ALGORITHMIQUE DU CONTROLE REPARTI

Télécom ParisTech

Module Algorithmique répartie de l'UE INF346

Irène CHARON

Mars 2009

Tables des matières

Chapitre 1 - Généralités

1.1. Introduction

L'algorithmique du contrôle d'un système réparti traite du contrôle de mécanismes de communication et de synchronisation de processus qui s'exécutent en parallèle sur un support réparti, c'est-à-dire lorsque les processus participants s'exécutent sur différents sites (cas par exemple d'un réseau local ou à grande distance) ; ces processus coopèrent à un même service. De plus, un algorithme de contrôle est dit distribué s'il n'utilise pas un même espace d'adressage dans lesquelles les différents processus puissent lire et écrire afin de permettre synchronisation et communication. Lorsque des processus parallèles sont contrôlés par des variables partagées et qu'ils disposent donc de mémoires communes, on dit que le contrôle est centralisé.
On supposera pour simplifier que l'ensemble des processus « participant à l'action » sont placés sur des sites distincts, un processus par site ; même si plusieurs processus sont effectués sur un même site, il n'y a pas d'inconvénient à les imaginer sur différents sites.
Le problème de base d'un système distribué est que chaque site ne connaît à un instant donné que son propre état, les informations qu'il obtient sur l'état des autres processus ne lui parviennent pas simultanément avec les changements d'état de ceux-ci, la réception d'une information venant d'un autre site l'informe sur un état antérieur mais pas sur l'état actuel : il n'y a pas de vision globale de l'état du système.

On peut distinguer deux types d'algorithmes répartis.
D'une part les algorithmes où les processus ont la possibilité de lire dans la mémoire des autres sites. Chaque processus tient à jour des variables (appelées *variables d'état*) qui rendent compte de leur état, et qui peuvent être consultées par les autres processus. Une variable qui peut être accédée à la fois en lecture et en écriture par un seul processus P et accédée en lecture par les autres processus s'appelle variable *propre* à P. Dans ce genre de méthode, on peut considérer que la communication se fait par demande d'informations d'un processus vers les autres processus.
D'autre part, les algorithmes basés sur l'échange de messages entre les différents sites. Dans ce type d'algorithme, on considère en général que c'est à chaque processus d'informer les autres processus de variations sur des paramètres qui concernent la collectivité, comme le désir d'accès à une ressource partagée, ou bien une information sur le résultat d'un calcul, ou bien la modification d'une donnée dupliquée sur les différents sites. Les échanges se font davantage par envoi d'informations que par demande d'informations. En grande majorité, les algorithmes distribués sont basés sur l'échange de messages.

Pour mettre en œuvre un algorithme basé sur les variables d'état, qui procède par lecture régulière d'informations sur l'état des autres processus, on peut en fait utiliser des messages : demande d'informations d'un processus P_1 à un processus P_2 puis réponse de P_2 vers P_1 ; mais cela entraîne des demandes d'informations inutiles vers des processus qui n'ont pas modifié les paramètres utiles à la synchronisation ; la mise en œuvre n'est en fait efficace que

sur une architecture centralisée. Mais on peut se demander pourquoi, lorsque l'architecture le permet, ne pas faire une mise en œuvre centralisée ; un avantage de n'utiliser que des variables locales ou propres est qu'on obtient souvent ainsi une meilleure tolérance aux pannes ; par ailleurs, les algorithmes basées sur l'utilisation des variables d'état sont souvent beaucoup plus claires que ceux où plusieurs processus peuvent modifier une même variable.

Dans le premier chapitre, nous donnons quelques généralités à propos des algorithmes basés sur l'échange de messages. Dans le deuxième chapitre, nous étudions le problème de l'exclusion mutuelle et nous donnons plusieurs algorithmes distribués de principes variés. Dans le troisième chapitre, nous étudions deux autres exemples de problèmes de contrôle : le problème de l'élection et la diffusion d'informations. Dans le dernier chapitre, nous examinons brièvement le problème de l'interblocage et de la détection de la terminaison.

Le premier chapitre décrit les différentes hypothèses qu'on peut faire sur le réseau, définit les qualités que l'on peut attendre du fonctionnement d'un réseau et donne différents types de méthode que l'on peut utiliser (estampillage, horloges vectorielles, calcul diffusant, utilisation d'un jeton).

1.2. Algorithmes basés sur l'échange de messages, généralités

1. Maillage

Une première caractéristique importante d'un système distribué est le maillage du réseau de communication, c'est-à-dire l'ensemble des liaisons directes entres les sites. Il s'agit en fait de maillage logique ; si certaines communications prévues par le maillage ne sont pas physiquement installées, il faut rajouter une couche logicielle qui effectue un routage pour envoyer un message d'un site à un autre site supposé connecté à lui dans les hypothèses de structure de l'algorithme ; il est évidemment nécessaire que le réseau physique soit connexe. Les principaux maillages utilisés sont :

- le maillage complet : tout processus peut communiquer avec tout processus.
- le maillage en anneau : le réseau est un cycle, tout processus possède deux voisins et il ne peut communiquer qu'avec ceux-ci.
- le maillage en étoile : un processus particulier peut communiquer avec tous les autres qui, réciproquement, ne communiquent qu'avec le processus particulier. Ce maillage présente l'inconvénient de particulariser un processeur et de rendre fâcheuse la panne du site central.
- le maillage en arbre : la panne de tout processus non terminal déconnecte le système logique.

Dans les trois derniers maillages, il est possible de prévoir une reconfiguration lorsque l'un des processeurs tombe en panne (si les connexions physiques le permettent).

2. Qualités des voies de communication

Des hypothèses doivent aussi être faites sur le comportement des voies de communication. Les caractéristiques les plus importantes à considérer sont :

- l'ordre de réception des messages d'un processus donné à un autre processus donné est-il le même que l'ordre d'émission ? Autrement dit, y a-t-il ou non déséquencement possible des messages ?

- est-il possible que certains messages se trouvent dupliqués au cours de leur transmission ?
- les messages peuvent-ils être altérés ?
- les messages peuvent-ils être perdus ?
- le délai d'acheminement des messages est-il borné, c'est-à-dire peut-on être certain d'une valeur maximum du délai entre émission et réception ?

Ces propriétés peuvent être assurées, si elles le sont, soit par la structure matérielle du réseau sur lequel est implanté l'algorithme, soit par une couche logicielle prévue à cet effet.

3. Qualités d'un algorithme de contrôle réparti

Voyons enfin les qualités que l'on peut souhaiter pour un algorithme de contrôle réparti. On peut souhaiter :

- que le nombre de messages échangés soit peu important.
- minimiser le temps d'exécution.
- que le trafic sur les voies de communication ne soit pas trop dense, qu'il soit bien réparti sur les lignes de communication et qu'il n'y ait pas de site trop sollicité par la réception ou l'envoi de messages.
- qu'il ne nécessite pas certaines hypothèses sur les voies de communication, par exemple qu'il admette le déséquencement des messages.
- qu'il ait une bonne résistance aux pannes, c'est-à-dire que le système contrôlé ne soit pas irrémédiablement interrompu si l'un des processeurs qui assurent le contrôle devient défectueux.

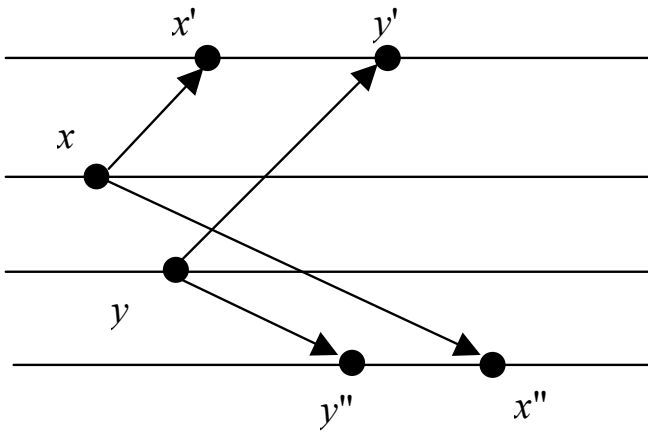
1.3. Ordonnancement des événements

1. Le problème

On appelle *événement* un changement local d'état sur un site ou bien l'envoi ou la réception de messages.

Les messages échangés dans un système réparti sont essentiellement destinés à la coordination entre les tâches, c'est-à-dire à la synchronisation. Le problème fondamental de cette synchronisation est que les transmissions ne sont pas immédiates ; les délais de transmission sont en général beaucoup plus longs que le temps séparant deux instructions d'un même processus, on peut évaluer très grossièrement qu'il y a un facteur de l'ordre de mille. Les conséquences sont :

- à un instant donné, on ne peut connaître, à partir d'un site donné, qu'une approximation de l'état d'un autre site
- deux événements observables quelconques d'un système peuvent être perçus dans un ordre différent d'un site à l'autre comme l'illustre le schéma ci-dessous :



Néanmoins, il faut souvent que les différents sites puissent s'accorder sur l'ordre « chronologique ». C'est entre autres le cas :

- lorsque l'on doit gérer une file d'attente virtuelle répartie, par exemple dans un algorithme d'exclusion mutuelle ; une telle file d'attente possède une représentation partielle ou totale sur chacun des sites concernés par l'algorithme : il faut que les ordres de ces représentations soient identiques.
- dans la mise en œuvre d'une mémoire répartie, où chaque site possède une copie d'une mémoire virtuelle ; il faut alors connaître la modification la plus récente.
- ...

Or, dans un système réparti, on ne peut pas utiliser une heure unique, car rien ne permet d'assurer que les horloges sont synchrones. Supposons qu'on décide d'attribuer à un événement l'heure indiquée par l'horloge du site sur lequel il se produit ; un message pourrait alors arriver « avant d'être parti », ce qui rend difficile la gestion cohérente de copies de variables.

2. Précédence causale

Définition
Soit E un ensemble et $\mathcal{R}$ une partie de $E \times E$ (c'est-à-dire une relation sur E). On appelle *fermeture transitive* de $\mathcal{R}$ l'intersection des parties $\mathcal{T}$ de $E \times E$ vérifiant :

- $\mathcal{R} \subset \mathcal{T}$
- si $(x, y) \in \mathcal{T}$ et $(y, z) \in \mathcal{T}$, alors $(x, z) \in \mathcal{T}$

L'ordonnancement des événements d'un système réparti repose sur la définition d'une *relation de précédence* définie comme suit : si a et b sont deux événements, on dit que a *précède directement* b si l'une des conditions suivantes est satisfaite :

- a et b arrivent sur le même site et a est antérieur à b sur ce site
- a est l'envoi d'un message depuis un site et b est la réception de ce même message sur un autre site.

La relation *précède* est la fermeture transitive de la relation ci-dessus ; on note $a \rightarrow b$ la relation « a précède b ». Cette relation est appelée relation de *précédence causale* ; en effet, un événement a ne peut avoir influé sur un événement b que si $a \rightarrow b$.
Deux événements a et b sont dits *concurrents* (ou *causalement indépendants*) si aucun des deux ne précède causalement l'autre.

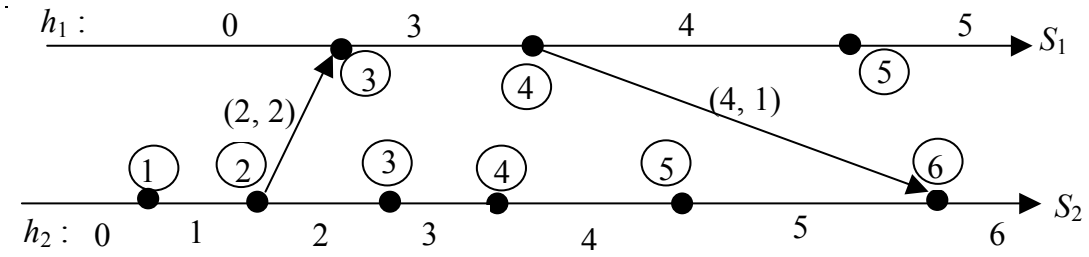
3. Estampillage, horloges logiques

Il s'agit d'une technique qui permet de définir un ordre partiel ou global entre les événements compatibles avec la relation de précédence causale. Pour cela, on effectue une datation des événements.

Chaque site S_i gère une variable h_i appelée *horloge logique*. h_i est une variable entière initialisée à 0, et qui ne fait que croître. Lorsqu'un événement a se produit sur un site S_i, la valeur de h_i est incrémentée de 1 ; la date de a, notée $h(a)$, est alors h_i. Par ailleurs :

- lors de l'émission d'un message par le site S_i, h_i est incrémentée de 1 ; un message m est alors *estampillé* par (h_i, i) ; il devient donc de la forme (m, h_i, i) ; h_i est la date d'émission du message m.
- lors de la réception par S_j d'un message (m, h_i, i), l'horloge h_j est actualisée à la valeur $(\max(h_i, h_j) + 1)$ et ce dernier nombre est considéré comme la date de réception du message.

Nous illustrons ci-dessous ce mécanisme ; les événements sont représentés par des points ; les dates des événements sont entourées.



Il peut arriver qu'on ne tienne compte que d'une partie seulement des événements, les autres événements n'étant alors pas datés. On peut remarquer que si $a \rightarrow b$, alors la date de a est inférieure à la date de b.

On peut définir un ordre sur les événements par : un événement a est *inférieur* à un événement b si et seulement si la date de a est strictement inférieure à la date de b ; cet ordre n'est que partiel car plusieurs événements peuvent être affectés de la même date. On peut étendre cet ordre en un ordre total strict, noté $<$, en décidant que, entre deux événements qui ont la même date (ce qui n'est pas possible pour deux événements distincts ayant eu lieu sur un même site), c'est celui qui a eu lieu sur le site de plus petit numéro qui est antérieur à l'autre ; autrement dit, si a et b sont deux événements ayant eu lieu sur les sites i et j :

$$(a < b) \Leftrightarrow date(a) < date(b) \text{ ou } (date(a) = date(b) \text{ et } i < j)$$

4. Horloges vectorielles

La relation $date(a) < date(b)$ n'est pas suffisante pour savoir s'il existe une relation de causalité entre a et b ; cette relation implique uniquement que b ne précède pas causalement a ; il est possible que a précède causalement b ou bien que a et b soient causalement indépendants. Il est quelquefois utile de pouvoir déterminer s'il y a ou non dépendance causale entre deux événements, par exemple pour la recherche des causes potentielles d'une erreur ou d'une panne.

Le mécanisme des *horloges vectorielles* a été introduit pour caractériser la dépendance causale.

Supposons qu'il y ait n sites. Sur un site, on appelle *passé* l'ensemble des événements (y compris les émissions et les réceptions de message) qui précèdent causalement le dernier événement qui a eu lieu sur le site concerné. Sur chaque site S_i, on définit une horloge

vectorielle V_i comme un vecteur d'entiers, de longueur n ; cette horloge est actualisée de telle sorte que :

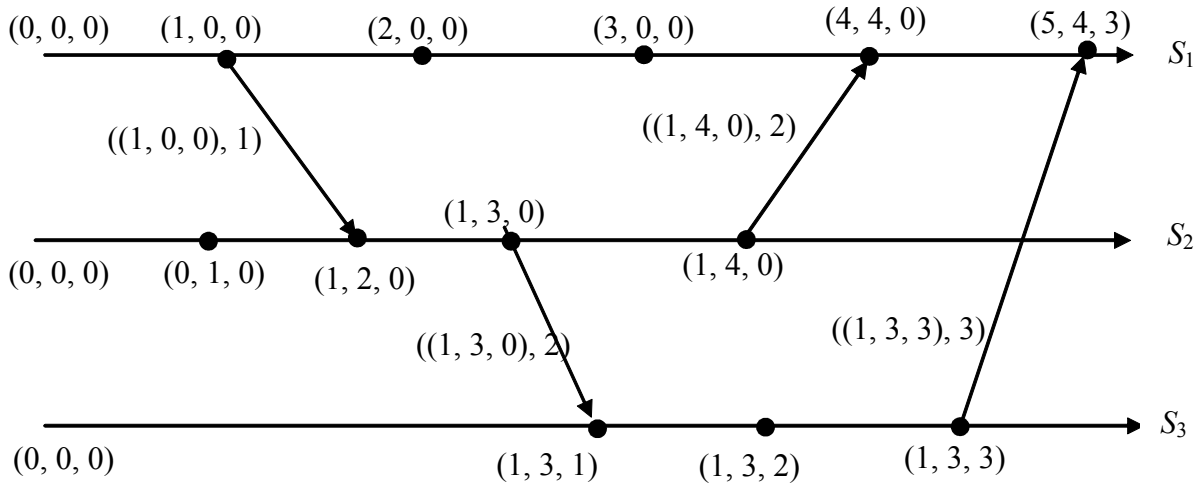
$V_i[j]$ = nombre d'événements du passé ayant eu lieu sur le site j.

On associe aussi à chaque événement a une *date* notée *date*(a) qui est égale à la valeur (vectorielle) de l'horloge vectorielle locale juste après cet événement.

Pour effectuer le calcul de V_i, on opère les opérations suivantes :

- au départ, sur chaque site, V_i est initialisé à (0, ..., 0).
- si un événement a se produit sur le site i, $V_i[i]$ est incrémenté de 1 ; on affecte le vecteur V_i à *date*(a) qui est une valeur d'horloge vectorielle ; on remarque que *date*(a)[i] comptabilise tous les événements ayant eu lieu sur S_i, y compris a.
- avant émission d'un message depuis le site i, $V_i[i]$ est incrémenté de 1 puis le message est estampillé par (V_i, i).
- si le site j reçoit un message du site i, estampillé par (V_i, i), alors $V_j[j]$ est incrémenté de 1 puis, pour $1 \leq k \leq n$, $k \neq j$, $V_j[k]$ devient max($V_i[k]$, $V_j[k]$).

Nous illustrons ci-dessous ce mécanisme, en mettant à côté de chaque événement sa date ; les valeurs des horloges vectorielles sont données par la date du dernier événement produit sur le site :



Si V et W sont deux valeurs d'horloges vectorielles, on dit que V < W si toute composante de V est inférieure ou égale à la composante correspondante de W et si l'une au moins des composantes de V est strictement inférieure à la composante correspondante de W.

On vérifie facilement que, si a et b sont deux événements :

$$a \rightarrow b \quad \Leftrightarrow \quad date(a) < date(b)$$

1.4. Différentes techniques utilisées

1. Le jeton circulant

Le maillage logique est un anneau ; le principe est de faire circuler un « privilège » unique sur cet anneau ; le privilège est reçu grâce à un message de type particulier venant du site

précédent de l'anneau ; ce message est appelé *jeton ;* le jeton peut transporter des valeurs de variables qui renseignent sur les autres sites au moment du dernier passage du jeton.

2. Le calcul diffusant

Ce type d'algorithme peut servir à diffuser ou collecter de l'information.
Le maillage logique est un arbre couvrant. Un processus peut déclencher l'algorithme diffusant, ce processus est appelé racine de l'arbre. Il émet pour cela un message vers tous ses voisins, et est considéré comme le père de ceux-ci. Ceux-ci à leur tour transmettent l'information à leurs voisins autres que leur père, et sont considérés comme le père de ces voisins, et ainsi de suite. Quand un processus atteint est une feuille de l'arbre, il envoie une réponse à son père ; quand un processus a reçu une réponse de tous ses voisins différents de leur père, il envoie une réponse à son père. L'algorithme s'arrête quand le processus racine a reçu une réponse de tous ses voisins.
Il se peut qu'un seul processus ait le privilège de lancer l'algorithme, ou bien que tout processus en ait la possibilité. Il se peut aussi que le maillage soit quelconque et que l'arbre soit construit au fur et à mesure ; l'algorithme est identique à la variante suivante près : un processus qui a déjà reçu le message lui donnant l'ordre de diffusion à tous ses voisins et reçoit à nouveau ce même ordre d'un autre processus se dispense de diffuser et répond simplement à son père qu'il fait déjà partie de l'arbre.
Plus de précisions peuvent être trouvées dans le paragraphe 4.2.

Chapitre 2 - Le problème de l'exclusion mutuelle

2.1. Introduction

Il s'agit ici d'étudier le contrôle de *n* processus entrant en compétition pour se partager une même ressource ; celle-ci peut être utilisée par un seul processus ; quand un processus l'utilise, on dit qu'il est en *section critique*. On suppose que l'utilisation de la ressource a toujours une durée finie. L'utilisation de la ressource par un processus suivra toujours le plan suivant :

- protocole d'acquisition de la ressource
- utilisation de la ressource en section critique
- protocole de libération de la ressource.

Un processus *P'* entamant son processus d'acquisition alors que la section critique est utilisée par un processus *P* devra attendre (rester dans la phase d'acquisition) au moins jusqu'à ce que *P* ait terminé son protocole de libération.

Voyons les embûches à éviter lorsqu'on met au point des protocoles d'exclusion mutuelle.

Imaginons ce premier protocole d'acquisition : si un processus découvre qu'il est en compétition avec d'autres pour l'acquisition, il laisse passer les autres : c'est l'exemple le plus simpliste qu'on puisse imaginer de ce qui conduit à une situation dite d'*interblocage* ; tous les processus entrant en compétition sont bloqués et la ressource reste éternellement non utilisée.

L'interblocage dans le cas de l'exclusion mutuelle peut s'énoncer ainsi :

interblocage =	il existe au moins un processus qui désire accéder à la section critique et aucun processus ne peut l'atteindre.

Considérons maintenant un système dans lequel opèrent trois processus P_1, P_2, P_3 qui utilisent une même ressource *R*. Si le protocole est tel que P_1 et P_2 puissent alterner l'utilisation de la section critique, ou bien même que P_1 puisse l'utiliser de façon répétitive, de telle sorte que P_3 n'accédera jamais à la section critique, on dit qu'il y a *famine*. Si un protocole rend impossible la famine, on dit aussi qu'il assure l'*équité*. On peut énoncer :

équité = non famine = tout processus demandant la section critique l'atteint en un temps fini.

La qualité d'un algorithme d'assurer l'équité s'appelle aussi la *vivacité.*

L'équité entraîne le non interblocage.

On s'intéresse aussi, lorsqu'il n'y a pas famine, aux différents ordres possibles de passage en section critique par les processus :

- si tous les processus accèdent à la section critique dans l'ordre où ils l'ont demandée, on dit que l'attente est *FIFO.*
- si tout processus demandeur peut se faire doubler pour l'accès en section critique au plus une fois par chaque autre processus, on dit que l'attente est *linéaire* ; un processus *P* se fait doubler par un processus Q quand, ayant débuté en premier son protocole d'acquisition, il passe après Q en section critique.

- si, pour un protocole donné, il existe une fonction $f(n)$ (n = nombre de sites) tel qu'un processus donné se fasse doubler par au plus f(n) processus, on dit que l'attente est *bornée par f(n)*.

Remarque :

Dans un contrôle distribué basé sur l'échange de messages, pour juger si une demande d'accès à la section critique est antérieure à une autre, on se réfère à des horloges logiques et non à l'unique échelle du temps (il est impossible à un contrôle distribué de connaître l'ordre réel d'une séquence d'événements ayant lieu sur différents sites car il n'y a pas d'horloge commune).

La qualité minimum requise par un protocole d'exclusion mutuelle est, en plus de réaliser l'exclusion, d'éviter l'interblocage.

Tous les algorithmes explicités dans ce chapitre se situent dans un contexte où il n'existe aucune variable partagée.

Nous utiliserons le modèle suivant : à chaque processus P_i est associé un autre processus G_i, qui effectue la gestion du contrôle de la ressource pour le processus P_i. Autrement dit, chaque processus P_i possède un gestionnaire G_i spécialisé dans le contrôle. Un processus et son gestionnaire sont situés sur le même site. Lorsqu'un processus P_i désire utiliser la ressource, il envoie un message *Demande_de_la_ressource* à son gestionnaire et attend alors un message *Ressource_disponible* de son gestionnaire ; lorsqu'il libère la section critique, il envoie un message *Restitution_de_la_ressource* à son gestionnaire. Cet échange de messages pourrait être remplacé par une communication plus directe entre un processus et son gestionnaire, par exemple par un partage de variables. Les messages échangés entre un processus et son gestionnaire ne seront pas comptés dans le calcul du nombre de messages nécessaires à un contrôle.

2.2. Jeton tournant sur un anneau : l'algorithme de Le Lann (1977)

Cet algorithme utilise une structure logique en anneau. On suppose que les messages ne peuvent pas être perdus et que les processeurs ne tombent pas en panne. L'algorithm consiste à faire tourner un unique « jeton ».
Le mot *jeton* représente ci-dessous un type de message.

Variables
    demandeur : booléen, initialisé à faux

Réception de messages par G_i, gestionnaire de P_i (les indices sont pris modulo n) :
Demande_de_la_ressource (venant de P_i)
    demandeur ← vrai ;

Jeton
    si *demandeur*, envoyer *Ressource_disponible* à P_i ;
    sinon envoyer *jeton* à G_{i+1} ;

Restitution_de_la_ressource (venant de P_i) *demandeur* ← faux ; envoyer *jeton* à G_{i+1} ;

A l'initialisation, un et un seul des sites exécute le texte correspondant à la réception du jeton.

On peut améliorer cet algorithme pour tenir compte des risques de panne de processeur et de perte du jeton.
L'inconvénient est le nombre de messages transmis, même si la ressource est très peu demandée : le nombre de messages échangés d'une utilisation de la ressource à l'autre est non borné.

2.3. Diffusion et estampillage : l'algorithme de Lamport (1978)

Remarque : La version présentée ici est légèrement modifiée par rapport à celle de Lamport ; l'algorithme initial de Lamport exigeait que les messages ne soient pas déséquencés, cette hypothèse n'est pas utile dans la version ci-dessous.
Cet algorithme utilise un maillage complet et deux techniques : les estampilles pour établir un ordre total entre les demandes d'accès à la ressource, et la gestion d'une file d'attente répartie des processus en attente de la ressource. On fait l'hypothèse que les lignes sont fiables ; les messages ne sont pas perdus.
Le principe de l'estampillage à l'aide d'horloges logiques a été exposé dans le premier chapitre ; sur un site, nous notons h la valeur de l'horloge et l'appelons heure. Un message envoyé par le gestionnaire G_i lorsque son horloge vaut h est estampillé par le couple: (h, i).

Il y a trois types de messages :
- le type requête : un message de ce type est diffusé par un gestionnaire qui désire obtenir la ressource vers tous les autres gestionnaire.
- le type libération : un message de ce type est diffusé par un gestionnaire qui vient de recevoir du processus auquel il est associé un message *Restitution_de_la_ressource* vers tous les autres gestionnaires.
- le type acquittement : lorsque un gestionnaire G_i reçoit d'un gestionnaire G_j un message de type requête, il répond à G_j par un message de type *acquittement*, autrement dit *un accusé de réception*, sauf si le site i est dans le protocole d'acquisition ; dans ce dernier cas, le site G_i ne répond rien.

Chaque site possède un tableau nommé T, indicé par 0, 1, ..., $n-1$ où G_i inscrit, à l'indice j, la valeur maximum des dates reçues dans les estampilles des messages venant du site j.

Une variable nommée *demandeur* permet de savoir si le site correspondant est ou non demandeur de la section critique.
G_i range également en $T[i]$ l'heure de sa dernière requête lorsqu'il est demandeur.

Un gestionnaire qui désire obtenir la section critique pour le processus i dont il gère le contrôle ne peut le faire que si $(T[i], i)$ est inférieur à tous les autres couples $(T[j], j)$; on rappelle que $(T[i], i) < (T[j], j)$ signifie que soit $T[i] < T[j]$, soit $T[i] = T[j]$ et $i < j$.

Réception de messages par G_i, gestionnaire de P_i :

Demande_de_la_ressource(venant de P_i)
    demandeur ← vrai ;
    diffuser (requête, h, i) ;
    $T[i] \leftarrow h$;

***Message*(*type*, *k*, *j*)**
    $h \leftarrow \max(h, k) + 1$; (∗ recalage et incrémentation de l'horloge ∗)
    si ($T[j] < k$), $T[j] \leftarrow k$; (* on prévoit un déséquencement *)
    si (*demandeur*)
        si ($\forall j \neq i$, $(T[i], i) < (T[j], j)$, envoyer *Ressource_disponible* a P_i ;
    sinon si (*type* = requête), envoyer (acquittement, h, i) à j ;

Restitution_de_la_ressource (venant de P_i)
    diffuser (libération, h, i) ;
    demandeur ← faux ;

Montrons que cet algorithme assure l'exclusion mutuelle. Supposons que G_i et G_j soient demandeurs. Notons respectivement T_i et T_j les tableaux de i et j. $T_i[i]$ « contient » la date de la requête de i et $T_j[j]$ celle de la requête de j.
Le site i n'a envoyé aucun message depuis sa requête et les messages de i précédents la requête avait une estampille inférieure à celle de la requête. On a :

$(T_i[i], i) \geq (T_j[i], i)$.

En effet, ce résultat est vrai si la requête de i n'a pas encore été reçue par j du fait du non déséquencement ; il est vrai aussi si la réception a déjà eu lieu car alors $T_i[i] = T_j[i]$.
De même, $(T_j[j], j) \geq (T_i[j], j)$.
Si $(T_i[j], j) > (T_i[i], i)$, alors :
$(T_j[j], j) \geq (T_i[j], j) > (T_i[i], i) \geq (T_j[i], i)$, ce qui entraîne :
$(T_j[j], j) > (T_j[i], i)$: G_j ne peut pas obtenir la ressource.
Seul le gestionnaire qui a effectué la demande la plus ancienne (selon l'ordre total défini par les estampilles) peut obtenir la ressource ; cette situation n'évolue qu'au moment où le gestionnaire envoie un message de libération. D'où l'exclusion mutuelle et plus précisément seul le gestionnaire qui a fait la requête de plus petite estampille peut obtenir la ressource.
Inversement, grâce aux messages d'acquittement, le gestionnaire qui a fait la requête non encore satisfaite de plus petite estampille ne peut pas être bloqué pour l'obtention de la ressource : il n'y a pas d'interblocage. Une requête aura au bout d'un temps fini la plus petite estampille ; il n'y pas famine.

L'algorithme peut être amélioré pour résister aux pannes des gestionnaires (ou des processus associés) ; lorsqu'un gestionnaire tombe en panne, un message est diffusé pour signaler cette panne et le test sur l'ancienneté de la requête n'est alors fait qu'entre les gestionnaires non en panne. Il faut prévoir aussi les remises à jour lors d'une réinsertion de gestionnaire.
On constate qu'une utilisation de la ressource par un processus nécessite l'échange entre les gestionnaires de $3(n - 1)$ messages : $n - 1$ pour la requête, au plus $n - 1$ pour les acquittements, $n - 1$ pour la libération. Ricart et Agrawala ont proposé en 1981 un algorithme qui ne nécessite que $2(n - 1)$ messages ; Carvalho et Roucairol ont présenté la même année un algorithme qui nécessite, selon l'occurrence de l'utilisation, entre 0 et $2(n - 1)$ messages.

2.4. Permissions de chaque site donné pour chaque utilisation de la section critique : un algorithme de Ricart et Agrawala (1981)

On étudie ici un algorithme assez naturel : lorsque le gestionnaire d'un processus désire obtenir la ressource, il demande la permission de l'utiliser à chacun des $n - 1$ autres gestionnaires ; il attend alors d'avoir reçu toutes les permissions.
Évidemment, lorsqu'un gestionnaire reçoit une demande de permission, il faut qu'il se pose la question des priorités ; différents procédés permettent de décider de la priorité ; dans l'algorithme que nous allons expliciter ci-dessous, chaque message est estampillé par l'heure logique et l'identité du demandeur ; le gestionnaire qui reçoit une demande envoie sa permission sauf s'il est demandeur et que l'estampille de sa demande est inférieure à celle du site demandant une permission. Dans le cas où le gestionnaire n'envoie pas sa permission immédiatement, il l'enverra lorsque le processus pour lequel il gère le contrôle de la ressource aura restituer celle-ci.
Les noms des sites sont supposés être 1, 2, ..., n.
Variables d'un gestionnaire :

- *demandeur* : booléen initialisé à faux ;
- *h :* entier positif initialisé à 0 ;
- *heure_demande* : entier ;
- *permissions* : ensemble d'identificateurs de sites initialisé à $\{i\}$;
- *en_attente* : ensemble d'identificateurs de sites initialisé à $\varnothing$.

Réception de messages par G_i, gestionnaire de P_i :

Demande_de_la_ressource (venant de P_i)
$h \leftarrow h + 1$;
demandeur ← vrai ;
heure_demande ← *h ;*
pour tout j de $\{1, 2, ..., n\} - \{i\}$,
envoyer *Demande_de_permission*(*heure_demande*, i) à j ;

***Demande_de_permission*(*k*, *j*)**
$h \leftarrow \max(h, k)$;
si (*demandeur*) et (*heure_demande*, i) < (k, j)), *en_attente* ← (*en_attente* ∪ $\{j\}$) ;
sinon (* alors $j \notin$ *permissions* *) envoyer *Permission*(i) à j ;

***Permission*(*j*)**
permissions ← *permissions* ∪ $\{j\}$;
si *permissions* = $\{1, 2, ..., n\}$, envoyer *Ressource_disponible* à P_i ;

Restitution_de_la_ressource **(venant de P_i)**
demandeur ← faux ;
pour tout j dans en_attente,
envoyer *Permission*(i) à j ;
permissions ← *permissions* − $\{j\}$;
permissions ← $\{i\}$;
en_attente ← $\varnothing$;

On peut prouver la sûreté de cet algorithme, même si les lignes de transmission des messages ne sont pas fifos (c'est-à-dire s'il est possible qu'il y ait déséquencement).
On remarque que les gestionnaires obtiennent la ressource dans l'ordre donné par les estampillages, ce qui assure la vivacité de l'algorithme.
L'obtention de la ressource nécessite toujours exactement $2(n-1)$ messages.
Si on suppose que le temps de transmission des messages entre gestionnaires est de T et que l'on néglige les temps de transmission des messages entre un processus et le gestionnaire qui lui est associé (ce qui est légitime puisqu'un processus et son gestionnaire se trouvent sur le même site), la durée pendant laquelle la ressource est inutilisée alors qu'au moins un processus est demandeur est comprise entre T et $2T$.

Enfin, remarquons qu'il est possible de borner les compteurs logiques de temps. En effet, par construction, les dates des demandes de permission en cours à un moment donné forment un intervalle d'entiers ; puisque le nombre de sites est n, si le site i fait une demande de permission estampillée avec l'heure heure_demande, c'est que les demandes de permission qu'il peut recevoir sont estampillées par une heure k vérifiant :

$$-(n-1) \leq heure\_demande - k \leq n-1.$$

Supposons que les heures soit en fait mémorisées modulo $2n$ - 1 (ou modulo un entier au moins égal à $2n-1$), les messages étant alors estampillés avec les heures modulo $2n-1$; le calcul de la différence $heure\_demande - k$ sera alors exact modulo $2n-1$; l'intervalle d'entiers $[-(n-1), n+1]$ contenant $2n-1$ entiers, le nombre effectif $heure\_demande - k$ peut être parfaitement déterminé.

2.5. Permissions de chaque site gardées tant qu'elles ne sont pas réclamées : l'algorithme de Carvalho et Roucairol (1983)

Cet algorithme est une amélioration du précédent dans la mesure où l'utilisation de la ressource nécessite en moyenne l'utilisation de moins de messages.

On peut considérer que, pour chaque paire de sites, il existe un unique jeton nommé jeton(i, j). Un processus i ne peut utiliser la ressource que si son gestionnaire possède pour tout j le jeton (i, j). Lorsqu'un gestionnaire veut obtenir la ressource, il fait donc des requêtes pour tous les jetons qui lui manquent aux sites concernés, et uniquement pour ceux-ci. Lorsqu'un site i reçoit d'un site j la demande de jeton(i, j), il l'envoie uniquement s'il calcule qu'il n'est pas prioritaire ; sinon, il enverra le jeton lorsqu'il aura eu satisfaction sur l'utilisation de la ressource. Le calcul des priorités se fait à l'aide d'estampillages de manière analogue à l'algorithme précédent.
On n'utilise pas en fait explicitement ces jetons ; si le site i n'a pas jeton(i, j), on traduit cela sur le site i par $j \in permissions$.

Variables :

- *état* : peut prendre les valeurs demandeur, utilisateur, ailleurs, initialisée à ailleurs ;
- *h :* entier positif initialisé à 0.
- *heure_demande* : entier. ;
- *permissions* : ensemble d'identificateurs de sites initialisé de telle sorte que :
  sur le site i, $j \in permissions \Leftrightarrow$ sur le site j, $j \notin permissions$;
- *en_attente* : ensemble d'identificateurs de sites initialisé à $\varnothing$.

Réception de messages par G_i, gestionnaire de P_i :
Demande_de_la_ressource (venant de P_i)
$h \leftarrow h + 1$;
si ($permissions = \{1, 2, ..., n\}$)
$état \leftarrow$ utilisateur ;
envoyer *Ressource_disponible* à P_i ;
sinon
$etat \leftarrow demandeur$;
$heure\_demande \leftarrow h$;
pour tout $j \notin permissions$, envoyer *Demande_de_permission*(h, i) à j ;

***Demande_de_permission*(*k*, *j*)**
$h \leftarrow \max(h, k)$;
si ($état$ = utilisateur) ou (($état = demandeur$) et ($heure\_demande$, i)) < (k, j)))
$en\_attente \leftarrow (en\_attente \cup \{j\})$;
sinon
envoyer *Permission*(i) à j ;
$permissions \leftarrow permissions - \{j\}$;
si ($état$ = demandeur)
envoyer *Demande_de_permission*(*heure_demande*, i) à j ;

***Permission*(*j*)**
$permissions \leftarrow permissions \cup \{j\}$;
si ($permissions = \{1, 2, ..., n\}$)
$état \leftarrow$ utilisateur ;
envoyer *Ressource_disponible* à P_i ;

Restitution_de_la_ressource (venant de P_i)
$état \leftarrow$ ailleurs ;
pour tout j dans en_attente,
envoyer *Permission*(i) à j ;
$permissions \leftarrow permissions - \{j\}$;
$en\_attente \leftarrow \varnothing$;

Sûreté et vivacité sont là encore assurées. L'utilisation de la ressource nécessite entre 0 et $2(n - 1)$ messages. Si le temps de transmission d'un message est de T, la durée pendant laquelle la ressource est inutilisée alors qu'elle est demandée est comprise entre 0 et $2T$.
En revanche, les décalages entre les horloges ne sont pas bornés ; en conséquence, il n'est pas possible de borner les valeurs des horloges.

2.6. Permissions sans estampillage : l'algorithme de Chandy et Misra (1984)

La différence essentielle entre cet algorithme et l'algorithme précédent est que les règles de priorité sont définies sans utiliser d'estampillage de messages. Avec les mêmes notations que pour la présentation de l'algorithme précédent, lorsqu'un site i reçoit de j la permission de i, il est prioritaire sur j pour une utilisation de la ressource ; lorsque cette utilisation aura été

effectuée, la priorité sera à j ; si le site j demande permission à i, i lui enverra cette permission.
L'absence d'estampillage des messages évite aux sites de gérer une heure logique ; elle évite aussi la nécessité que chaque site ait un nom. Il suffit que chaque site puisse identifier les canaux de transmission qui le relient aux $n - 1$ autres sites.

Nous nous intéressons au gestionnaire G du processus P, nous notons c_1, c_2, ..., c_{n-1} les canaux qui lient le site de ce gestionnaire aux autres sites.

Variables :
- *état* : peut prendre les valeurs demandeur, utilisateur, ailleurs, initialisée à ailleurs ;
- *utilisé*: tableau indicé par c_1, c_2, ..., c_{n-1} de booléens ; *utilisé*(c_j) est à vrai si G possède la permission venant du canal c_j et si le processus pour qui il gère le contrôle a utilisé la ressource au moins une fois depuis qu'il possède cette permission ; sinon cette variable vaut faux ;
- *permissions* : ensemble d'identificateurs de canaux ;
- *en_attente* : ensemble d'identificateurs de canaux initialisé à ∅.

La façon d'initialiser les variables « *permissions* » et « *utilisé* » est explicitée plus bas.

Réception de messages par *G*, gestionnaire de *P* :
Demande_de_la_ressource (venant de P)
si *permissions* = (c_1, c_2 …,c_{n-1})
état ← utilisateur ;
envoyer *Ressource_disponible* à P ;
sinon
etat ← demandeur ;
pour tout $c_j \notin$ *permissions*,
envoyer *Demande_de_permission* par c_j ;

Demande_de_permission **arrivant par le canal c_j**
(* l'arrivée de cette demande montre que G possède cette permission ou bien va la recevoir *)
si (*état* = utilisateur) ou (*utilisé*(c_j) = faux)
en_attente ← *en_attente* ∪ {c_j} ;
sinon
envoyer *Permission* par c_j ;
permissions ← *permissions* – {c_j} ;
si (*état* = demandeur) envoyer *Demande_de_permission* par c_j ;

Permission **arrivant par le canal c_j** (* nécessairement, état = *demandeur* *)
permissions ← *permissions* ∪ {c_j} ;
utilisé(c_j) ← faux ;
si *permissions* = (c_1, c_2 …,c_{n-1})
envoyer *Ressource_disponible* à P ;
état ← utilisateur ;

Restitution_de_la_ressource (venant de P)
    état ← ailleurs ;
    pour tout c_j
        si c_j dans *en_attente*,
            envoyer *Permission* par c_j ;
            permissions ← *permissions* – {j} ;
        sinon *utilisé*(c_j) ← vrai ;
    en_attente ← ∅ ;

Il faut être vigilant pour l'initialisation des ensembles *permissions* et des tableaux appelés « *utilisé* ». Précisons les hypothèses qu'on veut avoir à l'initialisation et conserver au cours de l'exécution.
Il faut tout d'abord que, pour chaque paire de gestionnaires, il y ait une permission virtuelle qui soit « chez » l'un, ou « chez » l'autre, ou en transition de l'un à l'autre, et en aucun cas « chez » les deux gestionnaires à la fois. Cela se traduit par le fait que, en appelant c le canal joignant le gestionnaire G au gestionnaire H :

(1) soit la permission est « chez » G :
$c \in$ *permissions* pour G et $c \notin$ *permissions* pour H
(2) soit la permission est « chez » H :
$c \in$ *permissions* pour H et $c \notin$ *permissions* pour G
(3) soit une permission est en transit sur c de G vers H
(4) soit une permission est en transit sur c de H vers G.

G est prioritaire sur H si et seulement si :

- on est dans le cas (1) et, pour G, *utilisé*(c) est à faux
- on est dans le cas (2) et, pour H, *utilisé*(c) est à vrai,
- on est dans le cas (4).

Considérons le graphe des priorités ; il s'agit du graphe dont les sommets sont les gestionnaires et où il existe un arc de G vers H si H est prioritaire sur G. Il est facile de vérifier que l'absence d'interblocage est assurée par l'absence de circuit dans le graphe des priorités ; en effet, la décision pour un site d'envoyer ou non une permission respecte les priorités ; un gestionnaire i prioritaire sur un gestionnaire j recevra la permission de celui-ci ; si le graphe des priorités est sans circuit, il existe un gestionnaire prioritaire sur tous les autres, celui-ci obtiendra la ressource.
L'initialisation doit assurer cette absence de circuits. Pour cela, il suffit de considérer au départ un ordre total sur les gestionnaires ; si G se trouve avant H dans cet ordre, on met la permission chez G (c $\in$ *permissions* pour G et $c \notin$ *permissions* pour H) ; par ailleurs, on donne la valeur vrai à toutes les composantes de tous les tableaux « *utilisé* ». Dans le graphe des priorités, l'arc entre G et H va de G vers H si et seulement si G est avant H dans l'ordre total considéré.
Les arcs du graphe des priorités ne s'inversent que lorsqu'un processus P libère la ressource ; le gestionnaire du processus P a alors un demi-degré intérieur nul dans ce graphe. Il ne peut appartenir à aucun circuit : aucune inversion d'arcs au cours de l'algorithme ne crée donc de circuit. L'initialisation proposée assure à l'algorithme l'absence d'interblocage.
On peut enfin vérifier que cet algorithme a la qualité de vivacité.

2.7. Jeton circulant par diffusion : un autre algorithme de Ricart et Agrawala (1983)

Cet algorithme est très performant en ce qui concerne le nombre de messages échangés pour chaque utilisation de la ressource, puisque ce nombre est, selon les cas, n ou 0 ; néanmoins, la taille de certains de ces messages est beaucoup plus grande que dans le cas de l'algorithme de Lamport ou des trois autres algorithmes cités ci-dessus.
On suppose que les messages ne sont jamais perdus ni altérés, mais on accepte l'éventualité de messages déséquencés.

L'algorithme est un peu intermédiaire entre l'algorithme de Le Lann (il utilise un jeton qui circule selon un anneau logique, mais en sautant les sites des processus non demandeurs) et l'algorithme de Lamport (un processus qui désire utiliser la ressource mais ne possède pas le jeton diffuse une requête suivant un maillage complet).
Un gestionnaire G_i ne peut envoyer un message *Ressource_disponible* au processus dont il gère le contrôle que lorsqu'il possède l'unique jeton. S'il veut obtenir la ressource et qu'il ne possède pas le jeton, il diffuse à tous les autres processus un message de type *requête* qui correspond à une requête faite pour avoir le jeton ; ce message comporte deux renseignements : le numéro de séquence de cette requête (ce numéro, initialisé à 1, est incrémenté de 1 à chaque requête), le numéro d'identification du gestionnaire (i pour G_i). Il attend alors le jeton.
Lorsque G_i reçoit du processus dont il gère le contrôle la *Restitution de la ressource*, il regarde s'il a reçu de G_{i+1} une requête pour le jeton non encore satisfaite ; si oui, il lui envoie le jeton ; si non, il reprend avec G_{i+2}, ..., puis G_{n-1}, puis G_0, ..., puis G_{i-1} ; s'il n'a ainsi trouvé aucune requête non satisfaite, il garde le jeton ; il pourra le réutiliser quand il en aura besoin (en envoyant un message *Ressource_disponible* au processus dont il gère le contrôle) sauf si, entre temps, il reçoit une requête pour le jeton qu'il doit alors satisfaire par envoi du jeton.
Le jeton se déplace à l'aide d'un message qui indique (en plus de son type) pour tout i compris entre 0 et $n-1$, le nombre de fois où le jeton est passé chez G_i c'est-à-dire le numéro de séquence (dont il a déjà été question plus haut) de la dernière requête satisfaite de G_i ; cette information sera notée *jeton*[i]. L'ensemble de ce message sera noté ci-dessous *jeton*. Le *jeton* est donc ici porteur de renseignements sur l'état global du système.
Dans le pseudo-code ci-dessous, on utilise une instruction d'écriture dans *jeton*, ce qui consiste en fait à copier le jeton, écrire dans la copie puis à recomposer le message *jeton* avec cette copie modifiée.
copi
Les variables locales à un site i sont :

- *num_seq* (pour numéro de séquence) : compteur croissant initialisé à 0 indiquant le nombre de requêtes du jeton faites par G_i.
- *tab_requête* : tableau indicé par 0, 1, ..., $n-1$; G_i note dans t*ab_requête*[j] le numéro de séquence de la requête la plus récente qu'il ait reçue de P_j. Si *tab_requête*[j] > *jeton*[j], c'est que G_j attend le jeton.
- *jeton_présent* : booléen initialisé à faux sauf pour l'un des processus où l'initialisation est à vrai ; *jeton_présent* est vrai si et seulement si G_i « possède le jeton ».
- *dedans* : booléen initialisé à faux qui est à vrai entre le moment où le gestionnaire envoie un message *Ressource_disponible* au processus dont il gère le contrôle et le moment où il reçoit de celui-ci un message *Restitution_de_la_ressource.* ;

- une variable *j* qui est un compteur de boucle.

Les actions effectuées à la réception d'un message sont supposées indivisibles.

Réception de messages par G_i, gestionnaire de P_i :

Demande_de_la_ressource (venant de P_i)

si non *jeton_présent*
num_seq ← *num_seq* + 1 ;
diffuser *requête*(*num_seq*, *i*) ;
sinon
envoyer *Ressource_disponible* à P_i ;
dedans ← vrai ;

***Requête*(*k*, *j*)**

si (*k* > *tab_requête*[*j*]) (∗ on prévoit un risque de déséquencement ∗)
tab_requête [*j*] ← *k*) ;
si (*jeton_présent*) et (*non dedans*)
jeton_présent ← faux ;
envoyer (*jeton*) à G_j ;

Jeton (*le processus associé attend nécessairement la ressource*)

jeton[*i*] ← *num_seq* ;
jeton_présent ← vrai ;
envoyer *Ressource_disponible* à P_i ;
dedans ← vrai ;

Restitution_de_la_ressource (venant de P_i)

dedans ← faux ;
j ← (*i* + 1) mod *n* ;
tant que (*jeton_présent* et *j* ≠ *i*)
si *tab_requête*[j] > *jeton*[*j*]
jeton_présent ← faux ;
envoyer (*jeton*) à G_j ;
j ← (*j* + 1) mod *n* ;

L'exclusion mutuelle est vérifiée du fait de l'unicité du jeton. On peut vérifier aussi l'absence d'interblocage ou de famine.

2.8. Gestion d'une file d'attente avec une arborescence : l'algorithme de Naimi et Trehel (1987)

Le plus simple pour gérer une ressource en exclusion mutuelle est la gestion interne d'une file d'attente. L'algorithme que nous allons exposer permet une telle gestion. Nous allons gérer une file d'attente dans laquelle un gestionnaire entre quand le processus dont il gère le contrôle demande la ressource et dont il sort lorsque celui-ci libère la ressource.

Le principe est de gérer une arborescence dont la racine correspond au gestionnaire qui est le dernier dans la file d'attente, sauf dans le cas où la file est vide. Lorsqu'un gestionnaire *H*

veut se mettre dans la file d'attente, il transmet sa demande en faisant « remonter » sa requête jusqu'à la racine de l'arborescence : le gestionnaire *G* qui s'y trouve pourra alors noter que *H* est le gestionnaire qui le suit dans la file ; au fur et à mesure que les messages « remontent », certains arcs sont modifiés pour que *H* devienne la racine de la nouvelle arborescence ; la façon de modifier ces arcs n'est pas unique ; celle choisie par Naimi et Trehel en est une parmi d'autres.
Les sites sont appelés 1, 2, ..., *n* et le gestionnaire qui est sur *i* est G_i.
Un message appelé *jeton* indique à un gestionnaire qu'il est le premier dans la file d'attente (c'est alors à lui d'utiliser la section critique).
Les variables utilisées sur un site *i* sont :

- *a_un_père* : booléen qui prend la valeur faux uniquement pour la racine de l'arborescence
- *père* qui indique, pour les gestionnaires autres que celui correspondant à la racine, le père de G_i dans l'arborescence ; toutes ces variables sont évidemment initialisées pour qu'au départ l'ensemble des définitions des « *père*(*s*) » constituent une arborescence ;
- *est_demandeur :* booléen qui indique si G_i est demandeur de la ressource ; ce booléen n'est utile que pour détecter le cas où la file est vide : la racine de l'arborescence n'est alors pas dans la file ;
- *a_un_suivant* : booléen à vrai si le gestionnaire est dans la file d'attente et pas le dernier dans cette file
- *suivant* : indique éventuellement le nom du suivant de G_i dans la file d'attente
- *jeton_présent* : booléen qui est à vrai si le site concerné utilise la ressource ou bien éventuellement si le gestionnaire est racine de l'arborescence et soit qu'aucun site n'est demandeur, soit que des messages des demandes sont en transit ; cette variable est initialisée à vrai uniquement pour le gestionnaire pour lequel *a_un_père* est initialisé à faux.

L'algorithme est tel que qu'on a toujours hors des blocs des codes, insécables, des fonctions :

- soit *a_un_père* vaut vrai ;
- soit *jeton_présent* vaut vrai ;
- soit *est_demandeur* vaut vrai et alors *a_un_suivant* vaut faux.

Réception de messages par G_i, gestionnaire de P_i :
Demande_de_la_ressource (venant de P_i)

```
    est_demandeur ← vrai ;
    si a_un_père
        envoyer requête(i) à père ;
        a_un_père ← faux ;
    sinon (*nécessairement, jeton_présent est vrai*)
        envoyer Ressource_disponible à Pi ;
```

```
Requête( j)
        si a_un_père, alors
                envoyer requête(j) à père ;
                père ← j;
        sinon
                a_un_père ← vrai ;
                père ← j;
                si  est_demandeur
                        a_un_suivant ← vrai ;
                        suivant ← j ;
                sinon (*nécessairement, jeton_présent est vrai*)
                         envoyer jeton à j ;
                        jeton_présent ← faux ;

Jeton
        jeton_présent ← vrai ;
        envoyer Ressource_disponible à P_i ; (*nécessairement, celui-ci en est demandeur*)

Restitution_de_la_ressource (venant de P_i)
         est_demandeur  ← faux ;
        si a_un_suivant
                envoyer jeton à suivant ;
                jeton_présent ← faux ;
                a_un_suivant ← faux ;
```

Le nombre de messages nécessaires à l'obtention du jeton est compris entre 0 et n. La moyenne de ce nombre est de O(ln (n)).
Si le temps de transmission d'un message est T, le temps de repos de la section critique alors qu'un site est demandeur varie entre 0 et nT, avec une moyenne de O(ln (n)).

2.9. Caractéristiques des algorithmes d'exclusion mutuelle

	Nombre de messages	Temps inutilisé	Compteurs non bornés	Maillage
Le Lann (77)	non borné	0 à nT	non	anneau
Lamport (78)	entre $2(n-1)$ et $3(n-1)$	T à $2T$	oui ?	complet
Ricart et Agrawala(81)	$2(n-1)$	T à $2T$	non	complet
Carvalho et Roucairol (83)	0 à $2(n-1)$	0 à $2T$	oui	complet
Chandy et Misra (84)	0 à $2(n-1)$	0 à $2T$	non	complet
Ricart et Agrawala (83)	0 ou n	0 à $2T$	oui ?	complet
Naimi et Trehel (87)	0 à n moy. = O(log n)	0 à $n\,T$ moy. = O(T log n)	non	variable

Chapitre 3 - Le problème de l'élection, le parcours de réseau

Nous traitons dans ce chapitre le problème de l'élection et nous étudions un mécanisme, les parcours du réseau, qui peut servir de base à de nombreux algorithmes lorsque le maillage est quelconque.

3.1. Le problème de l'élection

1. Généralités

De nombreux algorithmes de contrôle, bien que dits distribués, sont en fait centralisés dans le sens qu'un processeur particulier joue le rôle de coordonnateur et effectue certains services à la demande des autres processeurs ; c'est le schéma du protocole « client-serveur ». On peut ainsi mettre en œuvre n'importe quel algorithme centralisé dans un système distribué ; des échanges de messages permettent de calquer les lectures et écritures sur des mémoires communes et de faire des « appels de procédure à distance ».

Ce type d'algorithme présente un inconvénient majeur : la panne du processus coordonnateur entraîne l'arrêt du fonctionnement du système. Il faut alors que les processus en état de fonctionnement puissent « s'entendre » pour décider lequel d'entre eux prend désormais le rôle de coordonnateur. Cet accord s'obtient grâce aux algorithmes d'élection. Les protocoles d'élection peuvent aussi être utilisés chaque fois qu'un processus, sans nécessairement être coordonnateur, joue un rôle particulier.

Les processus sont en général identifiés par des numéros et on décide que le processus qui joue un rôle à part est celui qui possède le plus grand (ou le plus petit) numéro. Un algorithme d'élection a donc pour objectif de déterminer le processus de plus grand numéro.

Nous présentons un seul algorithme, l'algorithme de Chang et Roberts (1979) ; cet algorithme, qui utilise un maillage en anneau, nécessite au pire $O(n^2)$ messages, et en moyenne $O(n.\ln(n))$, pour une élection entre n processus ; d'autres algorithmes, un peu plus compliqués, utilisant aussi un maillage en anneau, ne nécessitent que $O(n.\ln(n))$ messages au pire pour le même travail. Par ailleurs, les algorithmes de parcours qui seront donnés dans la deuxième partie de ce chapitre permettent de définir d'autres algorithmes d'élection utilisant un maillage quelconque.

2. Algorithme de Chang et Roberts (1979)

Le maillage est un anneau logique unidirectionnel, où chaque processus sait adresser un message à son voisin de gauche par la primitive *envoyer_voisin_gauche*. Un ou plusieurs processus peuvent décider de lancer un protocole d'élection. Sur chaque site, on trouve :

- *mon_numéro* : entier constant qui donne l'identificateur du site ; rappelons que l'algorithme a pour objectif de déterminer le plus grand de ces numéros et de faire connaître aux différents sites la valeur de ce plus grand numéro.
- *plus_grand :* variable entière donnant à un instant donné le plus grand numéro de processus connu sur le site.
- *participant* : variable booléenne initialisée à faux, qui passe à vrai dès que le processus est touché par l'algorithme d'élection.
- *vainqueur* : variable entière qui, à la fin de l'exécution, aura pour valeur le plus grand des numéros, c'est-à-dire le numéro du processus élu.

Les messages sont de deux types :

- les messages de type *élection* ; un processus envoie un tel message soit quand il décide de lancer une élection (il lui fait alors contenir son propre numéro), soit quand il devient participant (il calcule alors *plus_grand* et fait contenir au message cette valeur), soit quand, déjà participant, il vient d'incrémenter *plus_grand* (il lui fait alors contenir *plus_grand*).
- les messages de type élu qui font circuler, après qu'il a été déterminé, le plus grand numéro de processus ; un processus qui reçoit un message de type élection portant son numéro sait qu'il est celui de plus grand numéro et envoie le premier message de type élu ; ce message est alors retransmis en faisant le tour de l'anneau.

texte exécuté lors de la décision de provoquer une élection

participant ← vrai ;
plus_grand ← *mon_numéro* ;
envoyer_voisin_gauche (élection, *mon_numéro*) ;

texte exécuté lors de la réception de (élection, j)

si non *participant*,
participant ← vrai ;
plus_grand ← max(*mon_numéro*, *j*).
envoyer_voisin_gauche (élection, *plus_grand*) ;
sinon
si *j* > *plus_grand*,
plus_grand ← *j* ;
envoyer(élection, *plus_grand*) ;
sinon si *j* = *mon_numéro*,
envoyer_voisin_gauche (élu, *mon_numéro*);

texte exécuté lors de la réception de (élu, j)

vainqueur ← j ;
participant ← faux ; (∗ initialisation pour la prochaine élection ∗)
si *j* ≠ *mon_numéro*, *envoyer_voisin_gauche* (élu, *j*) ;

Un message de type élection n'est pas retransmis par un processus participant si le numéro porté par le message n'est pas supérieur au plus grand numéro déjà connu sur le site ; un tel principe s'appelle principe de l'*extinction sélective*.

La preuve de l'algorithme se fait aisément en vérifiant qu'un et un seul numéro fait le tour de l'anneau à l'aide des messages de type élection : le plus grand des numéros.
Du point de vue de la performance en temps, il faut toujours le temps d'un tour d'anneau entre le moment où le vainqueur sait qu'il est vainqueur et la fin du protocole ; on peut cependant distinguer deux cas extrêmes :

- le futur vainqueur lance en premier (éventuellement en même temps que d'autres processus) une élection ; il faut alors le temps de la transmission de n messages (le tour de l'anneau) pour qu'il sache qu'il est vainqueur.
- le futur vainqueur est à droite d'un unique processus déclenchant l'élection ; il faudra alors le temps de $n - 1$ messages pour que le futur vainqueur devienne participant puis le temps de n messages pour que celui-ci découvre qu'il est vainqueur, soit le temps de la transmission de $2n - 1$ messages.

Dans tous les cas, le temps requis est en $O(n)$.

Du point de vue de la performance en nombre de messages, le cas le plus favorable est celui où seul le futur vainqueur lance l'élection ; le nombre total de messages envoyés est alors $2n$. Le cas le plus défavorable est celui où les numéros sont en ordre décroissants et tous les processus lancent l'élection en même temps ; il y alors $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ transferts de messages pour déterminer le vainqueur (suivis de n messages pour faire tourner l'information). Nous ne calculons pas ici le nombre moyen ($O(n.\ln(n))$) de transferts de messages.

3.2. Parcours d'un réseau

1. Généralités

De nombreuses applications nécessitent de faire parcourir par des messages le réseau qui supporte un système distribué. Certains de ces parcours doivent nécessairement se faire de façon séquentielle (on fait alors en général un parcours dit *en profondeur d'abord*) mais la plupart peuvent se faire de façon parallèle, ce qui permet de gagner du temps. Nous étudierons uniquement une solution donnant un parcours en parallèle.
Parmi les applications qu'on peut construire à l'aide d'algorithmes de parcours, citons :

- la diffusion d'une information
- l'étude de la fiabilité du réseau : existence de points d'articulation, calcul de la connectivité
- l'initialisation des systèmes (donner un nom à chaque processus alors que ne sont connues par le système que les voies de communication)
- les problèmes d'élection
- la détermination d'un interblocage
- la détermination de la terminaison (tous les processus sont terminés).

Lorsqu'on effectue un parcours de réseau à partir d'un site P_r, on construit une arborescence dont les sommets sont l'ensemble des processus et dont la racine est P_r.

2. Parcours en parallèle

Le hypothèses faites sur le réseau sont :
- le réseau est non orienté (les lignes sont bidirectionnelles) et connexe.
- les sites ont tous des noms différents ; ces noms sont des entiers naturels.
- les sites ne connaissent pas la topologie du réseau ; ils ne connaissent que les noms de leurs voisins, et les lignes les reliant à ces voisins.
- les lignes sont fiables.

Le fait que le graphe soit quelconque et que la connaissance du réseau soit locale facilite des modifications éventuelles du réseau ; la suppression ou l'ajout d'un site ne nécessite que la redéfinition (et la re-compilation) des sites dont le voisinage a été modifié.

Lors d'un parcours de graphe, le processus qui a lancé le parcours a souvent besoin d'être informé de la terminaison du parcours, afin de pouvoir enchaîner sur d'autres actions. L'algorithme que nous proposons atteint cet objectif.
Nous supposons ici qu'un seul processus, celui de nom *r*, peut lancer un parcours et que celui-ci ne peut lancer un parcours que quand le précédent est terminé. Nous laissons au lecteur le soin de modifier cet algorithme dans le cas où tous les processus peuvent simultanément lancer un parcours.

Le parcours sert ici à diffuser une information issue de *r*, noté *info*, et à collecter un ensemble d'informations se situant sur les différents sites ; on peut par exemple imaginer qu'il s'agit d'additionner des valeurs entières se trouvant dans les différents sites ; on notera *info'* l'information existant au départ sur un site puis collectée au fur et à mesure ; dans le cas de notre exemple, *info'* sera la somme des entiers déjà collectés.

Les identificateurs utilisés par le site P_i sont :
- *marqué* : booléen initialisé à faux, qui indique si P_i a déjà été atteint par le parcours ;
- *voisins* : ensemble constant d'entiers constitué des noms des voisins du sommet *i ;*
- *père* : entier qui contiendra (sauf pour P_r) le père de P_i dans l'arborescence ;
- *nb_attendus* : entier qui donne, après que P_i a été marqué, le nombre de messages que P_i attend encore avant d'indiquer à son père que le parcours est terminé dans la branche dont il est racine.
- *terminé* : pour P_r uniquement , initialisée à faux.

Tout message commence par l'indication du type de ce message. Les messages sont de deux types :
- *parcours* : un tel message, après avoir précisé son type, indique l'information *info* qu'il diffuse, puis le numéro du site émetteur. Un processus, qui n'est pas encore marqué, et qui reçoit (parcours, *info*, *j*) sait que *j* est son père et doit envoyer un message de type parcours à tous ses voisins autres que son père ; il initialise sa variable fils à l'ensemble des noms de ses voisins autres que son père et le nombre d'acquittements attendus au cardinal de cet ensemble. S'il est déjà marqué, il diminue de 1 le nombre de messages qu'il attend.
- *retour* : un tel message, après avoir précisé son type, indique en seconde position la valeur de *info'* ; le site qui reçoit ce message utilise *info*' et les données qui se trouvent chez lui pour actualiser *info*'. Chaque fois qu'un processus reçoit un message de type retour, il diminue de 1 le nombre de messages qu'il attend.

Protocole du site P_i

texte exécuté lors de la décision de P_r de lancer un parcours

marqué ← vrai ;
nb_attendus ← *cardinal* (*voisins*) ;
$\forall j \in$ *voisins*, envoyer (parcours, *info*, *r*) à P_j ;

texte exécuté lors de la réception de (parcours, info, j)

si *marqué*, *nb_attendus* ← *nb_attendus* – 1 ;
sinon
marqué ← vrai ;
père ← j ;
nb_attendus ← cardinal (voisins) – 1 ;
$\forall j \in$ *voisins – père*, envoyer (parcours, *info*, *i*) à P_j ;
si *nb_attendus* = 0, envoyer (retour, *info*') à *père*

texte exécuté lors de la réception de (retour, info') par P_i

nb_attendus ← *nb_attendus* – 1 ;
actualiser *info*' ;
si *nb_attendus* = 0, si $i \neq r$, envoyer (retour, *info*') à *père*;
sinon *terminé* ← vrai ;

Le processus P_r sait que le parcours est terminé quand sa variable *nb_attendus* vaut 0.
Sur chaque ligne du réseau transite au plus un message de type parcours dans chaque sens ; s'il y a un message de type parcours dans chaque sens, il n'y aura pas de message de type retour, ni dans un sens, ni dans l'autre ; s'il y a un seul message de type parcours, il y aura un message de type retour dans l'autre sens ; il circule ainsi deux messages sur chaque ligne. Le nombre de messages échangés est donc O(*e*), où e représente le nombre de lignes du réseau.
Si la plus grande distance en nombre de liaisons entre P_r et les autres sites est *d* et si le temps maximum de transfert d'un message est Δ, la complexité du temps est O(*d*, Δ).

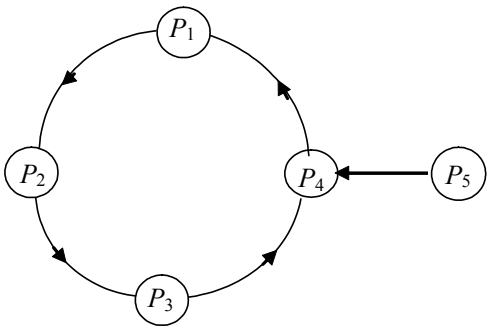
Chapitre 4 - Propriété globale

On appelle propriété globale une propriété qui concerne l'ensemble des sites. Nous verrons le cas de l'interblocage, de la terminaison et du calcul d'un état dit global.

4.1. L'interblocage

1. Généralités

Supposons qu'un processus P_1 attende l'occurrence d'un événement qui doit être produit par un processus P_2, que celui-ci attende l'occurrence d'un événement qui doit être produit par un processus P_3 et qu'enfin P_3 attende l'occurrence d'un événement qui doit être produit par le processus P_1 : les trois processus sont contraints, en l'absence d'intervention, à attendre éternellement : il y a *interblocage* ; le même phénomène peut avoir lieu avec tout ensemble d'au moins deux processus.
Étant donné un ensemble de processus participant à un système réparti, on peut considérer ces processus comme les sommets d'un graphe orienté appelé *graphe des attentes* : on joint par un arc un processus P_i à un processus P_j si le processus P_i doit attendre pour continuer à s'exécuter un événement produit par P_j.



Il y a interblocage lorsqu'il existe un circuit dans ce graphe des attentes ; néanmoins, les processus de ce circuit ne sont pas les seuls à être bloqués ; sur l'exemple ci-contre, le processus P_5 se trouve lui aussi bloqué. Tout processus se trouvant sur un chemin qui conduit à un circuit est bloqué.

Une situation d'interblocage peut se produire en particulier dans les cas suivants :
- plusieurs processus se partagent des ressources à accès exclusif ;
- le système utilise des primitives de communication synchrones : si P_1 attend P_2, si P_2 attend P_3 et si P_3 attend P_1, ces trois processus risquent d'attendre longtemps.

Dans ce paragraphe consacré à l'interblocage, nous nous intéresserons uniquement au cas du partage des ressources. Les ressources doivent être utilisées en exclusion mutuelle et certains processus doivent utiliser plusieurs ressources simultanément.
Dans les systèmes où un risque d'interblocage existe, le problème est d'empêcher la formation de circuits dans le graphe des attentes ou bien de détecter la présence de tel circuit. Une technique où on cherche à prévenir l'apparition d'interblocage est dite *a priori*, ou encore

pessimiste ; une technique où on cherche à détecter une situation d'interblocage pour la faire disparaître est dite *a posteriori*, ou encore *optimiste*.
Nous utilisons ci-dessous le terme de *transaction*, terme provenant des bases de données, pour désigner une action faite par un processus nécessitant l'utilisation de plusieurs ressources.

2. Prévention de l'interblocage dans l'allocation de ressources

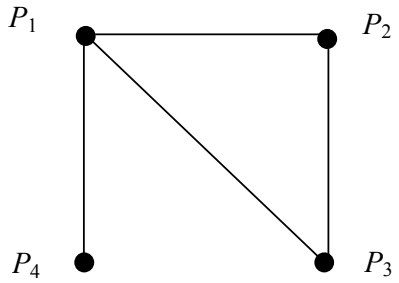
Les méthodes peuvent être différentes selon qu'on suppose ou non qu'une transaction peut déclarer à l'avance l'ensemble des ressources qui lui sont nécessaires.

a) Cas où les ressources nécessaires aux transactions d'un même processus sont fixes.

On suppose ici qu'un même processus fera les mêmes demandes de ressources pour toutes ses transactions. Prenons un exemple.

- Un processus P_1 utilise toujours simultanément R_1, R_2, R_3.
- Un processus P_2 utilise toujours simultanément R_1, R_3.
- Un processus P_3 utilise toujours simultanément R_1, R_4.
- Un processus P_4 utilise toujours uniquement R_2.

On considère alors le graphe ci-contre appelé le *graphe des conflits* :



Dans ce graphe, un sommet étiqueté par P_i est joint à un sommet étiqueté par P_j s'il n'est pas possible qu'une transaction de P_i soit simultanée avec une transaction de P_j.
Tout algorithme d'exclusion mutuelle utilisant des permissions peut alors être adapté pour résoudre le problème ; un processus ne fait des demandes de permission qu'aux processus qui lui sont voisins dans le graphe des conflits. Il ne commence une transaction que quand il a toutes les ressources nécessaires

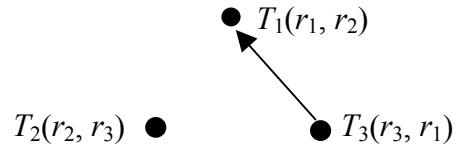
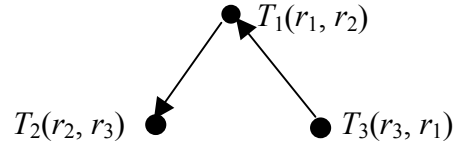
b) Ressources ordonnées

Nous supposons ici que les processus connaissent à l'avance les ressources qu'ils vont utiliser simultanément pour une même transaction.
On impose un ordre total sur les ressources, ordre dont les processus ont connaissance. Les processus demandent les ressources qui leur sont nécessaires selon l'ordre total qui existe sur celles-ci, ne demandant une ressource que lorsqu'ils ont obtenu la ressource précédente. Les ressources se chargent de satisfaire les requêtes en suivant la file d'attente des requêtes qu'elles reçoivent. Il est clair que ce procédé évite l'interblocage ; en effet, les ressources attendues par les processeurs d'un chemin du graphe des attentes sont croissantes le long d'un tel chemin.
On peut remarquer que s'il n'est pas possible que les ressources tiennent à jour les files d'attente centralisées, on peut mettre en œuvre pour chaque ressource un quelconque algorithme réparti pour assurer l'exclusion mutuelle sur chaque ressource ; un processus qui obtient une ressource la garde tant qu'elle n'a pas eu à sa disposition les ressources suivantes qui lui sont nécessaires et terminé sa transaction.

c) Construction du graphe des conflits potentiels

On suppose encore que les processus connaissent les besoins en ressources nécessaires à une même transaction ; il est alors possible d'allouer une ressource à une transaction que si cette allocation ne risque pas de conduire à un interblocage. Pour cela est construit au fur et à mesure *le graphe des conflits potentiels* comme indiqué par l'exemple suivant (à droite de chaque événement est noté l'état résultant du graphe des conflits potentiels) ; supposons que, dans l'ordre chronologique :

Événement	Graphe des conflits potentiels
• une transaction T_1 annonce qu'elle démarre et qu'elle utilisera des ressources r_1 et r_2.	● $T_1(r_1, r_2)$
• une transaction T_2 annonce qu'elle démarre et qu'elle utilisera des ressources r_2 et r_3.	● $T_1(r_1, r_2)$ $T_2(r_2, r_3)$ ●
• une transaction T_3 annonce qu'elle démarre et qu'elle utilisera des ressources r_3 et r_1.	● $T_1(r_1, r_2)$ $T_2(r_2, r_3)$ ● ● $T_3(r_3, r_1)$

• la transaction T_1 fait une requête pour r_1, requête acceptée	 T_1 utilise une ressource dont pourrait avoir besoin T_3 : T_3 dépend de T_1.
• la transaction T_2 fait une requête pour r_2, requête acceptée.	 T_2 utilise une ressource dont pourrait avoir besoin T_1 : T_1 dépend de T_2.
• la transaction T_3 fait une requête pour r_3	Requête refusée

Si T_3 était autorisée à utiliser r_3, il apparaîtrait un circuit dans le graphe des conflits potentiels car alors T_2 dépendrait de T_3 : l'allocation de r_3 est donc refusée, bien que cette ressource soit disponible.

Soit le contrôle est centralisé : un processus allocateur reçoit toutes les informations, construit le graphe des conflits potentiels et gère l'allocation des ressources. Soit l'algorithme est distribué et chaque site gère une copie du graphe des conflits ; des mécanismes adéquats doivent être mis en place pour gérer la cohérence des différentes copies et assurer ainsi que tous les sites prennent les mêmes décisions ; cette gestion peut s'effectuer par la gestion d'une file d'attente pour la modification de « l'original » du graphe des conflits.

d) Gestion locale des conflit potentiels

La gestion des copies de l'état global des transactions et de l'utilisation des ressources est lourde. Dans le cas où le choix n'est pas celui d'un allocateur unique de ressources, chaque site peut gérer uniquement les conflits dans lesquels il risque d'être impliqué. Pour cela, on utilise un mécanisme d'estampillage pour établir une relation d'ordre total entre les annonces.
Pour éliminer le risque d'interblocage, la solution suivante peut être retenue : chaque transaction ne peut se voir allouer une ressource libre que si celle-ci ne fait pas partie de l'annonce d'une transaction plus ancienne non terminée. Reprenons l'exemple ci-dessus ; on suppose que l'estampillage fait que l'annonce de T_1 précède l'annonce de T_2 qui précède l'annonce de T_3 ; tant que T_1 n'est pas terminée, T_2 se voit refuser r_2 qui fait partie de l'annonce de T_1 et se met en attente ; T_3 se voit refuser r_1 et r_3 ; les trois transactions s'exécutent en fait ici les unes après les autres.
Cette méthode réduit le nombre de messages par rapport à la construction distribuée du graphe des conflits potentiels car seules les annonces des transactions doivent être gérées et ordonnées ; les informations concernant l'utilisation des ressources ne sont pas utiles.

e) Prévention dynamique

De nombreux algorithmes ne nécessitent pas que les transactions déclarent à l'avance les ressources dont elles auront besoin ; les conflits sont alors gérés au niveau de la ressource ; lorsqu'une ressource est utilisée par une transaction T_1 et demandée par une transaction T_2, la

décision à prendre est entre deux possibilités : T_2 n'obtient pas la ressource et T_1 continue ou bien T_1 rend la ressource pour la donner à T_2 et dans ce cas T_1 est annulée; dans ce second cas, on dit qu'il y a préemption. Lorsqu'une transaction n'obtient pas la ressource, elle peut être mise en attente (elle garde les ressources dont elle dispose déjà) ou bien annulée (elle rend les ressources dont elle dispose). Dans le cas où une transaction est annulée, il faut restituer l'état dans lequel se trouvait le processus demandeur avant la transaction.
Dans ces méthodes, on estampille les transactions ; nous notons $E(T)$ l'estampille d'une transaction T. De façon à empêcher un phénomène de famine, lorsqu'une transaction est abandonnée, elle garde la même estampille pour redémarrer plus tard.
Nous donnons ci-dessous trois exemples d'algorithmes ; le premier est sans préemption ; le second est avec préemption ; le troisième est plus sophistiqué de façon à minimiser le nombre des abandons.

Nous supposons qu'une ressource *r* est utilisée par une transaction T_1 et qu'une transaction T_2 demande cette ressource ; les réactions de chacun des algorithmes sont décrites ci-dessous.

Algorithme Wait-Die sans préemption

si $E(T_1) < E(T_2)$ alors annuler T_2 (Die) sinon bloquer T_2 (Wait) (*dans tous les cas T_1 garde la ressource*)

Le long d'un chemin du graphe des attentes, les estampilles des transactions sont croissantes, ce qui interdit les circuits dans ce graphe.

Algorithme Wait-Die avec préemption

si $E(T_1) < E(T_2)$ alors bloquer T_2 (Wait) (*T_1 garde la ressource*) sinon annuler T_1 (Die) (*T_2 obtient la ressource*)

Dans cette version, une transaction n'attend jamais une ressource utilisée par une transaction plus récente. Le long d'un chemin du graphe des attentes, les estampilles des transactions sont décroissantes, ce qui interdit les circuits dans ce graphe.

Algorithme Wound-Wait
L'allocateur d'une ressource doit, pour pouvoir prendre une décision, connaître les transactions qui « attendent » et celles qui sont « blessées » (une transaction est blessée si une transaction plus ancienne est en attente d'une ressource qu'elle utilise). Au départ, aucune transaction n'est blessée.

```
si E(T1) < E(T2)  alors
        si T2 n'est pas blessée, alors bloquer T2
        sinon annuler T2
                (* T1 garde la ressource *)
sinon
        si T1 est bloquée alors annuler T1 (*T2 obtient la ressource*)
        sinon blesser T1 et bloquer T2  (*T1 garde la ressource*)
```

Dans le graphe des attentes, si on a un arc (T, T') avec $E(T) < E(T')$, alors il n'y a pas d'arc (T', T'') avec $E(T') > E(T'')$. Sur un chemin du graphe des attentes, les estampilles des transactions sont d'abord croissantes, puis décroissantes et les extrémités des arcs correspondant à des estampilles décroissantes sont des transactions blessées ; par ailleurs, on n'ajoute pas d'arc d'une transaction blessée à une transaction de plus petite estampille ni d'une transaction à une transaction de plus grande estampille et bloquée. On peut alors vérifier qu'il n'y aura pas de circuit dans le graphe des attentes.

3. Détection de l'interblocage dans l'allocation de ressources

Le principe de détection est unique : déterminer des circuits dans le graphe des attentes. Cette détection peut être faite soit par un seul site, soit par chaque site qui tient à jour une copie du graphe des attentes, soit par une coopération de l'ensemble des sites. Nous allons expliciter plus bas l'algorithme de Mitchell et Merrit qui donne un exemple d'une telle coopération.
Lorsqu'un interblocage est repéré, la solution adoptée est de contraindre un processus participant à l'interblocage à rendre les ressources qu'il utilise ; celui-ci doit alors retrouver l'état qu'il avait avant sa demande de ressource.

Il faut se méfier de la détection de faux interblocages. Prenons l'exemple suivant :

- un processus P_1 « verrouille » une ressource r_1 et attend une ressource r_2
- un processus P_2 « verrouille » une ressource r_2 et attend une ressource r_3
- un processus P_3 quasi-simultanément libère r_3 et demande l'accès à r_1

Si le message de libération de r_3 « traîne » par rapport à celui de la demande d'utilisation de r_1, un interblocage de ces trois processus peut être détecté à tort ; l'interblocage n'aurait réellement lieu que si P_3 ne pouvait libérer r_3 qu'après avoir disposé de r_1. Cette remarque conduit à des algorithmes où on conclut à un interblocage lorsqu'un circuit dans le graphe des attentes existe de façon continue pendant un temps donné (de tels algorithmes sont envisageables lorsqu'on connaît une borne supérieure sur les temps de communication).

L'algorithme de Mitchell et Merrit (1984)

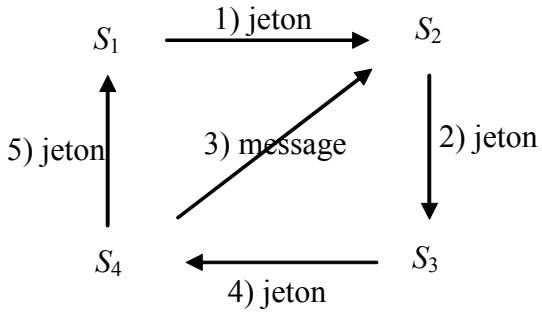
On suppose que les ressources sont demandées une à une : dans le graphe des attentes, tout processus a un degré extérieur de 0 (s'il n'attend rien) ou 1 (s'il attend une ressource).
On suppose que chaque processus P_i a un nom propre qu'on peut noté i et par ailleurs gère un entier p_i initialisé à 0, qui lui sert à enregistrer, lorsqu'il attend, le numéro d'ordre de son attente. Le couple (i, p_i) constitue l'« identificateur propre » de P_i à un instant donné. L'idée est de faire circuler certains identificateurs propres en suivant (à l'envers) les arcs du graphe des attentes pour observer si un identificateur revient à son propriétaire.
Pour faire circuler les identificateurs propres, chaque processus est dépositaire par ailleurs d'un identificateur qui n'est pas forcément le sien appelé l' « identificateur circulant ».

Lorsqu'un processus P_i commence à attendre qu'un autre processus P_j ait libéré une ressource, il pose $p_i \leftarrow p_i + 1$ et change en conséquence son identificateur propre. Par ailleurs, il recopie son identificateur propre dans son identificateur circulant.
Lorsqu'un processus attend, il demande de temps à autre son identificateur circulant à celui qui le fait attendre. Il prend alors comme nouvel identificateur circulant le plus grand entre le sien et celui dont il vient d'être informé.
Si P_i appartient à un circuit du graphe des attentes et a le plus grand identificateur propre de ce circuit, celui-ci finira par faire le tour ; le processus P_i, lors d'une demande à celui qui le fait attendre de son identificateur circulant, retrouvera son identificateur propre et en déduira qu'il y a interblocage. On voit ainsi que l'algorithme a la propriété de vivacité : s'il y a interblocage, celui-ci sera détecté.
Il est un peu moins facile de vérifier que l'algorithme est exact, c'est-à-dire que si un processus conclut à l'interblocage, c'est qu'il y a bien interblocage. Il faut pour cela remarquer que, nécessairement, si P_j a pour identificateur circulant l'identificateur propre d'un processus P_i en attente, alors P_j ne pourra pas être débloqué avant que P_i le soit.

4.2. La terminaison

Le problème de la terminaison est celui de la détection de la fin d'un calcul réparti. Pendant le fonctionnement d'un travail réparti, chaque processus (ou site) est soit dans un état actif s'il est en train d'exécuter du code, soit dans un état passif s'il est en attente ou s'il a terminé. Pour qu'il y ait terminaison, il ne suffit pas que tous les processus soient passifs ; en effet, un message en transit peut relancer le calcul. La terminaison n'est garantie que si à la fois tous les processus sont passifs et aucun message n'est en transit.
La terminaison, comme l'interblocage, est une propriété stable : une fois la propriété vraie, elle restera toujours vraie au cours du temps.

Considérons l'exemple d'un mauvais algorithme qui chercherait à détecter la terminaison d'un système comportant quatre sites S_1, S_2, S_3, S_4 ; chaque site est dans un état *passif* ou *actif* ; l'algorithme consiste à faire circuler un jeton selon une structure d'anneau $S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_1$; lorsqu'un site est passif et possède le jeton, il l'envoie au site suivant ; si le jeton, parti de S_1, revient à S_1, la terminaison est décrétée ; or, supposons que la succession des événements soit la suivante :
1) S_1, passif, envoie le jeton à S_2.
2) S_2, passif, reçoit le jeton et le renvoie à S_3.
3) S_4 envoie un message à S_2 pour lui demander un calcul : S_4 devient passif, en attente.
4) S_3, passif, reçoit le jeton et le renvoie à S_4.
5) S_2, recevant le message de S_4, devient actif.
6) S_4, passif, reçoit le jeton et le renvoie à S_1.
7) S_1 reçoit le jeton et conclut à la terminaison



Différents algorithmes peuvent servir à détecter la terminaison ; le choix de l'algorithme utilisé dépend des hypothèses faites sur les voies et sur le mode (synchrone ou asynchrone) de communication

1. Communications uniquement suivant un anneau unidirectionnel

Nous supposons ici que les communications ont uniquement lieu suivant un anneau unidirectionnel et que les messages ne peuvent pas se doubler ; la détection de la terminaison se fait avec un jeton circulant. On ne peut pas conclure à la terminaison lorsque le jeton fait un tour complet ou même plusieurs tours sans rencontrer de processus actif ; en effet, processus actifs et jetons pourrait se « courir » après autour de l'anneau à la même vitesse ; il faut donc prévoir un mécanisme supplémentaire. La terminaison est détectée quand le jeton a pu faire un tour complet, partant d'un site quelconque *S* et revenant au site *S*, de telle sorte que :

- le jeton n'a rencontré que des processus passifs
- *S* est resté passif durant le tour.

Pour savoir si un processus est resté passif entre deux passages du jeton, on attribue une « couleur », noire ou blanche, aux processus ; un processus activé devient (s'il ne l'est déjà) blanc ; le jeton, s'il trouve le processus blanc et inactif, le met à noir ; seul le jeton peut changer de blanc à noir la couleur d'un processus ; le jeton est muni d'un compteur qui enregistre le nombre de processus consécutifs trouvés à noir (le compteur est remis à zéro lorsque le jeton rencontre un processus actif) ; lorsque ce nombre devient égal à *n*, où *n* est le nombre de processus, si le jeton arrive sur un site passif, il y a terminaison.
On appelle début d'un traitement le tout début du processus ou bien l'arrivée d'un message alors que le processus est passif, message qui « réveille » donc le processus. L'algorithme est le suivant :

Nous donnons deux versions de l'algorithme.
Le jeton est envoyé lorsqu'on veut détecter la terminaison au départ à un site quelconque.

.

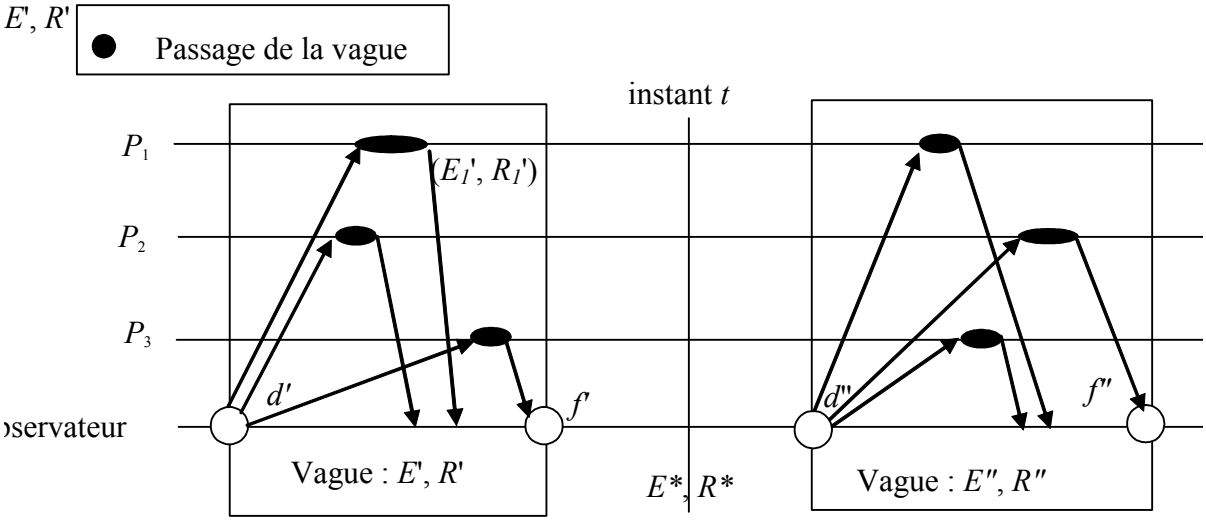
```
texte exécuté lors du début d'un traitement
    état ← actif ;

texte exécuté lors de la fin d'un traitement
    état ← passif ;
     si (jeton_présent)
            envoyer jeton(1) ;
            couleur ← noir
            jeton_present ← faux ;
```

```
texte exécuté lors de la réception de (jeton, j)
    si état = actif, jeton_présent ← vrai ;
    sinon   si (couleur = noir)
                    si ( j = n) terminaison détectée ;
                    sinon envoyer jeton(j + 1) ;
            sinon    couleur ← noir ;
                     envoyer_jeton(0) ;
```

Si les communications se font suivant une topologie plus complexe qui reste fortement connexe, et si l'on suppose toujours l'absence de déséquencement, l'algorithme peut être modifié comme suit ; on définit un circuit passant au moins une fois par chaque voie de communication ; chaque site connaît son successeur sur le circuit ; le jeton, lorsqu'il est envoyé, est envoyé à son successeur ; le nombre n de l'algorithme précédent est remplacé par la longueur du circuit ; toute réception de message autre que le jeton active le site récepteur. L'algorithme permet alors de détecter que tout site est passif et qu'aucun message n'est en transit.

2. Premier algorithme de Mattern

On utilise un observateur qui va essayer de conclure qu'à un moment donné aucun message n'est en circulation et tout processus est passif.
Un observateur envoie sur chacun des sites P_i une requête pour que ceux-ci indiquent les nombres E_i et R_i des messages respectivement reçus et envoyés depuis le début et attend les résultats. On dit ainsi qu'il a effectué une vague d'observation.



L'observateur calcule alors :

$$E = \sum_{i=1}^{n} E_i \, , \, R = \sum_{i=1}^{n} R_i$$

Si les estimations des nombres E_i et R_i avaient été effectuées exactement au même moment physique, l'observateur pourrait conclure, en cas d'égalité, qu'aucun message n'était en transit à cet instant. Mais ce n'est pas le cas.
En conséquence, l'observateur fait deux vagues d'observation, ne lançant la seconde vague que lorsqu'il a obtenu tous les résultats de la première. On note E' et R' les résultats de la première vague, E'' et R'' les résultats de la seconde vague. On note t un instant entre la fin de la première vague et le début de la seconde. On note enfin E^* et R^* les nombres exacts de messages qui ont été reçus et envoyés avant t.

On a : $R' \leq R^* \leq E^* \leq E''$.

Si l'observateur constate $E'' = R'$, il peut alors conclure que $R^* = E^*$ et donc qu'aucun message n'était en transit à l'instant t.

3. Second algorithme de Mattern pour la terminaison (1991)

Dans la version explicitée ci-dessous, un seul site est autorisée à lancer un calcul de terminaison, ce site est nommé *r*. Ce site *r* ne lance le calcul que s'il est passif.
On fait l'hypothèse que les lignes sont fifos.
Des message nommés ci-dessous *information_d'état* peuvent être envoyés d'un site à son père et pourront n'avoir que deux valeurs : passif ou actif.
Un site lance un parcours pour tester la terminaison en envoyant un message *détection_de_terminaison* à chacun de ses voisins ; chaque site atteint par le parcours (recevant un premier message *détection_de_terminaison*) décide que celui qui lui a envoyé ce message est son père et envoie un message *détection_de_terminaison* à chacun de ses voisins sauf à son père. Si un site est actif à la première réception d'un message *détection_de_terminaison*, son *état* (variable du site) est initialisé à actif, sinon son *état* est initialisé à passif. Un site qui apprend, par un message *information_d'état*, qu'un de ses descendants est actif ou bien qui reçoit un message autre que *détection_de_terminaison* ou *information_d'état* (c'est-à-dire un message qui le rend actif) alors qu'il a déjà été touché par le parcours pose qu'il est aussi actif. Dès qu'un site ayant déjà été touché par le parcours est actif, si ce n'est pas le site *r*, il informe son père de cet état et sinon, il conclut à la non terminaison ; il ne fera plus rien (relativement à la détection de la terminaison) à la réception des messages ultérieurs. Lorsqu'un site passif a reçu de tous ses voisins autres que son père soit un message *détection_de_terminaison* soit un message *information_d'état* portant la valeur passif (message qui vient alors d'un de ses fils dans l'arborescence du parcours), si ce site n'est pas *r*, il envoie un message *information_d'état*(passif) à son père, sinon il conclut à la terminaison.
Quand le site *r* conclut à la terminaison, il en informe tous les autres sites.
Les variables utilisées sont :

- *calcul_ commencé* qui peut prendre les valeurs vrai ou faux :
- *état* qui peut prendre les valeurs passif ou actif ;
- *nb_acq* qui est une variable entière ;
- *père* qui sera un nom de site ;

Sur le site i :

Décision de détecter la terminaison (* uniquement pour le site r *)
envoyer à tous ses voisins un message *détection_de_terminaison*(*i*) ;
nb_acq ← nombre de voisins – 1 ;
calcul_ commencé ← vrai ;

Réception d'un message détection_de_terminaison (j)
si (*non calcul_ commencé*)
père ← *j* ;
calcul_ commencé ← vrai ;
envoyer à tous ses voisins sauf à *père* un message *détection_de_terminaison*(*i*) ;
si le site est actif,
état ← actif ;
envoyer *information_d'état* (actif) à *père* ;

```
                sinon
                        état ← passif ;
                        nb_acq ← nombre de voisins – 1 ;
        sinon si (état = passif)
                nb_acq ← nb_acq – 1 ;
                si (nb_acq = 0)
                        si (i ≠ r) envoyer information_d'état (passif) à père ;
                        sinon conclure à la terminaison ;

Réception de message information_d'état(état_fils) (* vient d'un de ses fils *)
        si (état = passif)
                nb_acq ← nb_acq – 1 ;
                si (état_fils = actif)
                        état ← actif ;
                        si (i ≠ r) envoyer information_d'état (actif) à père ;
                        sinon conclure à la non terminaison ;
                sinon si (nb_acq = 0)
                        si (i ≠ r) envoyer information_d'état (passif) à père ;
                        sinon conclure à la terminaison ;

Réception d'un message autre que information_d'état ou détection_de_terminaison
        si (calcul_ commencé) et (état = passif)
                état ← actif ;
                si (i ≠ r) envoyer information_d'état(actif) à père ;
                sinon conclure à la non terminaison ;
```

On laisse au lecteur la vérification de l'exactitude de cet algorithme.

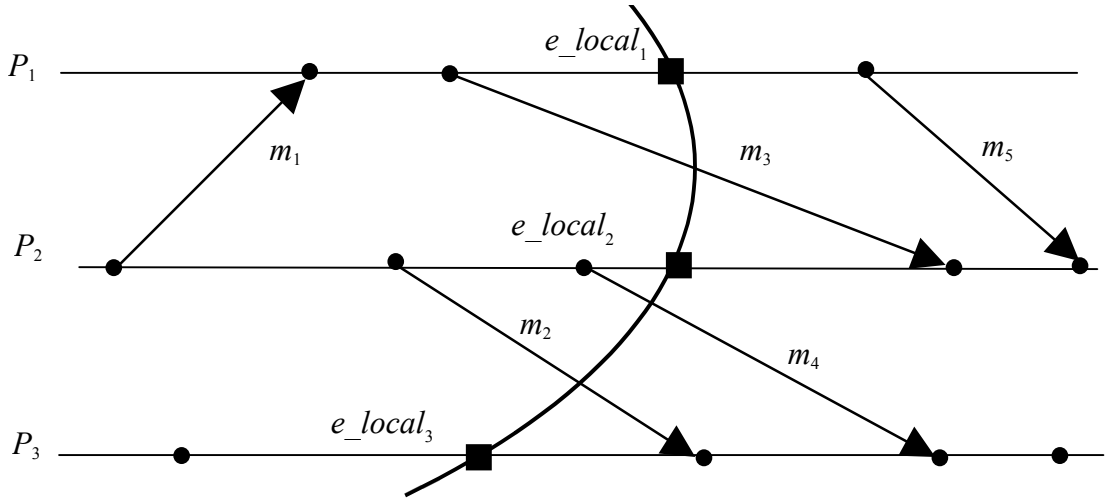
4.3. État global cohérent

Nous allons voir comment calculer ce qu'on appelle état global cohérent du système. Un tel calcul permet de calculer et mémoriser si nécessaire un point de reprise cohérent ; il permet aussi d'analyser la situation globale pour déterminer certaines propriétés.

L'état local d'un site est, à un instant donné, une description exhaustive de l'état du site, ce qui correspond essentiellement à l'ensemble des valeurs de ses variables.
On appelle *état global cohérent du système*, l'ensemble constitué de :

- pour chaque site i : un état local du site i noté e_local_i,
- pour chaque canal (i, j) : l'ensemble des messages de i vers j qui ont été émis de i avant le calcul de e_local_i et reçu en j après le calcul *de* e_local_j,

et tel qu'aucun message d'un site i à un site j n'ait été émis de i après le calcul de e_local_i et ait été reçu en j avant le calcul *de* e_local_j.



Pour l'exemple représenté ci-dessus :
$e\_canal_{1,2} = \{m_3\}$
$e\_canal_{2,3} = \{m_2, m_4\}$
$e\_canal_{1,3} = e\_canal_{2,1} = e\_canal_{3,1} = e\_canal_{3,2} = \varnothing$

1. Algorithme de Chandy et Lamport

Cet algorithme, qui a pour objectif de calculer un état global cohérent d'un système, fait les hypothèses suivantes :

- les canaux de communications sont fifos et fiables ;
- les canaux de communications peuvent être unidirectionnels ;
- le graphe est fortement connexe ;
- il n'y a simultanément qu'un seul calcul d'état global ;

Le calcul de l'état global peut être démarré par un des sites concernés par ce calcul ; les informations peuvent alors circuler à l'aide d'un parcours dynamique du type de ce qui est expliqué dans le chapitre précédent. On donne d'abord une version un peu plus simple : on suppose qu'il y a un site supplémentaire, dit *site collecteur*, qui seul lance le calcul de l'état global par des envois de messages aux différents sites ; ceux-ci envoient les informations concernant ce calcul directement au site collecteur.

Le principe de cet algorithme est fondé sur l'utilisation de messages particuliers appelés *marqueurs* :

- au moment où un site calcule son état local, à la demande du site collecteur, ou bien lorsqu'il reçoit un marqueur d'un de ses voisins, il envoie un marqueur à chacun de ses voisins extérieurs ;
- un site i ayant calculé son état local note pour chacun de ses voisins intérieurs j dans l'état du canal (j, i) tous les messages reçus de j jusqu'à ce qu'il reçoive de celui-ci le marqueur ; il aura alors bien noté tous les messages partis de j avant que celui-ci calcule son état local et arrivé après que lui-même ait fait ce calcul.

Les variables utilisés par le site i sont :

- e_local_i, destiné à contenir un état local du site ;
- Pour tout voisin intérieur j, *reçu_marqueur*(j) ;
- $e\_canal_{j,i}$: liste de messages reçus de j par i ;
- *état_local_noté*: variable booléenne initialisée à faux.

Le pseudo-code des fonctions utilisées par le site P_i est donné ci-dessous.

Départ local du calcul
    $e\_local_i \leftarrow$ résultat du calcul de l'état local ;
    état_ local_ noté ← vrai ;
    Pour tout voisin intérieur *j* :
        $e\_canal_{j,i} \leftarrow \varnothing$;
        reçu_marqueur(*j*) ← faux ;
    Pour tout voisin extérieur *j*, envoyer *marqueur* à *j* ;

Réception de la demande du collecteur de participation au calcul de l'état global
    si (*non état_ local_ noté*), départ local du calcul ; (* fonction ci-dessus *)

Réception d'un marqueur venant de j
    si (*non état_ local_ noté*) départ local du calcul ; (* fonction ci-dessus *)
    reçu _marqueur(*j*) ← vrai ;
    si (pour tout voisin intérieur *j*, *reçu _marqueur*(*j*))
        envoyer au site collecteur
            { e_local_i et, pour tout voisin intérieur *j*, $e\_canal_{j,i}$ } ;

Réception d'un message autre que le marqueur m venant de *j*
    si (*état_local_noté* et non *reçu_marqueur* (*j*))
        $e\_canal_{j,i} \leftarrow e\_canal_{j,i} \cup m$;

On peut adapter l'algorithme Chandy et Lamport de pour que le site collecteur soit un des sites participant au calcul et non un site supplémentaire ; les lignes doivent dans ce cas être bidirectionnelles (au moins ci-dessous en ce qui concerne l'envoi d'un message d'un site à son « père »). On utilise alors un parcours à l'initiative d'un site *r* qui commence par envoyer un marqueur à chacun de ses voisins ;

- un site recevant le marqueur d'un site *j* pour la première fois note son état local, note que *j* est son père dans ce parcours et envoie des marqueurs à tous ses voisins ;
- un site recevant d'un de ses voisins un marqueur alors qu'il a déjà noté son état global note que ce marqueur est arrivé ;
- un site *i* recevant un message d'un site *j* entre le moment où il a calculé son état global et celui où il reçoit le marqueur de *j* note ce message dans l'état de la ligne (*j*, *i*) ;
- après avoir reçu les marqueurs de tous ses voisins, un site *i* fait remonter à son père toutes les informations qu'il a récoltées (son état local et l'état des lignes adjacentes) ; le père fera à son tour remonter ces informations à son propre père jusqu'à ce que les information du site *i* soit arrivées à la racine.

La seule différence avec la version comportant un site collecteur est la façon de diffuser la décision du calcul de l'état global et la façon de récolter en un seul site le résultat du calcul.
Cet algorithme peut facilement être adapté pour détecter la terminaison ; il faut pour que la terminaison soit établie que tous les sites soient passifs quand ils reçoivent la demande du collecteur puis qu'ils ne reçoivent aucun message avant d'avoir reçu tous les marqueurs de ses voisins.

2. Principe de l'algorithme de Mattern (1988)

Cet algorithme a encore pour objectif de calculer un état global cohérent.

On n'utilise plus de marqueurs ; il n'est pas utile de supposer que les lignes sont fifos.
On explicite une version avec un site collecteur ; on pourrait comme pour l'algorithme de Chandy et Lamport utilisé un parcours à partir d'un des sites du système.
Chaque site a une couleur ; il est « noir » avant le calcul de son état local, « rouge » après. Chaque message est marqué de la couleur du site émetteur. Un site noir qui reçoit un message rouge calcule son état local et devient rouge ; il fait cela avant de tenir compte du message qu'il vient de recevoir ; il envoie son état local au site collecteur ainsi que la différence entre le nombre de messages qu'il a émis et le nombre de messages qu'il a reçus depuis le début du processus.
Quand le site collecteur possède tous les états locaux et toutes les différences, il calcule en sommant ces différences le nombre de messages en transit ; un site rouge envoie au fur et à mesure au collecteur tous les messages noirs qu'il reçoit ainsi que le canal correspondant ; le collecteur saura quand le calcul est terminé : ce sera quand il aura reçu un nombre de messages égal à la somme qu'il a calculée.

Bibliographie

- Algorithmique du parallélisme ; le problème de l'exclusion mutuelle (Michel Raynal,1984, Dunod)
- Algorithmes distribués et protocoles (1985, Michel Raynal, Eyrolles)
- Systèmes répartis et réseaux (Michel Raynal, 1987, Eyrolles)
- Synchronisation et état global dans les systèmes répartis (Michel Raynal, 1992, Eyrolles)
- Construction des systèmes d'exploitation répartis (ouvrage collectif, 1991, collection didactique de l'INRIA).