

# Précis de répartition

## Aspects algorithmiques

3ième Année Informatique et Mathématiques Appliquées

11 octobre 2004

### Table des matières

<b>1</b>	<b>L'exclusion mutuelle</b>	<b>2</b>
1.1	Rappel de la spécification du problème . . . . .	2
1.2	L'utilisation d'un jeton circulant . . . . .	2
1.3	Les algorithmes à base de permission . . . . .	3
1.3.1	Exemple d'algorithme à permissions d'arbitre . . . . .	3
1.3.2	Exemple d'algorithme à permissions individuelles . . . . .	4
<b>2</b>	<b>Le problème de la terminaison d'un calcul réparti</b>	<b>5</b>
2.1	La notion de calcul diffusant . . . . .	5
2.2	Spécification du problème . . . . .	5
2.3	Algorithme fondé sur la gestion d'un crédit (Mattern) . . . . .	6
2.4	Algorithme fondé sur la gestion de compteurs (Mattern) . . . . .	7
2.5	Algorithme fondé sur la notion de chemin réparti . . . . .	8
2.5.1	La modélisation du calcul par chemins . . . . .	9
2.5.2	Mise en œuvre de l'algorithme . . . . .	10
<b>3</b>	<b>Interblocage</b>	<b>12</b>
3.1	Rappel . . . . .	12
3.2	Interblocage des communications . . . . .	12
3.3	Un algorithme de détection (Chandy et Misra) . . . . .	13
3.3.1	Modélisation des processus applicatifs . . . . .	13
3.3.2	Déroulement du calcul diffusant . . . . .	13
<b>4</b>	<b>Exercices</b>	<b>14</b>

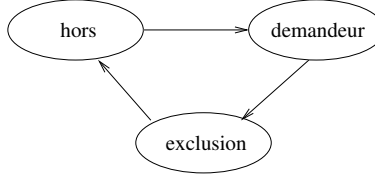


FIG. 1 – Comportement d'un processus

## 1 L'exclusion mutuelle

Le problème de l'exclusion mutuelle doit être revisité dans le contexte d'une architecture répartie. En fait, la répartition apporte de nouvelles stratégies d'implantation.

On peut distinguer plusieurs classes d'algorithmes :

- les algorithmes fondés sur l'unicité d'un jeton circulant matérialisant le privilège d'accès en exclusion mutuelle ;
- les algorithmes fondés sur la notion de permissions d'arbitres ;
- les algorithmes fondés sur la notion de permissions individuelles.

Les variantes (nombreuses) de ces trois classes d'algorithmes reposent en particulier sur des hypothèses différentes concernant la fiabilité des communications. Pour les algorithmes à base de permissions, le défi est de minimiser le nombre de permissions à obtenir pour avoir le droit d'entrer en exclusion mutuelle.

### 1.1 Rappel de la spécification du problème

On considère  $N$  processus (ou sites)  $P_i$  dont le graphe de transitions d'états peut être abstrait vis-à-vis du problème par la figure (1.1). Un processus cycle sur les trois états successifs : *hors*, *demandeur*, *exclusion*.

Les propriétés à vérifier par les  $N$  processus sont les suivantes :

1. Sûreté 1 : exclusion mutuelle

$$\forall i, j :: P_i.exclusion \wedge P_j.exclusion \Rightarrow i = j$$

2. Sûreté 2 : maintien des demandes

$$\forall i, j :: P_i.demandeur \text{ unless } P_i.exclusion$$

3. Vivacité : toute requête est finalement satisfaite

$$\forall i :: P_i.demandeur \rightsquigarrow P_i.exclusion$$

La propriété de vivacité ne peut bien entendu être vérifiée que si tout processus en exclusion finit par sortir, c'est-à-dire si la propriété suivante est vérifiée :

$$\forall i :: P_i.exclusion \rightsquigarrow P_i.hors$$

### 1.2 L'utilisation d'un jeton circulant

Pour obtenir les propriétés précédentes, une solution simple consiste à faire circuler un message "jeton" entre les sites. L'unicité du jeton garantit que, seul, le processus qui possède le jeton peut être en exclusion. Deux problèmes se posent cependant avec cette approche :

- la circulation du jeton entre les processus consomme des ressources même lorsque les processus ne sont pas demandeurs. Pour palier ce défaut, des algorithmes ont été proposés pour bloquer le jeton sur un site s'il n'existe pas de requête en attente. Un algorithme optimal vis-à-vis de la circulation du jeton consiste à guider celui-ci de site en site demandeurs (Algorithme de Naimi Trehel [NT87])

- cette approche repose sur la fiabilité de la communication. En effet, la perte du message jeton empêche tout processus d'entrer en exclusion. C'est pourquoi des algorithmes permettant de réengendrer un jeton ont été proposés. La difficulté principale tient alors à la détection de la perte du jeton (pas de fausse détection) et à l'unicité du jeton engendré (pas de double), ce qui soulève un problème d'élection (un seul processus "élu" doit engendrer un seul jeton).

### 1.3 Les algorithmes à base de permission

Ces algorithmes reposent sur un principe commun : tout processus qui veut entrer en exclusion mutuelle doit obtenir un certain nombre de permissions de la part des autres processus. Pour tout processus  $P_i$ , il faut donc fixer un ensemble  $D_i$  de processus à interroger.

Pour les algorithmes à permissions d'arbitres, chaque processus possède un droit unique qu'il peut prêter (distribuer) à tour de rôle aux processus qui le lui demandent. Autrement dit, il possède une ressource critique (la permission) qu'il peut accorder, allouer à un seul processus demandeur. Une fois prêté, le processus prêteur doit attendre que le processus "emprunteur" lui rende son droit d'arbitrage (sa ressource critique). Une condition nécessaire pour garantir l'exclusion est alors :

$$\forall i, j :: i \neq j, D_j \cap D_i \neq \emptyset$$

Autrement dit,  $P_i$  et  $P_j$  s'adressent au moins à un processus commun qui pourra servir d'arbitre.

Pour les algorithmes à permissions individuelles, chaque processus autorise individuellement les processus qui lui demandent à entrer en exclusion. Il peut donc donner son accord à plusieurs processus (tant que lui-même ne souhaite pas entrer en exclusion par exemple). Une condition nécessaire pour garantir l'exclusion est alors :

$$\forall i, j :: i \neq j, i \in D_j \vee j \in D_i$$

Autrement dit, une interaction aura lieu entre  $P_i$  et  $P_j$  si les deux processus deviennent demandeurs. Sans cette interaction, il ne serait pas possible d'empêcher 2 processus d'entrer en exclusion.

#### 1.3.1 Exemple d'algorithme à permissions d'arbitre

Une stratégie simple consiste à utiliser un UNIQUE arbitre. Tout processus qui demande à entrer en exclusion mutuelle demande à un serveur arbitre via par exemple un appel procédural à distance. Ceci revient à opter pour des ensembles  $D_i$  tous identiques et réduits au processus arbitre :

$$\forall i :: D_i = \{a\}$$

Cette solution est répartie au sens du contrôle (selon le classique schéma client-serveur) mais totalement centralisée du point de vue gestion de l'accès en exclusion mutuelle.

Une possibilité plus répartie est de définir des ensembles multi-arbitres. À titre d'exemple, le tableau suivant décrit le choix fait pour 5 processus s'adressant toujours à 2 arbitres :

<i>i prend pour arbitre :</i>	1	2	3	4	5
1		•	•		
2			•	•	
3		•			•
4		•			•
5		•	•		

Les processus 2, 3 et 4 servent d'arbitres. Si les processus 1 ou 5 s'arrêtent alors qu'ils ne sont pas en exclusion, le système global peut continuer. De même, si le processus 4 s'arrête sans être possesseur de la permission de 2, les processus 1 et 5 peuvent encore utiliser leurs arbitres. On obtient donc une certaine tolérance aux fautes de la solution.

Cependant, certains processus ont un rôle d'arbitre qui les particularise. Il est possible d'obtenir une solution plus symétrique en cherchant à obtenir des ensembles  $D_i$  vérifiant les deux propriétés suivantes :

1. Les ensembles  $D_i$  sont de même cardinalité  $\forall i : \text{Card}(D_i) = K$ , celle-ci étant la plus petite possible. Autrement dit, tous les sites ont le même nombre minimal de permissions à obtenir ;
2. Le nombre d'ensembles  $D_i$  auxquels appartient un site est identique pour tous les sites :

$$\forall i : \langle + j :: i \in D_j : 1 \rangle = P$$

Autrement dit, aucun site ne joue un rôle prépondérant par rapport aux autres.

En fait, le nombre total d'occurrences de sites dans les  $n$  ensembles  $D_i$  est donc  $n * K$  qui doit aussi être égal au nombre  $n * P$ . Par conséquent,  $K = P$  sous ces conditions. On ne peut cependant pas trouver une valeur de  $K$  minimale pour toute valeur de  $n$ .

On peut obtenir une solution simple en utilisant une matrice contenant les numéros de site comme éléments. Pour 9 sites, on obtient par exemple :

$$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array}$$

Un site demande alors aux sites situés sur la même ligne et sur la même colonne. Par exemple, le site 1 aura pour ensemble  $D_1 = \{2, 3, 4, 7\}$ . Pour un nombre de sites qui n'est pas un carré, certains éléments de la matrice restent vides.

En ce qui concerne la vivacité, tout algorithme à arbitrage multiple pose le problème classique des systèmes à ressources critiques. Les processus, dans leur rôle d'arbitre, gère une permission qui est une ressource critique. Obtenir plusieurs permissions revient donc à obtenir l'allocation (le droit d'accès) à plusieurs ressources critiques. Un tel système risque donc l'interblocage. La solution pour éviter l'interblocage est d'introduire un ordre dans les requêtes par un mécanisme d'estampillage.

### 1.3.2 Exemple d'algorithme à permissions individuelles

Avec des permissions individuelles, une façon de garantir simplement la propriété d'exclusion est de forcer un processus demandeur à s'adresser à tous les autres. Autrement dit :

$$\forall i : 0 \leq i < N :: D_i = \{0, 1, \dots, N - 1\} - \{i\}$$

Supposons que la seule condition pour délivrer une permission à un processus demandeur soit le fait que le processus interrogé soit hors exclusion (ni demandeur, ni en exclusion). Alors, une attribution désordonnée des permissions peut conduire à une situation d'interblocage : par exemple, deux processus demandeurs ne peuvent obtenir la permission (réciproque) de l'autre.

Comme pour les arbitres, il faut donc mettre en œuvre une stratégie pour éviter ce risque d'interblocage. Les permissions ne peuvent être attribuées sans respecter un ordonnancement global des requêtes. L'algorithme de Ricart et Agrawala adopte une telle approche en datant les requêtes d'entrée en exclusion à l'aide du mécanisme d'horloge de Lamport. La requête la plus ancienne est alors prioritaire.

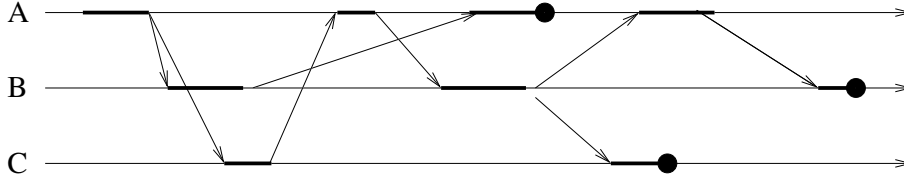


FIG. 2 – Un exemple de calcul diffusant

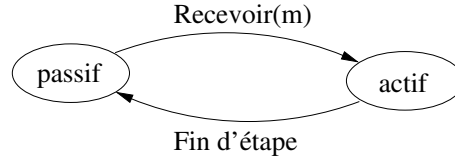


FIG. 3 – Comportement d'un processus

## 2 Le problème de la terminaison d'un calcul réparti

La terminaison d'un calcul dans un contexte réparti est un problème délicat et beaucoup plus complexe qu'en milieu centralisé. Il a été particulièrement étudié sur un modèle de calcul générique appelé calcul diffusant. Ce modèle a donné lieu à de nombreux algorithmes depuis Dijkstra [DS80] jusqu'à Mattern [Mat87, Mat88]. Il a été notamment mis en évidence l'étroite relation entre ce problème et ceux du ramasse-miettes [TM93] ou de la détection d'un interblocage dans un contexte d'exécution répartie [Ray92].

### 2.1 La notion de calcul diffusant

On considère un ensemble de processus ou sites  $\{P_i : 0 \leq i < N\}$  pouvant communiquer par messages de façon asynchrone. Le réseau de communication est supposé fiable mais aucune hypothèse d'ordonnancement des messages n'est nécessaire.

Un processus initial,  $P_0$  par exemple, débute le calcul en envoyant un ou plusieurs messages vers d'autres processus. Puis tous les processus, y compris ce processus initial, adoptent le comportement suivant :

```

loop
    /* un pas de calcul */
    recevoir( $m$ );
    traiter  $m$ ;
    envoyer 0 à  $N - 1$  messages;
end loop

```

Le chronogramme de la figure (2) montre un exemple de calcul diffusant. Un processus commute ainsi de l'état passif en attente de message à l'état actif pour exécuter un pas de calcul qui se terminera par l'envoi éventuel d'un ou plusieurs messages.

### 2.2 Spécification du problème

On considère  $N$  processus (ou sites)  $P_i$  dont le graphe de transitions d'états peut être abstrait vis-à-vis du problème par la figure (2.2). Un processus cycle sur les deux états successifs : *passif*, *actif*. Initialement, tous les processus sont passifs sauf un.

La détection de la terminaison repose sur un prédicat stable : tous les processus sont passifs et il n'existe plus de messages en transit. Si un tel état global est atteint, le calcul est terminé puisqu'aucun message ne

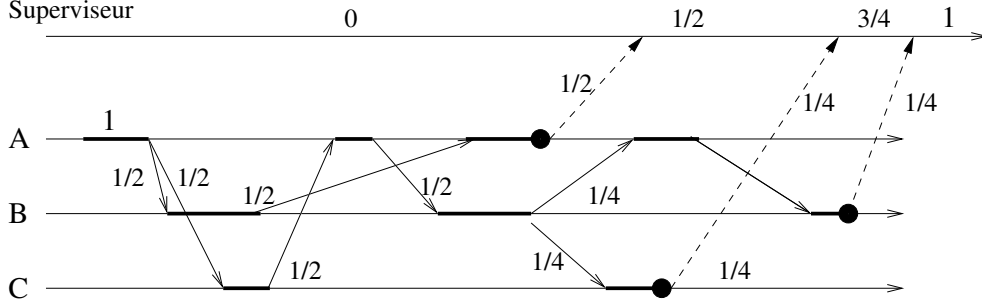


FIG. 4 – Détection par la méthode du crédit

pourra plus réactiver au moins un processus. Il s'agit d'un état stable que l'on peut spécifier sous la forme d'une propriété de sûreté :

$$\text{stable } \forall i :: P_i.\text{passif} \wedge \text{EnTransit} = \emptyset \quad (1)$$

La variable *EnTransit* est une variable abstraite ayant pour valeur l'ensemble des messages en transit à tout instant.

L'objectif est de définir un algorithme qui permette de détecter que l'état stable de terminaison est atteint. On suppose donc un  $(N + 1)$ -ième processus "superviseur" (appelé aussi "collecteur") qui peut interroger les processus pour essayer de déterminer si la terminaison est effective.<sup>1</sup>

On suppose que le processus superviseur gère une variable booléenne *Term*. La correction de l'algorithme exécuté par ce processus repose alors sur les propriétés suivantes :

- Sûreté : pas de fausse détection :

$$Term \Rightarrow (\forall i :: P_i.\text{passif} \wedge \text{EnTransit} = \emptyset)$$

- Vivacité : si le calcul se termine, la terminaison sera finalement détectée :

$$(\forall i :: P_i.\text{passif} \wedge \text{EnTransit} = \emptyset) \rightsquigarrow Term$$

### 2.3 Algorithme fondé sur la gestion d'un crédit (Mattern)

Une idée simple consiste à considérer que le processus "initiateur" possède un crédit qu'il partagera entre les messages initiaux qu'il enverra pour démarrer le calcul diffusant. à titre d'exemple, on peut choisir un crédit égal à 1 et si le processus initiateur envoie 3 messages, il transmettra un crédit égal à  $1/3$  à chacun des processus cibles. Les processus participants (y compris le processus initiateur) adopteront alors le comportement suivant :

- si le processus récepteur envoie un ou plusieurs messages en fin d'étape de calcul, il partagera, comme le processus initiateur, le crédit qu'il a reçu entre les processus cibles de ses messages ;
- si le processus récepteur ne propage pas le calcul, celui-ci enverra le crédit qu'il avait reçu vers le processus superviseur.

Le processus superviseur reçoit donc au cours du déroulement du calcul diffusant, les parcelles du crédit initial transmis par les processus ne propageant pas le calcul. Lorsque la somme de ces crédits atteint le crédit initial (ici 1), le calcul est terminé. La figure 4 montre un exemple avec un crédit initial de valeur 1.

Pour cet algorithme, le prédicat de détection de la terminaison est donc

$$Term \equiv \sum c_i = C$$

avec  $C$  pour valeur du crédit initial et  $c_i$  pour les valeurs de crédits reçus par le superviseur.

La difficulté de mise en œuvre de cet algorithme tient au partage du crédit. Des erreurs d'arrondi doivent absolument être évitées.

<sup>1</sup>On peut aussi supposer que c'est l'un des processus qui interroge les autres sans remettre en question le principe des algorithmes présentés.

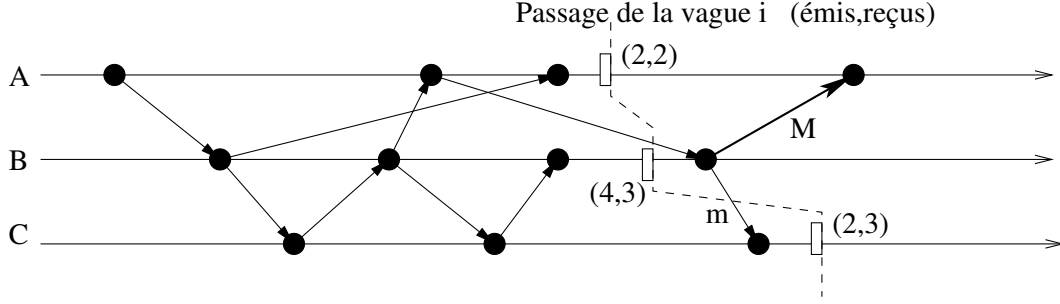


FIG. 5 – Un exemple de fausse détection

## 2.4 Algorithme fondé sur la gestion de compteurs (Mattern)

Une hypothèse simplificatrice peut être faite sur les périodes d'activité des sites : on peut supposer que tout pas de calcul est ininterrompible vis-à-vis de l'arrivée d'un nouveau message. Sur un chronogramme, on peut alors modéliser un pas de calcul par un point celui-ci ayant un caractère atomique. On évite ainsi la gestion explicite d'un état actif/passif des sites. Pour détecter la terminaison d'un calcul diffusant sous cette hypothèse, une idée simple est de compter les événements d'émission et de réception de messages globalement. Si l'on suppose connu à un instant  $t$  où aucun pas de calcul n'est exécuté,  $E(t)$  le nombre d'événements d'émissions et  $R(t)$  le nombre d'événements de réceptions, alors la terminaison du calcul est atteinte ssi  $E(t) = R(t)$  puisqu'il n'y a plus de messages en transit. On a bien :

$$E(t) = R(t) \Rightarrow \text{EnTransit} = \emptyset$$

Néanmoins, la connaissance des compteurs  $E$  et  $R$  n'est pas immédiate. En effet, pour connaître ces derniers, on ne peut que se contenter d'une approximation en allant interroger chaque site. On peut supposer que chaque site  $s$  gère deux compteurs  $E_s$  et  $R_s$  et, que par interrogation, un site observateur peut évaluer d'une part  $\sum_s E_s$  et d'autre part  $\sum_s R_s$ . Le mécanisme de vague peut être utilisé pour cette collecte. On note  $E^i$  la somme des émissions évaluée par la collecte de la  $i$ -ème vague et  $R^i$  la somme des réceptions.

$$E^i = \sum_{s=0}^{s=N-1} E_s^i, R^i = \sum_{s=0}^{s=N-1} R_s^i$$

Le problème de la détection de la terminaison serait alors résolu si l'on avait le prédicat suivant vérifié après une  $i$ -ème vague :

$$E^i = R^i \Rightarrow \forall t \geq f_i : E(t) = R(t)$$

où  $f_i$  est l'instant de fin de la  $i$ -ème vague.

Malheureusement, la figure (5) montre que ce n'est pas le cas. Le passage de la vague aux instants précisés sur le chronogramme conduit à une fausse détection. L'égalité entre les deux sommes est vraie, mais le calcul n'est pas terminé car un message en transit existe encore ( $M$ ). L'anomalie tient à la prise en compte d'un événement de réception d'un message ( $m$ ) dont l'événement d'émission n'a pas été comptabilisé. Nous reviendrons sur ce genre d'anomalie avec la notion de coupure cohérente.

Par conséquent, il faut trouver un autre prédicat en partie gauche de l'implication. En fait, il suffit d'utiliser deux vagues successives. Un prédicat suffisant est alors :

$$\text{Term} \equiv E^{i+1} = R^i$$

En effet, si le nombre de messages émis collecté par la vague  $i + 1$  est égal au nombre de messages reçus collecté par la vague précédente  $i$ , alors il existe un instant  $t$  tel que  $E(t) = R(t)$ . Cet instant  $t$  est au plus tard l'instant de fin de la vague  $i$ . Autrement dit, le calcul était terminé avant le début de la vague  $i + 1$ .

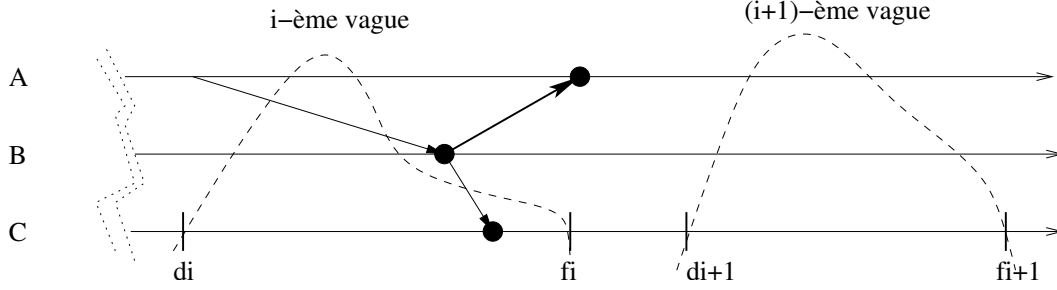


FIG. 6 – Vagues séquentielles

La preuve de la validité du prédicat peut s'appuyer sur les instants de début  $d_i, d_{i+1}$  et de fin  $f_i, f_{i+1}$  des vagues correspondantes, ces instants étant ordonnés dans le temps :  $d_i < f_i < d_{i+1} < f_{i+1}$  comme sur la figure (6).

- Au moment  $f_i$  de la fin de la vague  $i$ , comme à tout autre instant du calcul, le nombre de messages reçus ne peut être qu'inférieur ou égal aux nombre de messages émis, autrement dit :

$$\forall t : R(t) < E(t) \Rightarrow R(f_i) \leq E(f_i)$$

- Mais, le nombre de messages  $E(f_i)$  émis jusqu'à l'instant  $f_i$  est lui, inférieur ou égal à la somme  $E^{i+1}$  résultat de la collecte de la vague  $(i+1)$  et de manière analogue, la somme  $R_i$  résultat de la collecte de la vague  $i$  est inférieure ou égale au nombre de messages reçus à l'instant de la fin de cette vague  $i$ . Si le prédicat  $Term$  est vérifié, on obtient l'implication suivante :

$$\frac{E(f_i) \leq E(d_{i+1}), E(d_{i+1}) \leq E^{i+1}, \{ Term \equiv E^{i+1} = R^i \}, R^i \leq R(f_i)}{E(f_i) \leq R(f_i)}$$

Par conséquent, on en déduit que :  $E(f_i) = R(f_i)$ . à l'instant  $f_i$ , on avait donc bien l'égalité du nombre de messages émis et reçus. Cet état est stable, donc le calcul était bien terminé.

## 2.5 Algorithme fondé sur la notion de chemin réparti

On suppose, comme pour l'algorithme précédent, que tout pas de calcul est atomique et que l'envoi de plusieurs messages est une seule opération de diffusion. Nous verrons dans ce qui suit comment ces hypothèses peuvent être interprétées et implantées de différentes manières.

L'observation du calcul constitue l'originalité de l'approche. En effet, le calcul diffusant n'est pas vu comme une suite d'échanges de message, mais comme le déploiement d'un ensemble de chemins dans l'arbre du calcul. C'est cette interprétation (abstraction) qui va permettre de détecter la terminaison. En effet, au cours de l'exécution du calcul, de nouveaux chemins apparaissent et disparaissent. C'est leur comptage qui permettra de dire si le calcul est terminé ou non au lieu de compter les messages.

Plus précisément, l'algorithme s'appuie sur le comptage des chemins maximaux issus de la racine du calcul. Un chemin maximal est donc ici défini comme une suite d'arcs adjacents conduisant du nœud racine à une feuille. Dans ce qui suit, nous désignons sous le terme de chemins exclusivement des chemins maximaux issus de la racine.

Ainsi, un calcul diffusant apparaît plutôt sous la forme d'un calcul arborescent comme la figure (7) en donne une idée.

De façon classique, on suppose qu'un processus "collecteur" recevra les informations qui lui seront nécessaires pour détecter la terminaison du calcul.

Quelques remarques importantes avant d'aborder l'algorithme :

- l'algorithme utilise comme mécanisme de base la notion de vecteur de chemin. Tous les messages échangés sont surchargés par un tel vecteur de taille égale au nombre de processus ;

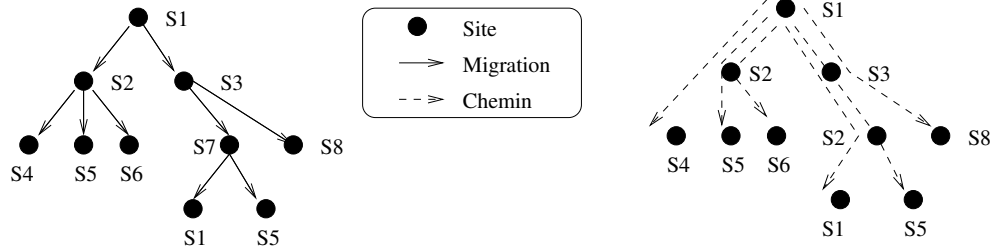


FIG. 7 – Une vision par chemins du calcul

- le simple passage d’un chemin via un processus n’implique aucune action dans l’algorithme. Autrement dit seules les actions locales entraînant une destruction de chemin ou le début d’un ou plusieurs nouveaux chemins entrent en jeu ;
- l’algorithme retarde la connaissance par le processus collecteur de la création de nouveaux chemins jusqu’au moment où un chemin disparaît. C’est en effet à ce moment qu’il faut savoir quel est le passé causal du chemin qui disparaît ;
- l’algorithme n’engendre pas de messages supplémentaires excepté un message par événement “ fin de chemin ” envoyé au processus collecteur. Ce principe de base de l’algorithme est l’utilisation classique de la surcharge des messages de l’application par un seul vecteur de taille fixe égale au nombre de processus ;
- si le réseau perd des messages, l’algorithme reste sûr (pas de fausse détection puisqu’aucune contrainte d’ordonnancement des messages n’est imposée) mais non vivace.

Le processus collecteur, grâce aux informations qu’il recevra au cours du calcul, pourra finalement déduire que le calcul est terminé.

Les propriétés les plus intéressantes de l’algorithme sont les suivantes :

- l’algorithme ne nécessite aucune propriété d’ordonnancement des messages par le réseau (les canaux peuvent être non FIFO par exemple). Il faut cependant que les communications soient fiables ;
- l’algorithme ne nécessite qu’un seul compteur local à chaque processus ;
- l’algorithme détecte la terminaison dès que le site collecteur a reçu tous les messages de “ fin de chemin ”.

En résumé, l’algorithme s’appuie sur les hypothèses de base suivantes :

- le réseau de communication est fiable et connexe ; la communication est asynchrone ;
- un processus collecteur peut recevoir des messages de tous les autres processus ;
- toute phase d’activité d’un processus est considérée comme atomique y compris l’envoi d’un ou plusieurs messages en fin d’étape ;

### 2.5.1 La modélisation du calcul par chemins

L’algorithme repose sur l’utilisation de vecteurs de chemins qui permettent, comme leur nom l’indique, d’évaluer le nombre de chemins en cours de déploiement.

Lorsqu’un message arrive à destination, il appartient toujours à un chemin et 3 cas distincts d’actions locales peuvent se produire :

1. **Follow** : l’action locale traite le message entrant et en réponse poursuit le calcul en envoyant un unique message vers un autre processus : dans ce cas, on peut considérer que le chemin entrant se poursuit ;
2. **End** : l’action locale traite le message entrant et n’envoie aucun message. Ce cas correspond à la fin d’un chemin ;
3. **Split** : l’action locale traite le message entrant et envoie  $p$  messages ( $p > 1$ ) vers d’autres processus<sup>2</sup>. Ce cas correspond à la création de  $p - 1$  nouveaux chemins. L’algorithme proposé impose que l’envoi de

<sup>2</sup>éventuellement, le processus peut s’envoyer un message à lui-même, ce qui peut être interprété comme une continuation du même processus.

ces  $p$  messages soit vu comme atomique c'est-à-dire que le protocole de terminaison doit savoir combien de messages seront envoyés avant l'envoi du premier de ces  $p$  messages. Ceci peut être assuré en gardant les messages dans un tampon associé à l'action locale émettrice tant que celle-ci n'a pas explicitement dit qu'elle se terminait. Dans le système Chorus, cette approche était adoptée pour assurer l'atomicité des étapes d'un acteur [ZBC<sup>+</sup>81].

La détection de la terminaison repose sur le comptage des chemins. Si l'on appelle  $C$  le compteur des chemins créés et  $T$  le compteur des chemins terminés, le prédicat de détection est simplement  $C = T$ . Le problème consiste donc à réussir à évaluer ce prédicat de façon répartie sans provoquer de fausse détection.

Pour le compteur de terminaison  $T$ , chaque action de type **End** se termine par l'envoi d'un message vers le processus collecteur. Ce message informe le processus collecteur qu'un chemin est terminé et qu'il faut donc incrémenter de 1 le compteur  $T$ . Nous allons voir que ce message doit contenir aussi un vecteur de chemins.

### 2.5.2 Mise en œuvre de l'algorithme

Pour compter les créations de chemins, le mécanisme de vecteur de chemins est mis en œuvre. La gestion de ces vecteurs d'entiers repose sur les principes suivants :

- d'une part, on associe un compteur local  $C_i$  à chaque processus. Celui-ci enregistre le nombre de chemins créés via ce processus. Initialement, ces compteurs sont nuls.
- d'autre part, un vecteur de chemins va surcharger tous les messages du calcul. Ce vecteur est transporté par les messages successifs qui construisent peu à peu le chemin et sa taille est égale au nombre  $N$  de processus. Soit  $V$  le vecteur associé à un chemin, ce vecteur va être modifié au fur et à mesure du déploiement du chemin. Il enregistrera le nombre de créations de nouveaux chemins issus de chaque processus et causalement antérieurs au message qui le contient. Un processus unique initialise le calcul en exécutant une action **Split** avec un vecteur initial nul.

#### Algorithme des processus participants

Soit  $V$  le vecteur du message reçu par un processus  $P_i$ . Les actions de mise à jour de ce vecteur sont les suivantes :

- Action de type **Follow** : le vecteur  $V$  est simplement recopié dans le message sortant ;
- Action de type **End** : le vecteur  $V$  est envoyé au processus collecteur ; il signale ainsi au processus collecteur qu'un chemin s'est terminé et qu'un ensemble de chemins existent. On peut remarquer que cette action n'est rien d'autre qu'une opération **Follow** avec le processus collecteur comme destinataire du message émis ;
- Action de type **Split** : le compteur local est mis à jour selon le nombre de chemins créés par l'action :  $C_i := C_i + (p - 1)$ . Après quoi, la composante  $i$  du vecteur  $V$  reçoit cette valeur :  $V[i] := C_i$ . Ce vecteur mis à jour est le vecteur qui sera placé dans *tous* les messages sortants. En effet, tous les chemins créés héritent de la même causalité issue de celle qui existait déjà (chemin "origine"). Tout vecteur  $V$  sortant d'un processus  $P_i$  connaît donc exactement le nombre de chemins créés par le processus  $P_i$ .

**Remarque** L'action **Follow** pourrait être considérée comme une action **Split** de degré 1. Cependant, une telle action **Split** entraîne une affectation ( $V[i] := C_i$ ) non nécessaire bien que tout à fait compatible avec l'algorithme.

#### Algorithme du processus collecteur

Grâce aux vecteurs qu'il reçoit, le processus collecteur évalue peu à peu le nombre de chemins créés et détruits. Il gère un vecteur  $MT$  initialement nul et le compteur  $T$  lui aussi initialement nul.

Le processus collecteur reçoit les vecteurs de chemins qui lui sont adressés jusqu'à ce qu'il puisse conclure à la terminaison. Pour chaque vecteur  $V$  reçu par le collecteur, celui-ci :

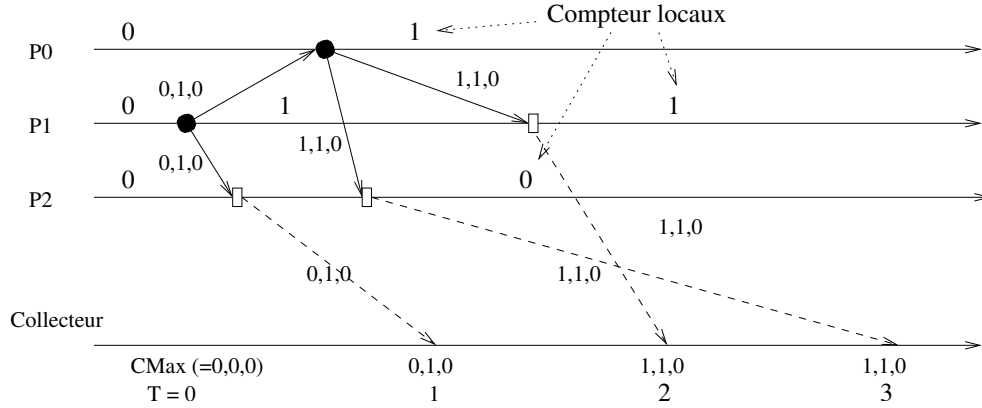


FIG. 8 – Exemple de détection

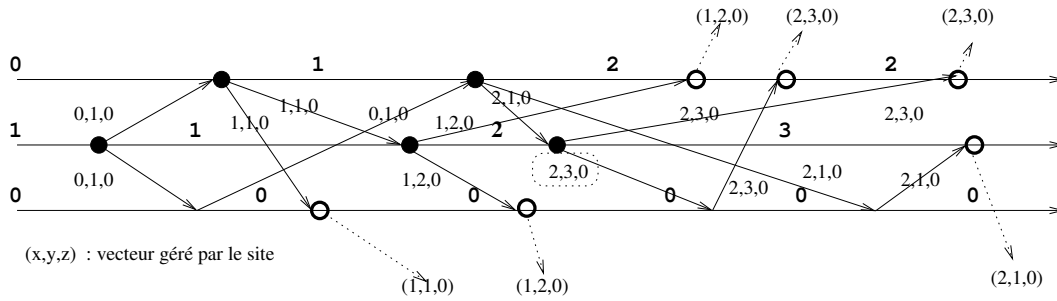


FIG. 9 – Exemple plus complet

- évalue le nouveau vecteur maximum  $MT$  à l'aide de la fonction  $max$  définie par :

$$\forall i :: max(A, B)[i] = \text{if } (A[i] < B[i]) \text{ then } B[i] \text{ else } A[i]$$

- enregistre la terminaison d'un chemin (incrémement de  $T$ )
- et teste le prédicat de terminaison en comparant la somme des éléments du vecteur  $MT$ <sup>3</sup> avec le nombre de chemins terminés indiqué par la variable  $T$ .

```

process Collecteur {
  integer MT[N] = [0, ..., 0]; /* vecteur maximum */
  integer T = 0; /* compteur des chemins terminés */
  repeat
    integer V[N];
    recevoir(V);
    MT, T := max(MT, V), T + 1; /* Term */
  until (Σ MT + 1 = T) /* Detect */
}

```

Le chronogramme de la figure (8) montre l'évolution du calcul sur un exemple simple.

L'exemple de la figure (9) présente un autre cas plus complet montrant l'importance du compteur local à chaque processus. Le processus collecteur n'est pas représenté. On notera que les messages envoyés vers le collecteur peuvent être reçus dans un ordre quelconque.

<sup>3</sup>On utilise la notation  $\Sigma X$  pour abrégier :  $\Sigma X = \langle + : 0 \leq k < N :: X[k] \rangle$ .

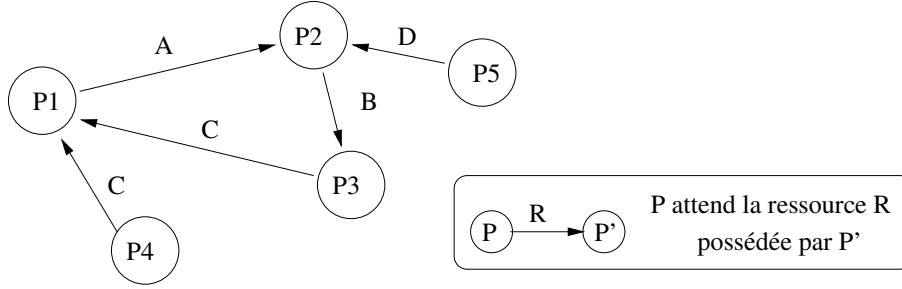


FIG. 10 – Cycle d’interblocage dans le modèle *et*

### 3 Interblocage

Le problème de l’interblocage ne disparaît naturellement pas avec la répartition. Tout schéma d’allocation de ressources critiques réparties conduit à un système parallèle sujet au risque d’interblocage. Pire, la communication elle-même peut être à l’origine de situations d’interblocage sous certaines conditions.

#### 3.1 Rappel

Rappelons qu’en milieu centralisé, l’allocation de ressources critiques peut prendre deux formes différentes :

- **Modèle *et*** : les processus acquièrent des ressources identifiées. Autrement dit, toute requête a pour objectif d’obtenir une ressource critique spécifique (éventuellement plusieurs simultanément, mais il faudra alors les obtenir toutes) ;
- **Modèle *ou*** : les processus acquièrent des ressources prises parmi des classes de ressources. Autrement dit, une requête demande l’obtention d’un exemplaire quelconque d’une ressource d’une certaine classe.

Dans le modèle de type *et*, un état d’interblocage est caractérisé par la détection d’un cycle dans le graphe d’attente des processus. La figure (10) montre un tel graphe dans lequel les processus  $P_1$ ,  $P_2$  et  $P_3$  sont en état d’interblocage.

Dans le cas du modèle de type *ou*, la détection d’un interblocage repose sur la formation d’une composante fortement connexe terminale (en abrégé CFCT ou knot en anglais). Un graphe  $G = (S, A)$  où  $S$  est l’ensemble des sommets et  $A$  est l’ensemble des arcs est un graphe CFCT ssi il satisfait la propriété suivante :

$$\forall s \in S : \text{succ}(s) \neq \emptyset \wedge \text{succ}(s) \subseteq S$$

Autrement dit, dans un tel graphe, tout sommet possède au moins un arc sortant et tout arc sortant conduit à un sommet qui appartient aussi au graphe.

La figure (11) illustre un tel graphe entre 5 processus. Le processus  $P_2$  (resp.  $P_5$ ) attend une ressource de classe  $B$  (resp.  $E$ ) sachant que les exemplaires de cette classe de ressources sont possédés par  $P_3$  et  $P_5$  (resp.  $P_1$  et  $P_4$ ).

#### 3.2 Interblocage des communications

Les échanges de messages peuvent conduire à des situations d’interblocage similaires au modèle *ou* précédemment décrit. En effet, considérons un système de processus communiquant par messages en mode synchrone : l’émission d’un message est bloquante tant que le message n’a pas été reçu.

Alors, les processus émetteurs possibles apparaissent comme des ressources critiques demandées par les récepteurs. Dans la figure (11), on peut donner ainsi à chaque arc la signification suivante : un arc existe de  $P$  vers  $P'$  si  $P$  attend un message issu de  $P'$ . On remarquera que cette interprétation implique de connaître les émetteurs qui peuvent envoyer un message vers le récepteur <sup>4</sup>.

<sup>4</sup>Au pire, et par défaut, il faudra considérer que tout autre processus peut être émetteur.

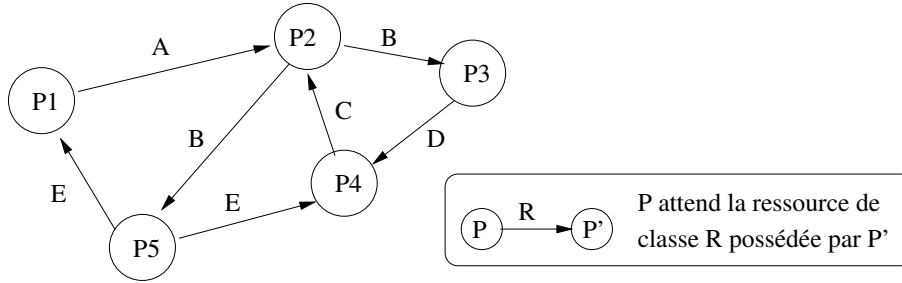


FIG. 11 – Graphe CFCT d’interblocage dans le modèle **ou**

Compte tenu des arcs existants, les processus de la figure (11) sont tous dans l’état  $\text{ij}$  en attente de réception  $\text{ij}$ . à titre d’exemple, le processus  $P_5$  peut être considéré comme en attente d’un message issu du processus  $P_1$  ou  $P_4$ .

Une communication en mode asynchrone peut aussi conduire à une situation d’interblocage mais il faut alors vérifier une condition supplémentaire : les canaux de communication sous-jacents aux arcs du graphe sont vides.

### 3.3 Un algorithme de détection (Chandy et Misra)

Un algorithme de détection a été proposé par Chandy et Misra sous la forme d’un calcul diffusant. Cet algorithme permet à un processus  $\text{ij}$  enquêteur  $\text{ij}$  (qui peut être l’un des processus participants) de savoir s’il appartient à une CFCT. Tant qu’il n’obtient pas de réponse, le doute subsiste. Attention, cet algorithme ne permet pas de savoir si une CFCT existe quelque part parmi les processus applicatifs.

#### 3.3.1 Modélisation des processus applicatifs

Chaque processus commute, au cours du calcul, de l’état actif à l’état passif lorsqu’il se met en attente de réception d’un message. La réception du message provoquera le retour à l’état actif.

Dans l’état passif, tout processus  $i$  possède un ensemble de dépendance  $D_i$  non vide constitué des processus dont il peut espérer un message.

#### 3.3.2 Déroulement du calcul diffusant

Le calcul diffusant, démarré par le processus enquêteur, est propagé par les processus visités dans l’état passif. Un processus visité alors qu’il est actif ignore l’enquête.

La propagation du calcul par un processus passif  $i$  se fait vers les sites appartenant à l’ensemble de dépendance correspondant  $D_i$ .

Le calcul diffusant parcourt ainsi les arcs du graphe d’attente. Pour détecter une CFCT, il faut détecter les cycles présents dans le graphe. Autrement-dit, dès qu’un processus est revisité par le calcul diffusant alors qu’il est toujours passif, un cycle a été parcouru (et détecté).

Pour éviter les fausses détections et assurer la terminaison du calcul diffusant (SI une CFCT incluant le processus enquêteur existe bien), au cours de son déploiement, le calcul diffusant place les processus visités dans un arbre de recouvrement selon les règles suivantes :

- d’une part, lors de la première visite, un processus passif prend pour père, le processus émetteur ;
- d’autre part, la propagation de la détection des cycles ne se fera d’un fils vers son père que lorsque le fils aura lui-même reçu une réponse de la part de chacun des processus appartenant à son ensemble de dépendance.

Autrement dit, la détection de la CFCT est équivalente à la réussite de la construction de l’arbre **ou**, ce qui est équivalent, à la terminaison du calcul diffusant qui le construit). Si l’arbre se construit ( $\equiv$  le calcul

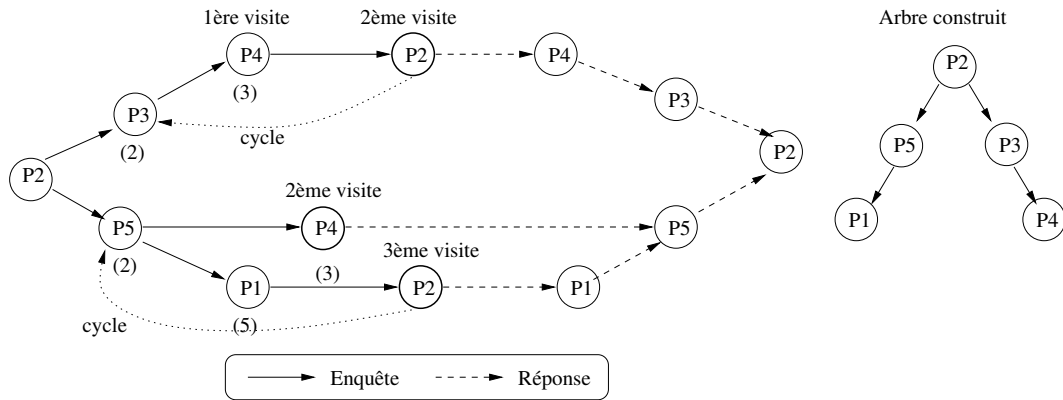


FIG. 12 – Calcul diffusant de détection d'une CFCT

diffusant se termine), il existe une CFCT. Si la construction de l'arbre ne se termine pas ( $\equiv$  le calcul diffusant ne se termine pas), il n'y a pas interblocage incluant le processus enquêteur.

à titre d'exemple, la figure (12) montre le déroulement du calcul diffusant débuté par  $P_2$  et conduisant à la conclusion que ce dernier appartient bien à une CFCT englobant les 5 processus. La détection de la CFCT repose ici sur la détection de 2 cycles.

## 4 Exercices

1. On utilise le principe du jeton circulant pour résoudre le problème de l'exclusion mutuelle. On suppose que le message jeton peut se perdre. Proposer un algorithme de détection de la perte du jeton. On supposera que les canaux de communication sont FIFO ;
2. La détection de la perte du jeton étant faite, il faut engendrer un nouveau jeton. Proposer un algorithme d'élection qui permettra d'assurer qu'un seul processus engendrera un jeton ;
3. L'algorithme de détection d'un interblocage par détection d'une CFCT utilise un calcul diffusant. Par ailleurs, cette détection repose sur la terminaison de la construction d'un arbre de recouvrement. Proposer une adaptation de cet algorithme en utilisant l'algorithme de détection de la terminaison d'un calcul diffusant utilisant les vecteurs de chemins.

## Références

- [DS80] E.W.D. Dijkstra and C.S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(4) :1–4, 1980.
- [Mat87] F. Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2 :161–175, 1987.
- [Mat88] F. Mattern. Virtual time and global states of distributed systems. In *Proceedings of the International Conference on Parallel and Distributed Computing*, pages 215–226. North-Holland Inc., 1988.
- [NT87] M. Naïmi and M. Trehel. Un algorithme distribué d'exclusion mutuelle en  $\log(n)$ . *Technique et Science Informatiques*, 6(2) :141–150, 1987.
- [Ray92] M. Raynal. *Synchronisation et état global dans les systèmes répartis*. Editions Eyrolles, 1992.
- [TM93] G. Tel and F. Mattern. The derivation of distributed termination detection. *ACM Transactions on Programming Languages and Systems*, 15(1) :1–35, 1993.
- [ZBC<sup>+</sup>81] H. Zimmermann, J-S. Banino, A. Caristan, M. Guillemont, and G. Morisset. Basic concepts for the support of distributed systems : the chorus approach. In IEEE, editor, *2nd international conference on distributed computing systems*, pages 60–66, April 1981.