de puissance $\mu_g = N.\mu$? Le temps d'attente est alors $T = \frac{1}{\mu_g - \lambda_g} = \frac{1}{N.\mu - N.\lambda} = \frac{1}{N}\frac{1}{\mu-\lambda}$. D'où un temps moyen par rapport à N serveurs divisé par N.

Mais :

- en général, le prix d'un serveur de puissance par exemple 10 000 mips est largement supérieur à celui de 10 stations de 1000 mips ;
- la tolérance au pannes est meilleure sur 10 stations car la "perte" d'une station n'est pas un catastrophe. Or les coûts de maintenance d'un serveur centralisé peuvent être prohibitifs
- e temps calculé n'est qu'un temps moyen : si un gros processus P_1 prend 95% du temps CPU (par exemple 0.95s), il bloque les 5% restant (par exemple, P_2= 0.05s) : le propriétaire de P_2 va attendre 1s ! !

D'où l'idée intermédiaire de "simuler" un serveur centralisé en regroupant les stations de travail : *grappe de processeurs*. Ainsi, la puissance théorique est $N.\mu$.

Malheureusement, cette idée n'est valable que si :

- le réseau ne pénalise pas trop les échanges (temps de chargement, algorithme d'ordonnancement, surveillance de la charge, ...)
- l'application est N-parallèlisable : en effet, si l'application ne peut être divisée qu'en par exemple 5 parties indépendantes, sur 10 processeurs, la moitié de ces processeurs ne fera rien.

En conclusion Le choix entre stations indépendantes, station en grappe et serveur centralisé dépend de la nature de la charge. Si tous les utilisateurs font des choses simples (mail, éditons de texte, projet étudiant, etc.) une station de travail individuelle de bas de gamme est suffisant. A l'opposé, s'ils utilisent principalement de gros programmes peu parallèlisés, la solution serveur centralisé s'impose. En fait, dans la réalité, très souvent, on se trouve dans une situation intermédiaire dans laquelle les utilisateurs font majoritairement des choses simples mais où ponctuellement, ils utilisent de gros logiciels (atelier de génie logiciel, calcul intensifs) de plus en plus souvent fortement parallélisé soit dans le code directement soit par une implantation multi-processus (clients-serveurs). L'utilisation de grappes de processeurs semble la plus adéquate dans cette situation.

6.4.2 Ordonnancement sans migration

6.4.2.1 Algorithme centralisé

Principe Un coordinateur gère une table d'utilisation comportant une entrée par utilisateur (ou application).

Ce coordinateur cherche a donner à chaque utilisateur ou application une part équitable de la puissance de calcul.

On suppose que chaque application ou utilisateur se voit affecter une machine particulière.

Ainsi, quand un processus doit être créé et lorsque la machine hôte de l'utilisateur décide que ce processus ne peut lui être affecté, elle demande au coordinateur de lui trouver un autre processeur :

- il en existe un libre et non demandé : affectation de ce processeur et FIN

– sinon : demande enregistrée mais rejetée

Quand un utilisateur exécute des processus sur d'autres machines que la sienne, il accumule des points de pénalité à chaque seconde. Par contre, lorsqu'une de ses demandes n'est pas satisfaite, on lui enlève des points de pénalité à chaque seconde. De même lorsqu'il n'utilise aucun processeur, mais avec une limite inférieure donnée.
Ainsi, lorsque sa note de pénalité est positive : l'utilisateur "sur-exploite" le système. Dans le cas contraire (<0) il a besoin de ressources.
D'où, lorsqu'un processeur se libère, il est affecté à l'utilisateur en attente ayant la note minimale.

6.4.2.2 Algorithme distribué

Trois approches lors du choix du processeur pour une tâche :

– au hasard : la machine qui ne peut accepter la tâche, tire au hasard une autre machine et lui envoie le processus à créer. Si cette machine est elle-même saturée, elle fait la même chose.
  → l'algorithme se déroule jusqu'à ce qu'une machine accepte le processus ou que celui-ci ait fait un nombre de sauts maximum (dans ce cas, il s'exécute là où il se trouve)
– sondage par tirage au sort : la machine qui ne peut accepter la tâche, tire au hasard une autre machine et lui envoi un message de sondage en lui demandant si sa charge est inférieure à un seuil donné. Si oui, elle lui envoie le processus, sinon, elle envoie la sonde à une autre machine.

Si au bout de S sondages, aucune machine n'accepte le processus, alors celui-ci s'exécute sur la machine d'origine Cet algorithme est stable et satisfaisant quelque soit le seuil fixé, le coût de transfert et le nombre de processus et de machines.

– sondage précis : la machine qui ne peut accepter la tâche, envoie message de sondage a toutes les autres machines en leurs demandant si leurs charges. Elle choisit la "meilleure".
  Problème : le temps de sondage peut être tel que les valeurs reçues par la machine ne soit plus à jour → la machine choisie n'est peut-être plus la meilleure ou pire elle est elle-même saturée (elle recommencera le sondage !).
  En fait pour être efficace, cet algorithme nécessite une complexité importante.

6.4.3 Ordonnancement avec migration

Définition On appelle *migration* d'un processus le transfert de tout ou d'une partie du contexte (utilisateur : adressage interne ; d'état : horloge, CO ; et/ou système : droit d'accès, propriétaire, ...) d'un processus d'un site à un autre.

6.4.3.1 Algorithme centralisé

Il est évident que l'algorithme centralisé précédent s'applique sans grande difficulté. La principale différence est que la décision d'ordonnancement est prise après chaque événement d'ordonnancement (processeur demandé, processeur libéré, interruption, blocage d'un processus...)

6.4.3.2 Algorithme réparti

Deux types d'algorithmes :

- par appel d'offre : un site surchargé émet vers les autres une demande précisant les besoins (mémoire, ressources, ...) des processus dont il veut se décharger (algo distribué précédent) ;
- par recrutement : un site sous-chargé demande aux autres sites s'ils ont besoin d'aide

Exemple d'un **algorithme de recrutement** : l'algorithme de Ni, Xu et Gendreau (1985).
On définit un taux de besoin $B_{i,k}$ d'un processus P_k sur un site S_i en fonction :

- des besoins du processus
- de sa priorité à migrer (dépendant par exemple de sa priorité dans l'application)

Chaque site maintient :

- son état à trois niveaux : LC (légèrement chargé), NC (normalement chargé), FC (fortement chargé)
- une table des charges qui contient le dernier état connu de chacun des autres sites

Fonctionnement
Un site S_i dans l'état LC émet un message "prêt" à tous les sites F_i dans l'état FC (donnés par la table des charges) et attend leur réponse.
Un site F_j qui reçoit un message "prêt" de S_i, calcule les taux de besoin B_{j,k_j} de chacun de ses k_j processus susceptibles de migrer et les envoie à S_i.
Lorsque S_i a reçu tous les B_{j,k_j} de tous les sites, il calcule un taux de besoin standard B_s égal à la moyenne de tous les B_{j,k_j}
Ce taux B_s est alors transmis au site F_j qui a répondu le $B_{i,k}$ le plus grand.
Le site F_j détermine alors, parmi ses processus, le processus P_l tel que $B_{j,l}$ soit maximal sur F_j et $B_{j,l}$ soit supérieur à B_s.
Si il existe un tel processus : il enclenche la migration sinon il répond "Trop tard" à S_i.
Quelques problèmes et limitations d'un tel algorithme :

- tables de états à maintenir : est-ce les sites qui diffusent leur nouvel état à chaque changement ou est-ce chaque site qui émet vers les autres sites une demande d'état ?
- aucune tolérance face à la perte de messages ;
- un des problèmes principaux est la migration et l'augmentation simultanée de la charge de Si (par création de processus locaux ou par l'arrivée d'autres processus en migration) ce qui peut amener une oscillation des processus entre les différents sites

CHAPITRE 7

Election

7.1 Introduction

Il se pose souvent le problème du choix d'un processus parmi N afin de résoudre un problème :

- remplacement d'un serveur : reprise sur panne par exemple ;
- choix d'un initiateur d'un algorithme quelconque ;

Or, suivant le problème à résoudre, les capacités d'un processus ou site à résoudre le problème peuvent changer : par exemple, un serveur NFS peut être très capable de résoudre le problème de la panne d'un serveur de fichier. Par contre, sa capacité à déterminer si une application est terminée, peut être nulle.

Donc, lorsque le choix doit être fait, il faut tenir compte du type de problème à résoudre. On suppose donc, que pour chaque type de problème, chaque processus Pi dispose d'une capacité ci à répondre à un problème de ce type. On impose qu'il existe une relation d'ordre sur les capacité soit par $c_i \neq c_j$ si $i \neq j$; soit par $(c_i, i) < (c_j, j)$ si $c_i = c_j$ et i < j. Lors du moment de faire ce choix deux possibilités : soit l'utilisateur intervient directement ou a fixé a priori le processus à choisir, soit il a autorisé l'application à choisir elle-même ce processus en fonction des capacités de ceux-ci à résoudre le problème qui se pose. Pour cela, on utilise des algorithmes d'*élection*.

7.2 Election sur un anneau unidirectionnel

Algorithme de Chang et Roberts

Le site initiateur P_{i_0} transmet le message $El(c_{i_0}, i_0, i_0)$

Un site $P_{j \neq i_0}$ qui reçoit le message $El(c_i, i, i_0)$ transmet $El(c_j, j, i_0)$ si $c_i < c_j$ au processus suivant sur l'anneau, sinon il transmet $El(c_i, i, i_0)$.

Lorsque P_{i_0} reçoit $El(c_i, i, i_0)$, il élit le processus $P_i \Rightarrow$ phase de proclamation $\Rightarrow$ message Pl(i).

7.3 Election sur un arbre couvrant

7.3.1 Cas où l'initiateur de l'élection P_{i_0} est la racine de l'arbre

Phase 1 - Diffusion de la demande des capacités

- init : P_{i_0} effectue :
  - (cmax,imax) = (c_{i_0}, i_0) ; N_{fils} = nombre de fils
  - SI $N_{fils} > 0$, P_{i_0} émet le message *ElDemande*() à l'ensemble de ses fils
    SINON Proclamation(cmax, imax)
- Lorsqu'un site P_i reçoit le message *ElDemande*()
  - (cmax,imax) = (c_i, i) ; N_{fils} = nombre de fils
  - SI $N_{fils} > 0$, P_i émet le message *ElDemande*() à l'ensemble de ses fils
    SINON il transmet ***REP(cmax, imax)*** à son père

Phase 2 - Remontée des informations

- Lorsqu'un site P_i, reçoit le message $REP(c_j, j)$ d'un de ses fils, il effectue :
  - (cmax, imax) = $max((c_j, j), (cmax, imax))$; $N_{fils} = N_{fils} - 1$
    SI $N_{fils} == 0$
    SI P_i n'a pas de père alors Proclamation(cmax, imax)
    SINON Pi transmet REP (cmax, imax) à son père
    SINON NOP

Phase 3 - Proclamation :

- Proclamation(cmax, imax) :
  - P_{i_0} émet le message ELU(cmax,imax) vers tous ses fils SI N_{fils} = nombre de fils > 0 SINON FIN
- Lorsqu'un site P_i, reçoit le message ELU(cmax,imax)
  - il mémorise l'élu ; N_{fils} = nombre de fils
  - SI N_{fils} > 0, P_i transmet ce message ELU(cmax,imax) à l'ensemble de ses fils
    SINON il transmet OK à son père
- Lorsqu'un site P_i, reçoit le message OK d'un de ses fils, il effectue :
  - $N_{fils} = N_{fils} - 1$
    SI $N_{fils} == 0$
    SI P_i n'a pas de père alors FIN
    SINON Pi transmet OK à son père
    SINON NOP

7.3.2 Cas où le site initiateur n'est pas la racine

Utilisation de l'algorithme précédent

P_{i_0} demande à la racine de faire l'élection [1] ce qui revient en gros à l'algorithme précédent, avec une

1. ce qui revient à dire implicitement, que la capacité à résoudre le problème de l'élection est maximale pour la racine !

phase d'initialisation de l'algorithme.
Phase 0 - Initialisation de la demande d'élection : P_{i_0} émet le message InitEL() vers son père. Lorsqu'un site autre que la racine reçoit le message InitEL() il transmet ce message à son père. Lorsque la racine reçoit ce message, elle effectue l'algorithme précédent. **Principe d'un algorithme plus général**
Phase 1 - Initialisation
P_{i_0} émet le message ElDemande() vers tous ses fils et vers son père :
Lorsqu'un site non feuille reçoit le message ElDemande()
de son père, il passe à l'état descendant et il transmet ce message à l'ensemble de ses fils
d'un de ses fils, il passe à l'état ascendant et il transmet ce message a ? son père et à l'ensemble de ses fils excepté au processus fils P_f qui lui a transmis ce message.
Lorsqu'un noeud feuille P_i reçoit le message ElDemande(), il transmet à son père le message $El(c_i, i)$.
Phase 2 - Remontée des informations
Lorsqu'un site descendant $P_{j \neq i_0}$ a reçu tous les messages $El(c_i, i)$ de ses fils, il transmet à son père le message $El(c_k, k)$ tel que $c_k = max(\{c_i\}, c_j)$
Lorsqu'un site ascendant $P_{j \neq i_0}$ a reçu le message $El(c_p, p)$ de son père et $El(c_i, i)$ de tous ses fils excepté P_f il transmet à son fils P_f le message $El(c_k, k)$ tel que $c_k = max(\{c_i\}, c_p, c_j)$
Lorsque la racine P_{racine} a reçu le message $El(c_i, i)$ de tous ses fils excepté P_f il transmet à son fils P_f le message $El(c_k, k)$ tel que $c_k = max(\{c_i\}, c_{racine})$
Phase 3 - Élection
Lorsque le site P_{i_0} a reçu le message $El(c_p, p)$ de son père et $El(c_i, i)$ de tous ses fils, il élit P_k tel que $c_k = max(\{c_i\}, c_p, c_{i_0})$
Phase 4 - Proclamation
P_{i_0} proclame le résultat par l'émission du message ELU(k) vers son père et vers l'ensemble de ses fils
Lorsqu'un site reçoit le message ELU(k) , il mémorise l'information puis s'il est non feuille :
s'il a eu le message de son père, il transmet ce message ELU(k) à l'ensemble de ses fils
sinon il transmet ce message à son père et à l'ensemble de ses fils excepté au processus fils P_f qui lui a transmis le message ELU(k).

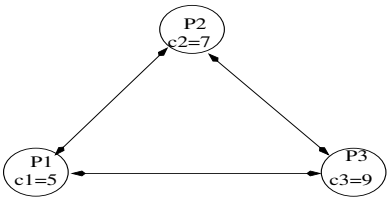
7.4 Election dans un graphe quelconque

7.4.1 Algorithme du plus fort

On se place dans le cas d'un graphe de communication FIFO à liaisons bidirectionnelles.
On suppose disposer d'un algorithme de diffusion stable.
Principe de "Bully algorithm" de Garcia-Molina
Le site initiateur Pi0 diffuse le message $El(c_{i_0}, i_0)$ à tous les processus . Puis il attend.
A la réception d'un $El(c_k, k)$ un processus P_i

- répond Ack(i) à P_k si sa capacité ci est inférieure ou égale à c_k.
- émet $El(c_i, i)$ en diffusion si sa capacité ci est strictement supérieure à c_k.

Si tous les processus répondent par un Ack à P_{i_0}, alors de façon évidente P_{i_0} à la meilleure capacité et donc il passe dans l'état Elu, et peut proclamer le résultat. Par contre, si un site P_i répond par $El(c_i, i)$ c'est que sa capacité est supérieure (P_i "prend le pouvoir") et donc, P_{i_0} arrête d'attendre et doit répondre un acquittement.
Dans l'exemple suivant :



un déroulement possible est :

Site	réception	action
P1	–	diffuse El(5,1)
P2	El(5,1)	diffuse El(7,2)
P3	El(5,1)	diffuse El(9,3)
P1	El(7,2)	Ack(1)------>P2
P1	El(9,3)	Ack(1)------->P3
P2	Ack(1)	–
P3	Ack(1)	–
P2	El(9,3)	Ack(2)-------> P3
P3	El(7,2)	diffuse El(9,3)
P1	El(9,3)	Ack(1)------->P3
P2	El(9,3)	Ack(2)-------> P3
P3	Ack(2)	Proclame
P3	Ack(1)	
P3	Ack(2)	re–proclame

Il y a 13 messages échangés (hors proclamation). En fait, on voit que le pire des cas est quand les diffusions se font dans l'ordre croissant des capacités (si c'était P_3 qui initiait l'élection le nombre de message échangés serait de 4).
Malheureusement, on ne peut rien faire pour le choix de l'initiateur. Par contre, on voit que P_3 diffuse deux fois sa demande. On peut limiter le nombre de messages en transit par "si lorsqu'un site à déjà émis un El , il ne le fait plus".

L'algorithme

D'où, le nouvel algorithme par :
Initialement, tous les P_i sont dans l'état *Peut-être*
Le site initiateur P_{i_0} diffuse le message $El(c_{i_0}, i_0)$ à tous les processus et passe à l'état Enquête. A la réception d'un $El(c_k, k)$, un processus P_i : s'il est dans l'état *Peut-être* :
il répond Ack(i) à P_k si sa capacité ci est inférieure ou égale à c_k.
il émet $El(c_i, i)$ en diffusion si sa capacité ci est strictement supérieure à c_k et passe dans l'état *Enquête* s'il est dans l'état Enquête :
il ignore ce message si sa capacité ci est supérieure à c_k.
il répond Ack(i) à P_k si sa capacité ci est inférieure ou égale à c_k et passe dans l'état Perdu [2]

2. il ne pourra plus être élu : il peut jeter tous les Ack qu'il a déjà reçu

L'algorithme
s'il est dans l'état Perdu :
il répond Ack(i) à P_k si sa capacité ci est inférieure ou égale à c_k.
Ce qui nous donne sur l'exemple :

Site	état	réception	nouvel état	action
P1	Peut etre	–	Enquete	émet El(5,1)
P2	Peut etre	El(5,1)	Enquete	émet El(7,2)
P3	Peut etre	El(5,1)	Enquete	émet El(9,3)
P1	Enquete	El(7,2)	Perdu	Ack(1)−−−−> P2
P1	Perdu	El(9,3)	Perdu	Ack(1)−−−−> P3
P2	Enquete	Ack(1)	Enquete	–
P3	Enquete	Ack(1)	Enquete	–
P2	Enquete	El(9,3)	Perdu	Ack(2)−−−−−−>
P3	Enquete	El(7,2)	Enquete	
P3	Enquete	Ack(2)	Elu	Proclame

Coût de l'algorithme
Le pire des cas est le cas où les demandes d'élection est faite par le processus de plus petite capacité car cela peut entraîner une demande d'élection de tous les sites si ceux-ci la reçoivent avant tout autre message. Ce qui nous donne :

- N - 1 messages (diffusion initiale)
- Les N - 1 autres sites émettent alors en diffusion, c'est-à-dire, chacun N - 1 messages, soit au total : (N - 1)(N - 1) messages
- Enfin, il y a $0 + 1 + 2 + \cdots + (N - 1)$ acquittements, soit N(N-1) messages.

D'où au total $\frac{3N(N-1)}{2}$ messages.
A noter qu'il serait plus simple que P_{i_0} demande à chaque site sa capité et calcule le maximum à partir de ces réponses : cout en (N-1)

7.4.2 Algorithme d'élection sans diffusion

On suppose que l'on ne dispose pas d'un algorithme de diffusion atomique fiable : un processus ne connaît que les processus avec qui il est directement connecté dans le graphe de communication. Il connaît par contre le nombre N de processus.
Principe Le site initiateur va chercher à calculer la capacité maximale des sites qu'il connait. Il va donc leur demander directe- ment celles-ci.
Phase 1 - Diffusion de la demande des capacités
Ces sites vont essayer de faire la même chose (forme de récursivité). Problème : les demandes risquent de se diffuser sans limite. En effet, un site peut recevoir une demande par différents canaux. Pour résoudre cela, un site ne refera jamais une demande. Lorsqu'il recevra une demande supplémentaire, il repondra "NOK". En effet, se capacité sera prise en compte dans la réponse qu'il fera à son premier demandeur. Elle n'a plus besoin d'être prise en compte par ce demandeur supplémentaire.
Phase 2 - Remontée des informations
Lorsqu'un site aura reçu une réponse (soit une capacité soit un NOK) de tous ses voisins, il répondra la capacité maximale à celui qui lui a fait la demande en premier.
Phase 3 - Proclamation :

Le site initiateur enverra le proclamation à tous les sites qui connait. Par contre, un site n'enverra cette proclamation qu'au sites qui ne lui ont pas répondu "NOK".

- Coût : Lors du calcul de la capacité maximale, chaque site émet une demande par site qu'il connait et reçoit une réponse. Le coût est donc égal au nombre M de liens entre les sites. Pour la proclamation, il n'y a plus que (N-1) liens "actifs" car on obtient une structure d'arbre. D'où un coût total de (M+N-1). A noter que $M \leq N$ [3]

3. il ne pourra plus être élu : il peut jeter tous les Ack qu'il a déjà reçu

CHAPITRE 8

Déterminer un état global dans un système distribué

8.1 Introduction

8.1.1 Problématique

- de détecter des propriétés caractérisant l'exécution des programmes (vérification d'assertions, terminaison, inter-blocage, ...) ;
- de calculer de performances ;
- de capturer un état de reprise en cas de panne ;
- ...

Or

- pas d'horloge globale
- temps de transfert des messages (dans le cas général) non borne

$\Longrightarrow$ impossibilité d'effectuer une observation "simultanée" de l'état des différents processus et canaux de communication. L'exemple de Chandy et Lamport est parlant : Un groupe de photographe observe une scène dynamique, par exemple un vol d'oiseaux migrateurs. La scène est si vaste qu'elle ne peut être prise par un seul photographe. Les différents photographes doivent donc se charger de prendre chacun une partie de la scène tout en sachant :

- que les photographies ne peuvent être prise au même moment
- que ces opérations ne doivent pas perturber le phénomène à observer (par exemple, il n'est pas raisonnable de demander aux oiseaux d'arrêter leur vol pendant les prises de vue)

Il faut aussi que la scène reconstituée ait "un sens" : reste à définir ce qu'est "avoir un sens" et de déterminer la manière de faire ces prises de vues.

8.1.2 Définition de l'état global

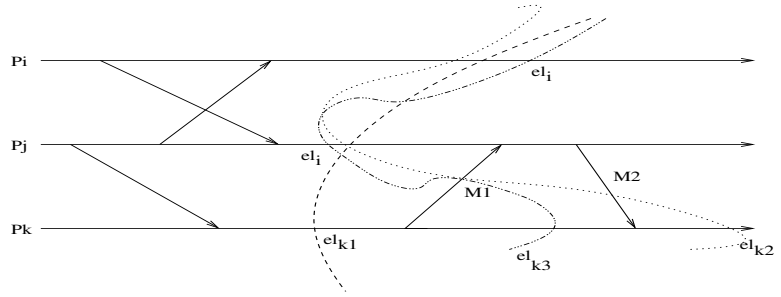
Chaque processus P_i et chaque canal Ci,j (entre P_i et P_j) possède à tout instant un état local. Soit $el_i(k)$ l'état local d'un processus P_i et $ec_{i,j}(k')$ celui d'un canal $C_{i,j}$ (ensemble des messages en transit sur $C_{i,j}$) à l'instant t.
Trois sortes d'événements peuvent survenir :

- un événement interne à un processus : provoque la transition de P_i de $el_i(k)$ à $el_i(k+1)$
- correspondant à l'émission de m par P_i sur $C_{i,j}$ et noté $emission_{i_r}(m)$: provoque la transition de P_i de $el_i(k)$ à $el_i(k+1)$ et l'affectation $ec_{ij}(k'+1) = ec_{ij}(k') \cup \{m\}$
- correspondant à la réception de m par P_i sur $C_{i,j}$ et noté $reception_{i_r}(m)$: provoque la transition de P_i de $el_i(k)$ à $el_i(k+1)$ et l'affectation $ec_{ij}(k'+1) = ec_{ij}(k')\backslash\{m\}$

Notons el_i l'état courant du processus P_i et ec_{ij} celui du canal $C_{i,j}$.
L'état global d'un système est alors : $S = \{\bigcup_i el_i, \bigcup_{i,j} ecij\}$
Définition Un état global d'un système est **cohérent** si :

- $\forall\, i,\ el_i$ est un état local de P_i
- les deux conditions suivantes sont respectées :
  - **C1 :** si l'événement $emission_{i_k}(m)$ est capté dans el_i alors l'événement $reception_{j_r}(m)$ est soit capté dans el_j, soit $m \in ec_{ij}$
  - **C2 :** si l'événement $emission_{i_k}(m)$ n'est pas capté dans el_i alors l'événement $reception_{j_r}(m)$ n'est pas capté dans el_j

Introduction Définition de l'état global Ainsi :



- $S_1 = \{e_i, e_j, e_{k_1}\}$ forme un ensemble d'états locaux cohérents (correspond à une "coupe cohérente")
- $S_2 = \{e_i, e_j, e_{k_2}\}$ ne forme pas un état global cohérent (C1 : violée pour M1, C2 violée pour M2)

Introduction Définition de l'état global

- $S_3 = \{\{e_i, e_j, e_{k_3}\}, \{c_{k,j} = \{M1\}\}\}$ forme un état global cohérent.

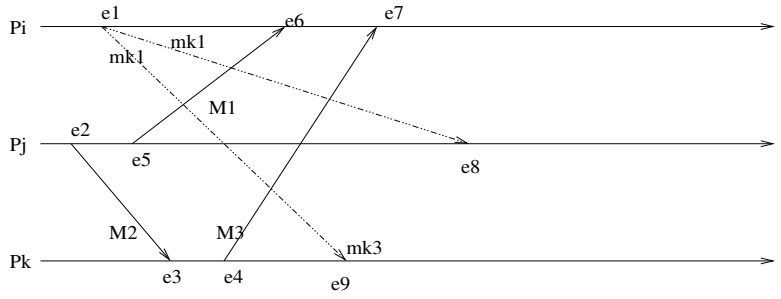
8.2 Solution pour des canaux FIFO

Rappel
Dans un canal fifo C_{ij} :
$m_1 < m_2 \Leftrightarrow emission(m_1) < emission(m_2) \Leftrightarrow reception(m_1) < reception(m_2)$

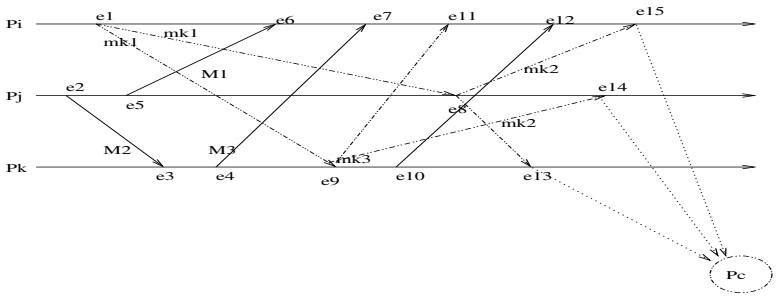
Soit el_i l'état de P_i à un instant I donné : seuls les messages émis avant cet instant I sont captés par el_i. Pour assurer la cohérence entre el_i et un el_j de P_j seules les réceptions des messages émis avant I doivent être captées par el_j (évident).
Pour mettre en oeuvre cette contrainte, P_i émet, en I un message de contrôle noté Mk sur le canal C_{ij}.
Le processus P_j doit alors enregistrer son état à la réception de ce message.
La "démonstration" qu'alors C1 et C2 sont vérifiées est évidente.
L'algorithme de Chandy et Lamport, 1985, est alors :
(Il peut exister un processus Pc chargé de récupérer l'état global du système : **facultatif**) Chaque processus P_i

- dispose d'une variable el_i qui lui permet de mémoriser son état local
- dispose de deux tableaux :
  - $R_i[1..N]$ qui représente C_{ji} (ensemble des messages reçus après la mémorisation de l'état local et avant la réception de Mk de P_j), initialisé à $\emptyset$;
  - $EL_i[1..N]$: qui permet du dire si P_i a reçu Mk de P_j , initialisé à Faux
- Un processus P_i qui reçoit un message m (autre que Mk) de P_j le stocke dans $R_i[j]$ si $EtatMemorise_i$ = Vrai et $EL_i[j] = k$
- Un processus P_i qui décide de mémoriser son état local exécute :
  - Prise_en_compte() ;
    - el_i = état local du processus P_i ; //P_i sauvegarde son état
    - $EtatMemorise_i$ =Vrai ;
    - Pour tout $j \neq i\{R_i[j] = \emptyset$; $mettre\ Mk\ vers\ P_j\}$
  - Un processus P_i qui reçoit un Mk de P_j :
    - Si $EtatMemorise_i$ = Faux, il exécute Prise_en_compte()
    - Dans tous les cas, il marque $EL_i[j] = Vrai$.
    - Si ($\forall j \neq i, EL_i[j] = Vrai$), il envoie l'état $\{el_i, R_i\}$ au processus Pc.

Exemple
On suppose qu'il n'y a pas d'événements locaux
Initialement, P_1 lance la captation d'un état global (cela pourrait être fait aussi par Pc : il suffit qu'il diffuse Mk)



Exemple
Ce qui nous donne



Solution pour des canaux FIFO Rappel **Exemple**

E	1	2	3	4	5	6	7	8	9
Etat	el_1								
R[2]/EL[2]						M1, Faux	M1, Faux		
R[3]/EL[3]							M3, Faux		
Act	D(mk1)					R(M1)	R(M3)		
Etat	∅							el_2	
R[1]/EL[1]								∅, vrai	
R[3]/EL[3]								∅, Faux	
Act		E(M2)			E(M1)			D(mk2)	
Etat	∅								el_3
R[1]/EL[1]									∅, vrai
R[2]/EL[2]									∅, Faux
Act			R(M2)	E(M3)					D(mk3)

Solution pour des canaux FIFO Rappel **Exemple**

E	10	11		13	14	15
Etat						
R[2]/EL[2]		M1, Faux	M1, Faux			M1, Vrai
R[3]/EL[3]		M3, Vrai	M3, Vrai			M3, Vrai
Act			R(M4)			E(el1,{∅,{M1},{M3}})-->Pc
Etat						
R[1]/EL[1]						
R[3]/EL[3]						
Act					E(el2,{∅ ∅ ∅})->Pc	
Etat						
R[1]/EL[1]				∅, Vrai		
R[2]/EL[2]				∅, Vrai		
Act	E(M4)			E(el3,{∅∅∅})->Pc		

Exemple

- Pourquoi $E(el_1\{\emptyset, \{M1\}, \{M3\}\}\} \rightarrow Pc$?
  - à cause de C1 : les émissions M1 et M3 sont captées par el_2 et el_3, comme les réceptions ne sont pas captées par el_1, elles doivent apparaître dans les canaux
  - à cause de C2 : l'émission de M4 n'est pas captée dans el_3, la réception ne doit pas être captée dans el_1, et M4 ne doit pas apparaître dans le canal.
- Pc peut maintenant mémoriser l'état global par : $S = \{\{el_1, el_2, el_3\},$

C_{ij}	j=1	2	3
i=1	$\varnothing$	$\varnothing$	$\varnothing$
2	M1	$\varnothing$	$\varnothing$
3	M3	$\varnothing$	$\varnothing$

$\}$

Solution pour des canaux FIFO Rappel **Exemple**

Revient donc à dire que avec l'état el_1 , P_1 n'a pas pris en compte M1 et M3.

- La remise à Faux de toutes les variables $EtatMemorise_i$ pourra alors être faite par Pc

8.3 Solutions pour des canaux non FIFO

8.3.1 Solutions sans messages de contrôle

8.3.1.1 Méthode cumulative (mais rapide) de Lai et Yang

Basé sur la notion d'état des processus : état avant enregistrement de l'état local / état après l'enregistrement

On associe une "couleur" à chacun de ces états : **vert** pour l'état AVANT , **rouge** pour l'état APRÈS.

D'où l'algorithme de **Lai et Yang**

Chaque processus obéit aux règles suivantes :

- **R1** : Un processus qui n'a pas encore enregistré son état est vert et tous les messages qu'il émet sont verts.
- **R2** : Lorsqu'un processus enregistre son état local, il devient rouge et tous les messages qu'il émet ensuite sont rouges.
- **R3** : Un processus peut à tout moment enregistrer son état local et doit l'enregistrer (si ce n'est pas déjà fait) lorsqu'il reçoit un message rouge, sans le prendre en compte.

D'où l'algorithme de **Lai et Yang**

- **R4** : Les messages verts émis et reçus sont stockés par le processus (aspect cumulatif).
- **R5** : Chaque processus qui devient rouge transmet au processus collecteur Pc, son état local et l'ensemble des messages verts qu'il a stockés

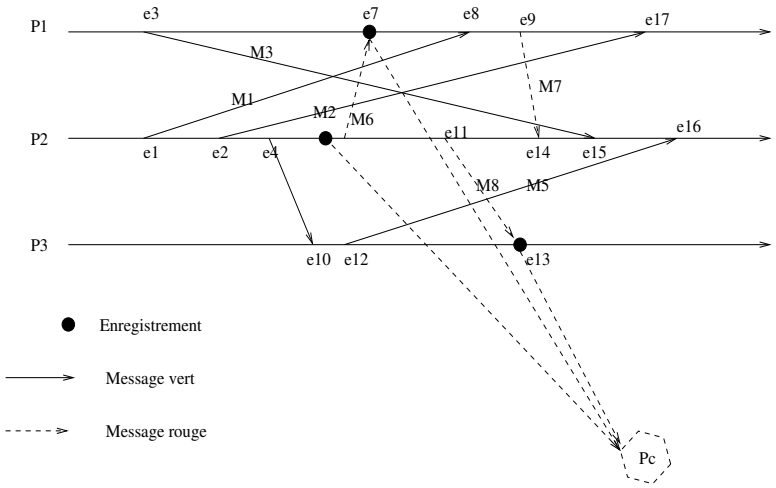
L'état global est calculé par Pc lorsqu'il a reçu les informations de tous les processus.

Remarque : l'état ec_{ij} d'un canal est la différence entre l'ensemble des messages verts émis par P_i vers P_j (noté msg_$emis_i[j]$) et l'ensemble des messages verts reçus par P_j depuis P_i (noté msg_$recu_j[i]$). Montrons que C1 et C2 sont bien vérifiées :

- si l'émission de m est captée dans el_i , alors m est coloré en vert (R1) et $m \in msg\_emis_i[j]$ (R4) :
  - si sa réception est captée dans el_j (c-à-d qu'il était présent au moment de l'enregistrement) alors $m \in msg\_recu_j[i]$ (R4) (au msg_recu_j qui est transmis à Pc) et donc $m \notin ec_{ij}$
  - si sa réception n'est pas captée dans el_j alors $m \notin msg_emis_i[j]$ (R4) et donc $m \in ec_{ij}$.

  Ceci assure C1
- si l'émission de m n'est pas captée dans el_i, alors m est coloré en rouge (R2). Lorsque m est reçu par P_j, ce dernier processus a déjà enregistré son état local, ou doit le faire sans prendre en compte m (R3). Donc la réception de m n'est pas captée dans el_js.

Ce qui assure C2.



Evénement	1	2	3	4	5	6	7	8	9	10
Couleur	V						R			
E1[2]			M3				$\emptyset$			
R1[2]								M1		
E1[3]										
R1[3]										
Action			E(M3,v)				R(M6,R), el1	R(M1,v)	E(M7,R)	
Couleur	V				R					
E2[1]	M1	M1, M2			$\emptyset$					
R2[1]										
E2[3]				M4	$\emptyset$					
R2[3]										
Action	E(M1,v)	E(M2,v)		E(M4,v)	el2	E(M6,R)				
Couleur	V									
E3[1]										
R3[1]										
E3[2]										
R3[2]										M4
Action										R(M4,v)

Evénement	11	12	13	14	15	16	17	18	19
Couleur									
E1[2]									
R1[2]							{M1,M2}		
E1[3]									
R1[3]									
Action							R(M2,v)		
Couleur									
E2[1]									
R2[1]					M3	M3			
E2[3]									
R2[3]						M5			
Action	E(M8,R)			R(M7,R)	R(M3,v)	R(M5,v)			
Couleur			R						
E3[1]									
R3[1]									
E3[2]		M5	$\emptyset$						
R3[2]		M4	$\emptyset$						
Action		E(M5,v)	R(M8,R),el3						

Pc a reçu, en plus des états locaux : $E_1[2] = \{M3\}$, $E_2[1] = \{M1, M2\}$, $E_2[3] = \{M4\}$, $E_3[2] = \{M5\}$ et $R_3[2] = \{M4\}$
Comme, $\forall(i,j), i \neq j, ec_{ij} = E_i[j] \backslash R_j[i]$, on obtient

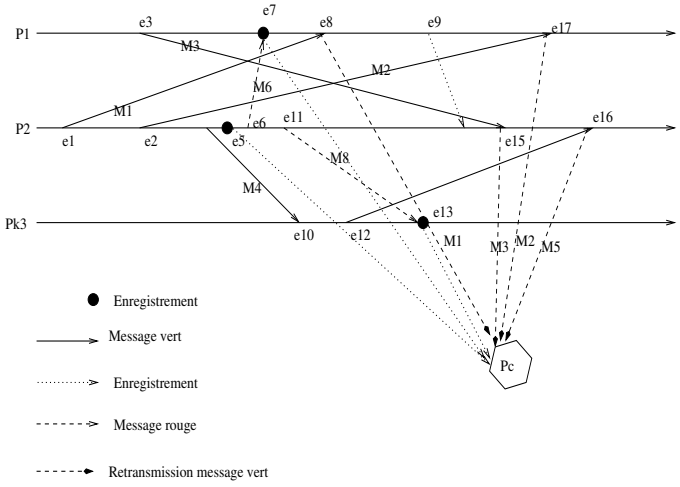
P_i	canal	1 $C_{y,1}$	2 $C_{y,2}$	3 $C_{y,3}$
1	$C_{1,x}$	x	{M3}	$\emptyset$
2	$C_{2,x}$	{M1, M2}	x	$\emptyset$
3	$C_{3,x}$	$\emptyset$	{M5}	x

Revient à dire que P_1 n'a pas pris en compte $\{M1, M2\}$, P_2 n'a pas pris en compte $\{M3, M5\}$, P_3 a tout pris en compte.
Problème : le volume des messages à stocker peut devenir important. D'où une méthode plus lente mais moins gourmande

8.3.1.2 Méthode non cumulative (mais lente) de Mattern

L'algorithme est basé sur la propriété suivante : Les messages en transit sur c_{ij} sont exactement les messages verts émis par le processus P_i (captés dans el_i) et reçus par le processus rouge P_j (non captés dans el_j).
Or si un processus P_j sait que les messages verts qu'il a reçu après son enregistrement sont les messages en transit sur le canal, il ne sait pas si il les a tous reçus car un message rouge peut très bien avoir doubler des messages verts. On va utiliser des compteurs de messages :
Chaque fois qu'un processus P_i enregistre son état local, il rajoute, aux informations qu'il envoie au processus centralisateur, la différence d_i entre le nombre de messages émis (donc verts) et le nombre de messages verts reçus AVANT l'enregistrement. Le nombre exact mt de message en transit sur l'ensemble des canaux est alors : $mt = \sum_i d_i$
D'où l'algorithme de **Mattern :**
Chaque processus obéit aux règles suivantes :

- **R1** : Un processus qui n'a pas encore enregistré son état est vert et tous les messages qu'il émet sont verts.
- **R2** : Lorsqu'un processus enregistre son état local, il devient rouge et tous les messages qu'il émet ensuite sont rouges.
- **R3** : Un processus peut à tout moment enregistrer son état local et doit l'enregistrer (si ce n'est pas déjà fait) lorsqu'il reçoit un message rouge, sans le prendre en compte.
- **R'4** : Chaque processus qui devient rouge transmet au processus collecteur Pc, son état local et son d_i.
- **R'5** : Un processus rouge transmet à Pc tous les messages verts qu'il reçoit
- Le processus Pc, lorsqu'il reçoit :
  - un enregistrement (el_i, d_i) : il exécute N = N ? 1 (où N est initialisé au nombre de processus) et $mt = mt + d_i$
  - un message vert (M, (site emetteur P_i, site recepteur P_j)) (en provenance donc de P_j) : il rajoute M à c_{ij} et décrémente mt
- Lors que N = 0 et mt = 0 , Pc dispose de tous les état.s locaux et de tous les messages en transit : il a donc l'état global.



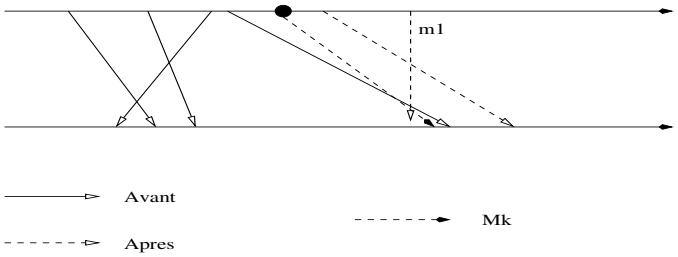
8.3.1.3 Remarque importante

Dans ces deux algorithmes, contrairement à l'algorithme de Chandry, le processus Pc est indispensable : il est le seul capable de calculer l'état global (dans l'algorithme de Chandry, chaque processus connaissait son état local et l'état de chacun de ses canaux : Pc ne servait qu'a avoir un vision "unique")

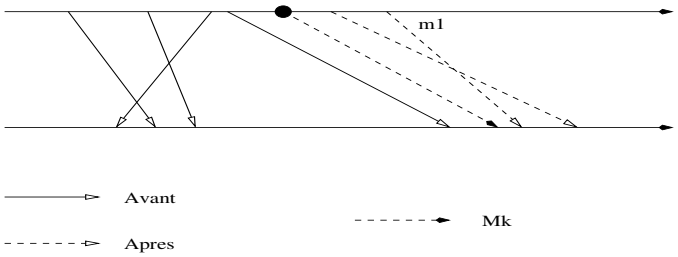
8.3.2 Solutions utilisant des messages de contrôle

On peut adapter l'algorithme de Chandy et Lamport si on dispose de marqueur particulier. En effet une première adaptation de cet algorithme consiste à introduire un type de messages chargés de séparer l'ensemble des messages émis AVANT enregistrement de el_i et de l'ensemble des messages

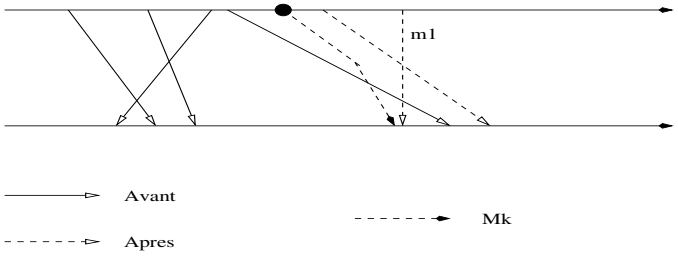
émis APRÈS. En effet, sans contrainte sur les messages on peut obtenir avec Chandy et Lamport si le réseau n'est plus FIFO.



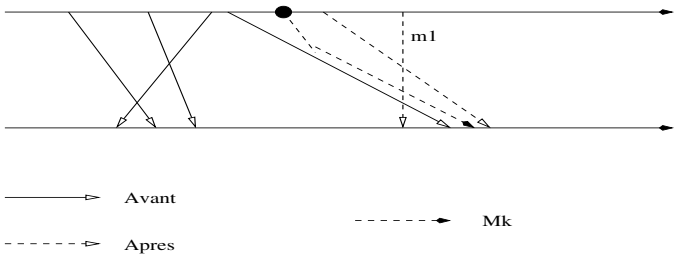
Si on introduit des messages Marqueur qui assure que tout message émis AVANT arrive AVANT et tout message émis APRÈS arrivera APRÈS, on obtient :



Or on voit que l'ordre de réception, sur cet exemple, est différent, ce qui interdit d'utiliser cette méthode car une des caractéristiques (contrainte) d'un algo de détermination d'un état global est qu'il ne doit pas perturber le phénomène observé (cf vol des oiseaux).
On peut par contre introduire des marqueurs du type :
ct_futur : (dit contraint par le futur) qui garantissent simplement que tout message émis APRÈS arrivera
APRÈS



la réception de m1 peut éventuellement être retardée : mais l'ordre des messages reçus APRÈS le marqueur est conservé.
ct_passe : (dit contraint par le passé) qui garantissent simplement que tout message émis AVANT arrivera
AVANT



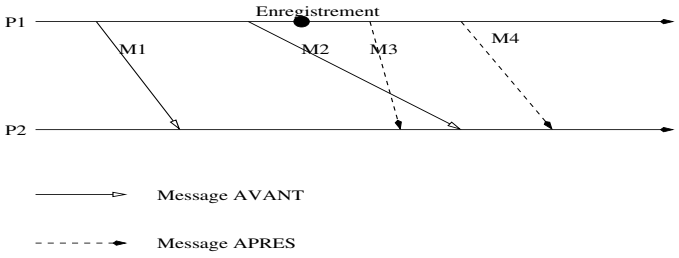
et alors on peut utiliser l'**algorithme d'Ahuja :**

Algorithme d'Ahuja :

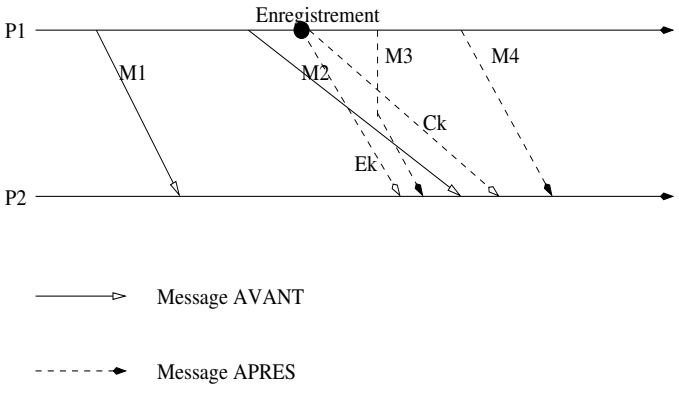
Chaque P_i enregistre son état local el_i et l'état de ses canaux en respectant les règles suivantes :

- **R1** : P_i lorsqu'il enregistre son état local, envoie sur tous ses canaux de sortie, un message Ek du type **ct_futur** avant d'envoyer tout autre message
- **R2** : à la réception d'un message Ek, le processus P_i enregistre son état selon la règle **R1**, s'il ne la pas déjà fait.
- **R3** : P_i lorsqu'il enregistre son état local envoie sur CHACUN de ses canaux de sortie, après le message Ek (du type **ct_futur**) et avant d'envoyer tout autre message, un message Ck du type **ct_passé** avec le même numéro que le dernier message émis sur ce canal : les messages sont numérotés par canal (le message estampillé t est le t-ieme message envoyé sur ce canal).
- **R4** : à la réception d'un message Ck numéroté t , sur un canal C_{ij} , P_j enregistre ec_{ij} comme étant l'ensemble des messages de numéro inférieur ou égal à t reçus après l'enregistrement de l'état local el_j.

Ainsi sur l'exemple suivant :



On obtient :



- **R1** et **R2** assure C2 car elles permettent de capter des états locaux cohérents :
  En effet, si un message m, traversant C_{ij}, est tel que $emission_i$(m) n'est pas captée dans el_i (exemple M3 et M4) , alors m est émis APRÈS le message Ek sur C_{ij} ; m est donc reçu APRÈS ce message (car celui-ci est **ct_futur**) et donc P_j aura déjà capté son état (dès la réception de Ek) et donc $reception_j$(m) ne sera pas captée dans el_j.
- **R3** et **R4** assure C1 car elles permettent de bien saisir l'état des canaux : En effet, soit un message m, traversant C_{ij}, émis AVANT l'enregistrement de el_i ($emission_i$(m) est donc captée dans celui-ci) (exemple M1 et M2) : son estampille t_m est nécessairement $\leq$ t de Ck.

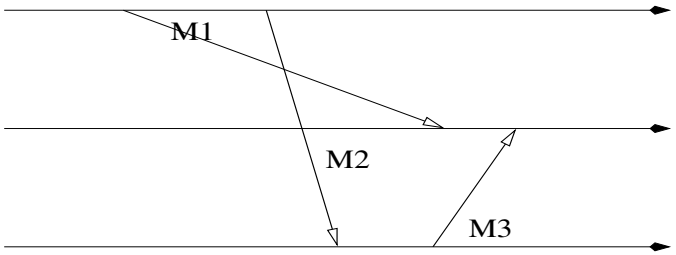
Deux cas :

- soit m arrive avant Ek : pas de problème, $reception_j$(m) sera captée dans el_j (exemple M1)
- soit m arrive après Ek : alors il arrive nécessairement AVANT Ck car celui-ci ne peut être dépassé (ct_passe) (exemple M2) Donc, lorsque P_j enregistrera l'état du canal, il aura nécessairement reçu m comme $t_m \leq t$ il sera mis dans l'état du canal. Réciproque, si un message mm (émis APRÈS Ck arrive AVANT Ck (exemple M3) son estampille *tt* sera nécessairement supérieure à t et *mm* ne sera pas captée dans ec_{ij}.

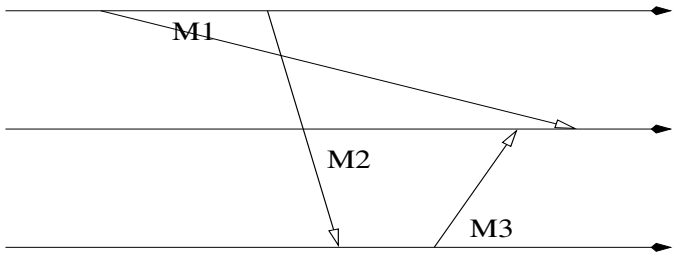
donc C2 est vérifiée. L'avantage de cet algorithme par rapport à Lai ou à Mattern, c'est qu'il n'y a pas besoin de site centralisateur.

8.4 Solutions fondées sur l'ordre causal

Rappel : dans un système, on dira que l'ensemble des messages respecte l'ordre causal si pour la relation "précède" noté $\rightarrow$:
$\forall\ P_i, P_j, P_k, \forall m_1$ emis sur $C_{ij}, \forall m_2$ emis sur $C_{kj} : emission_i(m_1) \rightarrow emission_k(m_2) \Rightarrow reception_j(m_1) \rightarrow reception_j(m_2)$
Exemple :



On voit que, par exemple que $emission_1(m_1) \rightarrow emission_1(m_2)$ et $emission_1(m_2) \rightarrow emission_3(m_3)$ d'où $emission_1(m_1) \rightarrow emission_3(m_3)$ d'où il faut $reception_2(m_1) \rightarrow reception_2(m_3)$ ce qui est bien le cas.
Attention : l'ordre causal définit un ordre sur la réception des messages strictement plus fort que celui imposé par des canaux FIFO. De fait, si la propriété de causalité est respectée alors chaque canal a un comportement FIFO. Alors que le fait que tous les canaux soit FIFO ne garantit pas l'ordre causal.
Exemple :



De ce fait, on peut d'attendre à des algorithmes plus simples.

8.4.1 Solution centralisée de Acharya et Badrinah

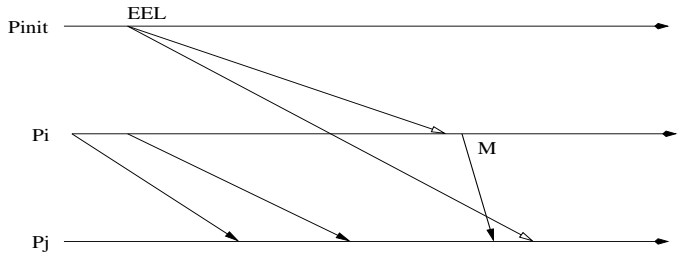
Calcul des états locaux :

- **R1 :** L'initiateur P_{init} diffuse (y compris a lui même) un message de contrôle enreg_etat_local
- **R2 :** Lorsqu'un processus P_i reçoit ce message de contrôle, il enregistre son état local el_i

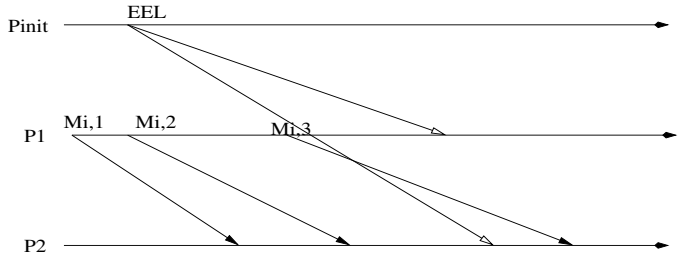
Démonstration de C2

Considérons un message m envoyé sur le canal c_{ij} tel que $emission_i$(m) n'est pas capté dans el_i mais $reception_j$(m) captée dans el_j, ce qui serait une violation de C2.

Par exemple : Calcul des états locaux :



Le message m a donc été émis par P_i après que celui-ci ait reçu le dernier message de contrôle $enreg\_etat\_local$, donc : $reception_i(enreg\_etat\_local) \rightarrow emission_i(m)$ d'où $emission_{init}(enreg\_etat\_local) \rightarrow reception_i(enreg\_etat\_local) \rightarrow emission_i(m)$ d'où $emission_{init}(enreg\_etat\_local) \rightarrow emission_i(m)$.
Si sa réception était captée dans el_j on aurait $reception_j(m) \rightarrow reception_j(enreg\_etat\_local)$ et l'hypothèse d'ordre causal serait violée.
Le cas suivant est toujours possible :



Pour assurer la captation de l'état des canaux, chaque P_i gère deux tableaux :

- $S_i[1..N]$ tel que $S_i[j] = n_{ij} \Leftrightarrow P_i$ a envoyé n_{ij} messages à P_j

– $R_i[1..N]$ tel que $R_i[j] = n_{ji} \Leftrightarrow P_i$ a reçu n_{ji} messages de P_j

Calcul de l'état des canaux :
Dans l'exemple $(S_1 = \{3\}, R_1 = \{\})$ et $(S_2 = \{\}, R_2 = \{2\})$
R2' : Lorsqu'un P_i reçoit le message enreg_etat_local, il enregistre son état local el_i et les deux tableaux, puis il envoie à P_{init} le message $etat\_local(el_i, S_i, R_i)$
Soit m le t^{ieme} message émis par P_i vers P_j , c'est donc aussi let^{ieme} messages qui sera reçu par P_j depuis P_i (ordre causal) :
Supposons que $R_j[i] < t \leq S_i[j]$
$\Leftrightarrow reception_j(m)$ non captée dans el_j (puisque m n'avait pas encore été reçu par P_j) mais par contre $emission_i(m)$ est captée dans el_i
$\Leftrightarrow$ m en transit relativement aux états locaux el_i et el_j d'où $m \in c_{ij} \Leftrightarrow$ m doit donc être mis dans l'état du canal.
D'où, pour garantir C1 :
R3 : lorsque l'initiateur P_{init} a reçu tous les messages $etat\_local(el_i, S_i, R_i)$, il possède tous les états locaux et calcule l'état des canaux par :

– $\forall\, j,\ c_{init,j} = \emptyset$
– $\forall\ i \neq init, \forall\ j, c_{ij} = \{M_{i,R_j[i]+1 \cdots M_i,S_i[j]}\}$ ce sont tous messages que P_i a envoyés à P_j et que P_j n'a pas encore reçus

(On suppose que P_{init} a accès a tous les messages : fréquent dans des algorithmes chargés de garantir la causalité)

8.4.2 Solution répartie de Alagar et Venkatesan

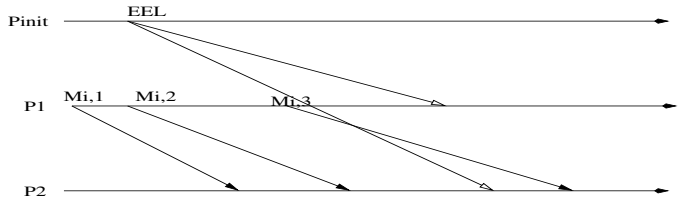
Chaque processus va devoir pouvoir calculer son état local mais aussi l'état de ses canaux entrants.
On utilise un technique de coloration non pas à l'émission, mais à la réception des messages : lorsque P_j reçoit un message m, celui-ci est coloré en **rouge** si et seulement si : $emision_{init}(enreg\_etat\_local) \rightarrow emission_i(m)$ (les structures de données maintenues par les protocoles de maintient de l'ordre causal permettent à P_j de faire ce test)

– **R1 :** L'initiateur P_{init} diffuse (y compris a lui même) un message de contrôle enreg_etat_local
– **R2" :** Lorsque P_i reçoit le message enreg_etat_local, il enregistre son état local el_i, initialise l'état de ses canaux entrants à $\emptyset$ et répond à P_i un message FAIT.

Solutions fondées sur l'ordre causal Solution répartie de Alagar et Venkatesan

– **R3 :** Lorsque, après avoir enregistré son état local, P_j reçoit un message sur c_{ij} il teste si ce message est à colorer en **rouge**. S'il ne l'est pas, il le met dans l'ensemble ec_{ij}
– **R4 :** Lorsque P_{init} a reçu un FAIT de tous les processus, il diffuse TERMINE
– **R5 :** Lorsque P_i reçoit TERMINE il enregistre l'état de ses canaux.

Exemple :



Solutions fondées sur l'ordre causal Solution répartie de Alagar et Venkatesan **Démonstration**
Montrons que lorsque que P_j reçoit TERMINE, il a nécessairement reçu de chaque $P_{i\neq j}$ tous les messages M enregistrés dans les états locaux des $P_{i\neq j}$
Soit m un tel message sur c_{ij} (donc en transit entre P_i et P_j avec $emission_i(m)$ captée dans el_i). On a :

- $emission_i(m) \rightarrow reception_i(enreg\_etat\_local)$ car m est capté dans el_i
- $reception_i(enreg\_etat\_local) \rightarrow emission_i(FAIT)$ (règle **R"**)
- $emission_i(FAIT) \rightarrow reception_{init}(FAIT)$ (par définition)
- $reception_{init}(FAIT) \rightarrow emission_{init}(TERMINE)$ (**R4**)

d'où $emission_i(m) \rightarrow emission_{init}(TERMINE)$ (transitivité de $\rightarrow$) Solutions fondées sur l'ordre causal Solution répartie de Alagar et Venkatesan **Démonstration**
d'où, d'après l'ordre causal : $reception_j(m) \rightarrow reception_j(TERMINE)$: il a bien reçu tous les messages en transit lorsqu'il enregistre les états de ses canaux.
Pour démontrer C1 il faut encore montrer que les messages reçus après $reception_j(enreg\_etat\_local)$ qui sont donc tel que $reception_j(m)$ est non captée dans el_j seront bien mis dans l'état du canal c_{ij} . Soit m tel que $reception_j(m)$ est non captée dans $el_j \Leftrightarrow reception_j(enreg\_etat\_local) \rightarrow receptionj(m)$
Or m n'est pas à colorer en **rouge**. En effet, $emission_i(m)$ est captée dans el_i, donc $emission_{init}(enreg\_etat\_local)$ not $\rightarrow emission_i(m)$
$\Rightarrow$ m n'est donc pas coloré en **rouge**, d'où P_j enregistre le message. Solutions fondées sur l'ordre causal Solution répartie de Alagar et Venkatesan **Démonstration**
$\Rightarrow$ le message m est bien enregistré dans ec_{ij} (**R3** et **R5**)
Donc, lorsque P_i reçoit TERMINE, il a enregistré tous les messages reçus soit dans son état eli soit dans les canaux de communication. La condition C1 est donc bien respectée.
Réciproquement, il est évident que si m est enregistré dans ec_{ij} on a nécessairement : $reception_j(enreg\_etat\_local) \rightarrow reception_j(m) \rightarrow reception_j(TERMINE)$ et m n'est pas coloré en rouge.
D'où si m est enregistré dans ec_{ij} alors nécessairement :

- $reception_j(m)$ n'est pas captée dans el_j
- $emission_i(m)$ est captée dans el_i

Ce qui permet d'assurer C2.

CHAPITRE 9

Terminaison

9.1 Introduction

Soit un ensemble de n tâches interdépendantes au sein d'une application. Ces n tâches ou processus sont répartis sur une graphe de communication. Chaque processus en cours d'exécution, exécute un algorithme séquentiel et échange des messages avec les autres processus via le graphe de communication.
Le problème est le suivant : l'arrêt de tous les processus correspond-il à l'arrêt définitif de l'application ou n'est-ce qu'un état transitoire ? (c'est-à-dire un message peut être en transit et provoquer le redémarrage d'un, puis de tous les processus) ?
Définitions

- Un processus est soit actif soit inactif : il est actif s'il exécute du code, il est inactif s'il n'a rien à faire ;
- Seuls les processus actifs peuvent envoyer des messages ;
- Un processus ne peut passer d'un état inactif à un état actif que sur réception d'un message : à tout message correspond du code a exécuter. Par contre, il peut passer de l'état actif à inactif à tout moment (il a fini son code) ;
- Tous les processus sont actifs au lancement de l'application.

Un deuxième problème auquel nous ne nous intéresserons pas, c'est de savoir si les processus ont "bien terminés" c'est-à-dire de savoir si le résultat est satisfaisant localement (chaque processus a bien fait le calcul prévu : cette estimation est en général faite par le processus lui-même) et globalement (le résultat global est bien le résultat recherché).
Hypothèses

- Tout processus qui ne reçoit plus de message se termine dans un temps fini.
- Le réseau est FIFO

9.2 Cas d'un graphe en anneau uni-directionnel

Cas d'un graphe en anneau uni-directionnel

9.2.1 Principe de l'algorithme (Dijstra et Van Gasten - 1983)

On utilise un jeton à deux états Termine ou État_Transitoire. Le processus P_0 qui veut déterminer si l'application est terminée, émet le jeton dans l'état Terminé. Si celui-ci lui revient dans le même état, l'application est terminée. Sinon (donc le jeton revient dans l'état État_Transitoire), il faut relancer la détection de la terminaison.

9.2.2 L'algorithme

Chaque processus Pi maintient les deux variables locales suivantes :

- $couleur_i$ = Blanc ou Noir initialisé à Blanc
- $etat_i$ = Actif ou Inactif initialisé à Actif

Initialement, le jeton est à l'état Terminé en P_0 qui lance la détection dès que son état passe à Inactif en transmettant le jeton à P_1.

État du processus et transmission du jeton

- Lorsque P_i reçoit un message, il passe à l'état Actif
- Un processus Actif ne transmet pas le jeton, il le transmettra dès qu'il deviendra inactif.
- Lorsqu'il a fini de calculer, il passe à l'état Inactif.
- Lorsqu'il envoie un message vers un processus P_j tel que j < i, il passe à l'état Noir.

Pourquoi ?

Parce qu'il se peut que le processus P_j soit ré-activé par ce message APRÈS que P_j ait transmis le jeton à l'état État_Transitoire. Donc, si P_i transmet un jeton Termine et que tous les processus P_k>i sont inactifs, alors le jeton arrivera en P_0 dans l'état Termine. Donc si P_i se contentait de transmettre le jeton en signalant qu'il est inactif, rien ne préviendrait P_0 du réveil de P_j. D'où lorsque P_i recevra le jeton, dès qu'il sera inactif, il transmettra le jeton dans l'état Etat_Transitoire (en fait, cela revient à ce que P_i demande un nouveau tour). Puis il pourra repasser à Blanc.

D'où, un processus Inactif transmet le jeton d'état :

- identique à l'état du jeton reçu si il est dans l'état Blanc
- Etat_Transitoire si il est dans l'état Noir, puis passe à l'état Blanc.

Détection de la terminaison

Si le jeton revient en P_0 avec l'état

- Terminé alors FIN : Terminaison Détectée.
- Etat_Transitoire ou si P_0 est actif, alors dès que P_0 redevient inactif, il remet le jeton à l'état Terminé et devient Blanc. Puis il relance la détection en transmettant le jeton à P_1.

9.3 Cas d'un arbre couvrant

Rappel Lorsque l'on est en phase d'initialisation des calculs, seuls les pères peuvent transmettre des demandes vers les fils directs.

Pour un sommet Pi de l'arborescence, posons

$$etat_{i,local}(P_i) = \begin{cases} 1\ si\ P_i\ a\ fini\ son\ code \\ 0\ sinon \end{cases} \tag{9.1}$$

et D_{P_i} ensemble des descendants de P_i à qui P_i a transmis une demande.
Soit $etat_i(P_j)$ la vision qu'a le processus P_i de l'état du processus P_j alors

$$etat_i(P_i) = \begin{cases} etat_{i,local}(P_i)\ si\ D_i = \emptyset \\ etat_{i,local}(P_i) \times \prod_{P_j \in D_i} etat_i(P_j)\ sinon \end{cases} \tag{9.2}$$

Lorsque l'on est en phase de calcul : tous les calculs ont été initiés, il suffit alors que chaque processus transmette la vision de son état dès que celui-ci est égal à 1, pour qu'un noeud puisse calculer son état. D'où :

En phase d'initialisation

- $etat_0(P_0)$ = 0
- chaque fois qu'un message part d'un processus P_i vers un fils P_j :
  - P remet $etat_i(P_i)$ à 0 [1], $D_i = D_i \cap \{j\}$ puis P_i "attend" l'état de P_j pour recalculer son propre état ;
  - P_j remet $etat_j, local(P_j)$ et $etat_j(P_j)$ à 0, passe à l'état Actif et exécute son code.

En phase de détection

- dès que $etat_{i\neq 0}(P_{i\neq 0}) = 1$, P_i transmet 1 à son père
- dès que $etat_0(P_0) = 1$, l'application est terminée.

9.4 Cas général

Algorithme de Misra (1983)

On suppose que le le graphe de communication est fortement connexe (c'est- à-dire que tout processus Pi peut envoyer un message à tout $P_{0<j\neq i\leq n}$: pas nécessairement directement mais en passant éventuellement en passant d'autres processus). Dans ce cas, la théorie des graphes nous assure qu'il existe un circuit [2] C qui comprend chaque arc (au moins une fois) du graphe du réseau.
On va utiliser un jeton pour trouver le nombre nb de processus visités suivant le circuit C qui soient restés inactifs entre deux passages de ce jeton.
Il est donc évident que l'application sera terminée lorsque nb = taille(C), où taille(C) est la taille du circuit en nombre de visites de processus (peut être supérieur à n car on peut passer plusieurs fois par un même processus). Posons :

- succ(P_i) successeur de P_i donné par le circuit [3] C
- Pour chaque processus P_i les quatre variables locales suivantes :
  - $couleur_i$ = Blanc ou Noir initialisé à Blanc

1. $etat_i(P_i)$ a été remis à 0 à la réception P_i du message demandant un calcul
2. ce circuit n'est pas toujours évident à trouver avec un algorithme réparti
3. que l'on suppose construit

- $etat_i$ = Actif ou Inactif initialisé à Actif
- jeton_$present_i$ = Vrai ou Faux initialisé à Faux sauf pour P_k, k choisi aléatoirement
- nb_i = entier initialisé à 0

Etat d'un processus

Lorsque P_i reçoit un message, il passe à l'état actif par : $etat_i$ = Actif

Chaque fois qu'il envoie un message, il mémorise ce fait par : $couleur_i$ = Noir. Ainsi, indirectement, il mémorise qu'il est possible qu'il ait réveillé un processus.

Lorsqu'il a fini de calculer, il passe à l'état inactif par : $etat_i$ = Inactif

Lorsqu'il reçoit le jeton, il effectue : nb_i = nb et jeton_$present_i$ = Vrai puis

s'il est inactif :

- soit il détecte la terminaison : FIN
- soit il doit retransmettre le jeton

s'il est Actif : il ne le transmettra que lorsqu'il redeviendra inactif.

Transmission du jeton

Si P_i est Inactif (ou si P_i passe à Inactif) avec jeton_$present_i$ = Vrai alors /* Pi transmet le jeton */ :

si $couleur_i$= Noir [4]

alors /*on recommence la détection */

nb_i = 1

sinon /* on incrémente le nombre de processus inactifs déjà visité */

$nb_i = nb_i + 1$

$couleur_i$ = Blanc jeton_$present_i$ = Faux

envoyer(jeton, nb = nb_i) à succ(P_i)

Détection de la terminaison

si nb = taille(C) et $couleur_i$ = Blanc alors Terminaison Détectée

4. il a donc envoyé un message pendant sa phase d'activité