Introduction à l'algorithmique répartie

Denis Conan

CSC4509

Télécom SudParis Mai 2025

Table des matières

Ce cours a bénéficié des relectures de François Meunier, Léon Lim, et Michel Simatic.

2

Utilisation du cours

- Apprentissage en formation en ligne
  - Démarche conseillée :
    - Pour chaque page du support de cours, étudiez la page du support de cours (diapositive + commentaires)
- Étude des exercices en présentiel
- Puis auto-évaluation des connaissances avec les QCM (une série par section)
- Les QCM ainsi que les corrigés des exercices sont fournis à part dans moodle

NB : Certaines pages du cours sont marquées par un astérisque (« * ») à la fin de leur titre. Ceci correspond à un contenu d'approfondissement. Ne l'étudiez pas en détail avant de maîtriser les autres points de la section. Ces diapositives doivent être étudiées avant de faire certaines questions (optionnelles) des exercices.

Plan du document

3

Une définition communément admise d'un système réparti est : « plusieurs ordinateurs inter-connectés par un ensemble de réseaux de communication effectuant un travail ensemble et communicant par échange de messages ». Cette définition suggère deux propriétés principales des systèmes répartis : la non-unicité de lieu (un utilisateur travaille en local ou à distance) et la non-unicité de temps (chaque ordinateur possède sa propre notion du temps à travers son horloge physique). Par ailleurs, cette définition diffère de celle d'un système parallèle (ou dit fortement couplé) dans lequel les communications peuvent s'effectuer via une mémoire partagée et dans lequel il y a unicité de lieu et de temps.

Les systèmes répartis sont difficiles à concevoir et à comprendre parce qu'ils ne sont pas intuitifs. Peut-être est-ce aussi parce que notre vie est, par bien des aspects, fondamentalement séquentielle ? Nous devons donc développer une intuition pour la répartition. Dans cette discipline, il existe une tension inévitable entre les partisans de la modélisation et de l'analyse, et ceux de l'observation expérimentale. Cette tension illustre la dichotomie classique entre la théorie et la pratique. Dans ce cours d'algorithmique répartie, nous nous placerons plus du côté pratique.

Nous commençons par la présentation du *modèle de système réparti* dans la section introductive. Ensuite, les problèmes étudiés sont des problèmes fondamentaux basiques de l'algorithmique répartie à partir desquels sont construites des architectures de services répartis complexes. Le principe de l'*élection* est de partir d'une configuration dans laquelle tous les processus sont dans le même état, pour arriver dans une configuration dans laquelle un seul processus est dans l'état « gagnant » et tous les autres dans l'état « perdant ». La *diffusion* est une primitive de communication permettant à un processus d'envoyer le même message à tous les autres processus en respectant des propriétés d'ordre (FIFO, causal, total) dans la transmission des messages. L'*exclusion mutuelle* consiste à faire circuler un jeton entre des processus répartis sur le réseau pour n'autoriser qu'un seul d'entre eux à entrer en section critique. C'est l'expression répartie des sémaphores. L'*interblocage* se produit lorsqu'un ensemble de processus est tel que chacun d'eux tient au moins une ressource, et pour poursuivre sa progression, est en attente d'une ressource tenue par l'un des autres. Des algorithmes sont proposés pour prévenir ou détecter les interblocages. La *détection de terminaison* autorise qu'un algorithme réparti se termine de façon implicite, c'est-à-dire sans que les processus atteignent leur état final (fin du processus par appel de la fonction `exit`), autrement dit, « parce qu'il n'y a plus de travail à faire » (tous les processus sont en attente d'un message et les canaux de communication sont vides).

1 Éléments introductifs

4

Cette section introductive est reprise des références suivantes :

- G. Tel, *Chapter 2 : The Model*, dans *Introduction to Distributed Algorithms*, Cambridge University Press, pp. 43–72, 1994.
- C. Fidge. *Logical Time in Distributed Computing Systems*, dans *IEEE Computer*, pages 28–33, August 1991.
- G. Tel, *Chapter 6 : Wave and traversal algorithms*, dans *Introduction to Distributed Algorithms*, Cambridge University Press, pp. 177–221, 1994.
- H. Attiya et J. Welch, *Chapter 2 : Basic Algorithms in Message-Passing Systems*, dans *Distributed Computing : Fundamentals, simulation, and advanced topics*, Wiley, pp. 9–30, 2004.
- J.H. Saltzer, M.F. Kaashoek, *Principles of Computer System Design : An Introduction*, Morgan Kaufmann, 2009.
- L. Lamport, *Time, clocks and the ordering of events in a distributed system*, Communications of the ACM, 21(7), July 1978.
- C. Fidge, *Logical Time in Distributed Computing Systems*, IEEE Computer, pages 28–33, August 1991.

Le premier élément d'introduction est le modèle de système réparti. Ce modèle à base de messages est utilisé dans tout le reste du cours. Il est assez général pour être utile aussi bien lors de la conception que lors de la vérification (même si nous ne nous focalisons par sur les preuves des algorithmes étudiés). Puis, nous introduisons les conventions de codage utilisées dans ce cours : soit l'orientation contrôle soit l'orientation évènement. Ensuite, le dernier élément général introduit pour la suite du cours est la notion de dépendance causale qui permet de construire un ordre partiel des évènements d'une exécution répartie. Enfin, parmi les problèmes fondamentaux que nous étudions, beaucoup peuvent s'exprimer à l'aide de sous-tâches génériques comme les vagues. C'est par exemple le cas de la diffusion ou de la détection d'interblocage étudiée un peu plus loin dans ce cours. Nous présentons donc à la fin de cette section le principe des algorithmes de vagues et y ferons référence dans les autres sections.

1.1 Modèle de système réparti

5

Les paramètres du système réparti les plus discriminants sont les suivants : synchronisme (« performance » des nœuds et du réseau), type de fautes des nœuds et des communications (catégorie de fautes matérielles et logicielles [persistantes $\neq$ transitoires]), topologie (forme du graphe du système de communication) et déterminismes des processus (comportement prévisible des algorithmes répartis). Ces hypothèses sont importantes car elles conditionnent les résultats d'impossibilité (comme l'atteinte d'un consensus) ainsi que les approches algorithmiques. Par exemple, il n'existe pas de solution déterministe à tous les problèmes dans tous les cas. Mais, avant de détailler ces éléments discriminants, nous présentons les éléments constitutifs du modèle, c'est-à-dire dans notre cas ceux du modèle de transitions.

Ce cours se limite à l'étude des algorithmes répartis déterministes. Ainsi, nous ne présentons pas d'algorithme avec comportement aléatoire, par exemple ceux du type « Las Vegas » (avec une solution correcte mais une distribution probabiliste sur la durée d'exécution) ou ceux du type « Monte Carlo » (avec une probabilité sur la correction de la solution mais un bornage de cette probabilité).

1.1.1 Modèle de transitions

6

- Algorithme : assigne des valeurs à des variables
- Processus : exécution d'un algorithme sur un nœud du système réparti
- Canal de communication : lien logique réseau entre deux processus
- Exécution = état initial puis succession d'actions sur les états
- État global ou configuration = juxtaposition d'états locaux, un par processus
- Histoire = séquence des actions d'un processus
  - Histoire répartie = ensemble des histoires locales des processus
- Répartition $\Longrightarrow$ actions internes, d'émission et de réception
- Toute propriété est ou se décompose en propriétés de :
  - Sûreté (en anglais, *safety*) : une assertion est vraie dans chaque configuration de l'algorithme
    - Quelque chose de non désiré n'arrive jamais
  - Vivacité ou progression (en anglais, *liveness*) : une assertion est vraie dans certaines configurations de chaque exécution
    - À terme, quelque chose de « bien » arrive

Tous les modèles de spécification pour les systèmes répartis sont basés sur la notion d'action atomique et de machine à états. Parmi les modèles les plus couramment choisis, citons le modèle des transitions, le modèle des actions temporelles logiques, et le modèle des automates (pouvant être temporisés). Dans ce cours, nous utilisons le modèle le plus simple, celui dit des transitions, que nous décrivons maintenant.

Un algorithme assigne des *valeurs* à des *variables*. Un *état* est l'affectation de valeurs à des variables. Une *action* (encore appelée un *évènement*) a représente la relation entre un ancien état s et un nouvel état t notée $s\mathsf{a}t$. Les actions sont atomiques : l'action a provoque le changement « instantané » de l'état de s à t. Autrement dit, on n'observe pas l'état du système pendant l'exécution de a.

Un *algorithme séquentiel* $\mathcal{A}$ est un objet syntaxique construit selon la grammaire d'un langage de programmation. Une *exécution* de l'algorithme séquentiel $\mathcal{A}$ dans un processus P débutant à l'instant t à partir de l'état initial s^0, est la succession d'un nombre infini d'actions sur des états notée $\mathsf{a}^0 s^0 \mathsf{a}^1 s^1 \mathsf{a}^2 s^2 \mathsf{a}^3$, etc. L'état s^0 est appelé l'*état initial*. L'action initiale a^0 est fictive et correspond à la création du processus P. L'exécution d'un algorithme séquentiel produit donc une séquence d'actions. La séquence d'actions est appelée l'*histoire* de l'algorithme séquentiel. De façon duale, lorsqu'il s'agit de formuler ou de démontrer une propriété, il est souvent très intéressant de modéliser une exécution comme étant une séquence d'états, une action étant la transition d'un état à un autre [1,2].

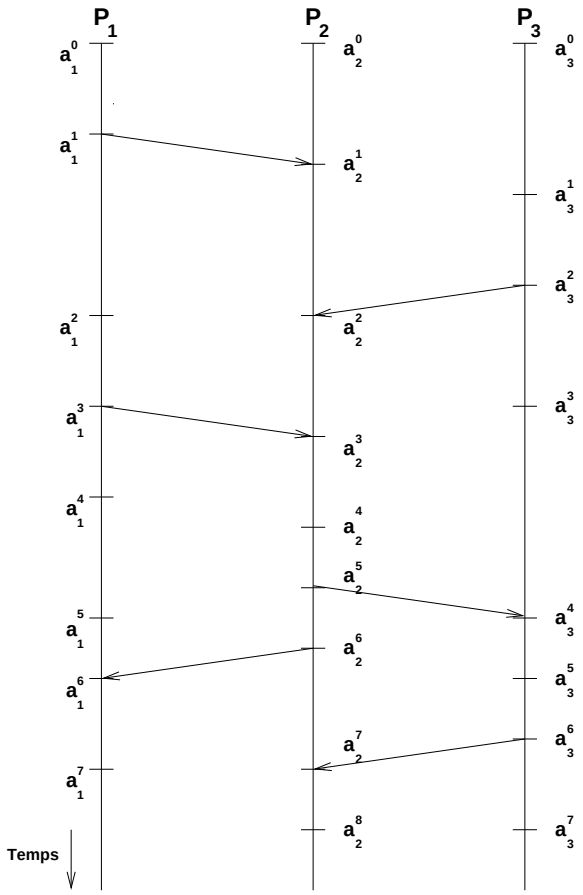
Un *algorithme réparti* $\mathcal{A}$ est composé d'algorithmes séquentiels exécutés dans des processus $P_1 ... P_n$ qui communiquent par échange de messages. Considérant le processus P_i, l'action d'*émission* émettre(P_j, m) d'un message m de P_i vers P_j ajoute m au canal c_{ij}. Pratiquement, m est transmis de P_i vers P_j par le réseau de communication et est gardé dans la mémoire du nœud où s'exécute P_j. L'action de *réception* `recevoir`(m) d'un message m par P_j sur l'ensemble des canaux c_{*j} assigne à m le premier message arrivé par l'un des canaux. Si aucun message n'est arrivé alors P_j attend jusqu'à l'arrivée d'un message par l'un des canaux. Toute action autre qu'une action d'émission ou de réception d'un message est appelée une *action interne*.

L'*état d'un canal* c_{ij} à l'instant physique t est constitué de l'ensemble des messages émis et non encore reçus. L'*état global* (encore appelé *configuration*) s d'un système réparti à l'instant physique t est composé des états locaux de tous les processus du système réparti à l'instant t. L'exécution d'un algorithme réparti débutant à l'instant physique t à partir de la configuration s est constituée de la juxtaposition des exécutions des processus. Par déduction, l'histoire répartie d'un système réparti est constituée de la juxtaposition des histoires des processus.

1. R.W. Floyd. Assigning meanings to programs. In J.T. Schwartz, editor, *Proceedings of Symposia in Applied Mathematics, Mathematical Aspects of Computer Science*, volume 19, pages 19–32, Providence, Rhode Island, USA, 1967. American Mathematical Society.
2. C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, October 1969.

Le modèle que nous venons de définir est appelé le modèle de transitions sous l'hypothèse de *communication asynchrone* : les opérations d'émission ne sont pas bloquantes alors que les opérations de réception le sont. Dans le cas des *communications synchrones*, les opérations d'émission sont elles-aussi bloquantes. Dans ce cas, clairement, dans toutes les configurations dans lesquelles tous les processus viennent d'exécuter une action interne, les canaux sont vides.

L'exécution d'un algorithme réparti est classiquement représentée par un diagramme de séquences (à la UML) aussi appelé diagramme temporel ou chronogramme. La figure qui suit trace un tel diagramme pour un algorithme réparti composé de trois algorithmes séquentiels. Le déroulement du temps est décrit par une ligne continue (ligne qui est à tiret et s'appelle « ligne de vie » en UML) pour chaque algorithme séquentiel. Les actions sont symbolisées par des tirets sur les « lignes de vie ». Les messages sont matérialisés par des flèches connectant une action `émettre` à une action `recevoir`.



Enfin, dans un système réparti, il est important de faire la distinction entre les propriétés de sûreté (en anglais, *safety property*) et de progression ou vivacité (en anglais, *liveness property*). Une propriété de sûreté exprime « ce que le système ***ne doit pas*** faire » ; elle est de la forme « l'assertion est vraie dans chaque configuration de l'algorithme (en considérant une exécution finie) ». Pratiquement, la propriété de sûreté sert à exprimer que quelque chose de non désiré n'arrive pas. Si une propriété de sûreté est violée, elle l'est à un moment donné de l'exécution. La technique de base pour montrer que l'assertion est toujours vraie est de démontrer que c'est un invariant : vrai dans l'état de départ de l'exécution, et si vrai dans la configuration atteignable s alors vrai dans toutes les configurations atteignables directement à partir de s.

Une propriété de vivacité, quant à elle, exprime « ce que le système ***doit*** faire » ; elle est de la forme « l'assertion est vraie à terme ou ultimement (en considérant une exécution infinie) ». Pratiquement, la propriété de vivacité sert à exprimer que quelque chose de désiré arrive ; sans propriété de vivacité, c'est-à-dire uniquement avec la sûreté, on n'exige pas que l'algorithme fasse réellement quoi que ce soit. La technique de base pour montrer que l'assertion est vraie à terme est soit d'utiliser une autre propriété de vivacité (par exemple, le message est reçu à terme par un processus atteignable car tous les canaux de communication du système transmettent *in fine* tous les messages émis par l'émetteur du canal), soit par induction en utilisant une métrique qui progresse dans le temps jusqu'à atteindre un seuil auquel l'assertion est vraie (pour le même

exemple, parmi chaque pas d'exécution considérant l'émission ou la réception d'un message, de temps en temps, il y a un message qui s'approche de récepteur en récepteur du destinataire final, la métrique utilisée étant le nombre [fini] de processus entre l'émetteur initial et le destinataire final).

1.1.2 Synchronisme

7

- Synchrone[a] :
  - ∧ Durée de transmission d'un message d'un nœud à un autre bornée et borne connue
  - ∧ Durée d'éxécution d'une action interne d'un processus bornée et borne connue
- Asynchrone :
  - ∨ Pas de borne ou borne inconnue sur la transmission d'un message
  - ∨ Pas de borne ou borne inconnue sur la dérive des horloges
  - ∨ Pas de borne ou borne inconnue sur la durée d'un traitement

a. Il est important de ne pas confondre synchronisme des communications et synchronisme du système. Ici, c'est le synchronisme du système qui est considéré.

Un système réparti est dit « synchrone » si et seulement si :

- la durée de transmission d'un message d'un nœud à un autre est bornée et la borne est connue ; et,
- la durée d'exécution d'une action interne d'un processus est bornée et la borne est connue.

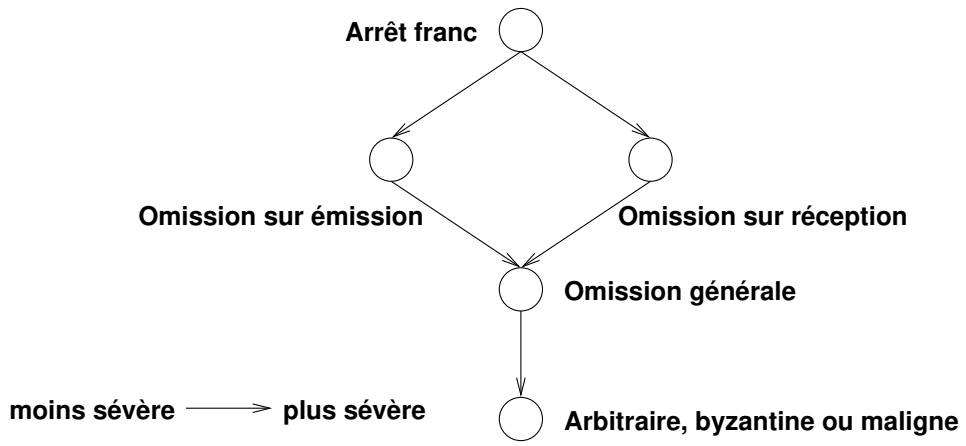
Dans un tel système réparti, un processus émettant un message peut faire l'hypothèse qu'il est reçu, voire traité, après une durée limite calculable (car les bornes sont connues).

Un système réparti est dit « asynchrone » s'il n'existe pas de borne (connue ou non) sur la transmission d'un message, la dérive des horloges ou la durée d'exécution d'une action interne. Dans la pratique, construire un algorithme pour un système asynchrone signifie ne pas s'occuper des caractéristiques matérielles (qualité des nœuds ou des communications). En d'autres termes, dès que des aspects temporels sont introduits dans les algorithmes (hypothèse sur les durées d'exécution ou de transmission, test de fiabilité de transmission à l'aide de temporisation, etc.), le système considéré n'est plus « complètement asynchrone », mais dit « partiellement asynchrone ». L'acception « partiellement asynchrone » recouvre le modèle « synchrone » et trente-et-un autres modèles, tous entre « complètement asynchrone » et « synchrone »[1]. Nous ne détaillons pas ces nombreux modèles et n'abordons pas la tolérance aux fautes dans ce manuscrit. C'est l'objectif des études d'articles réalisées par groupe dans le cadre du module.

1. D. Dolev, C. Dwork, and L. Stockmeyer, *On the minimal synchronism needed for distributed consensus*, Journal of the ACM, 34(1), January 1987.

8

1.1.3 Types de défaillances



- Hormis lorsque précisé, les algorithmes présentés ne tolérent pas les défaillances
- Dans tous les cas, les défaillances arbitraires sont exclues de l'étude

Un processus ou un nœud (dans le cas où l'on ne considère qu'un processus par nœud) est dit « défaillant » lors d'une exécution si son comportement diffère de la spécification de l'algorithme qu'il exécute. Sinon, il est dit « correct ». Il en est de même pour les canaux de communication entre processus. Un modèle de défaillances définit les défaillances rencontrées (et prises en compte ou tolérées). La littérature liste communément les types de défaillances suivants :

- arrêt franc initial (en anglais, *initial crash*) : un processus n'exécute aucune action de son algorithme local, autre que l'action initiale. Ce type de défaillance n'est pas montré sur la figure ; il se trouverait avant le nœud « arrêt franc » dans le graphe ;
- arrêt franc (en anglais, *crash*) : un processus s'arrête prématurément et ne fait rien ensuite ; avant l'arrêt, son exécution est correcte. Dans le cas d'un canal de communication, celui-ci est définitivement coupé ;
- « omission sur émission » : un processus s'arrête prématurément, omet d'émettre des messages par intermittence ou les deux. Dans le cas d'un canal de communication, l'omission sur émission correspond à une perte de messages. Par exemple, un processus jetant des messages parce que son *buffer* de messages en émission est plein, subit des omissions sur émission ;
- « omission sur réception » : un processus s'arrête prématurément, omet de recevoir des messages par intermittence ou les deux. Dans le cas d'un canal de communication, l'omission sur réception correspond à une perte de messages. Par exemple, un processus jetant des messages parce que son *buffer* de messages en réception est plein, subit des omissions sur réception ;
- « omission générale » : un processus est sujet à omission sur émission ou sur réception, voire les deux ;
- « arbitraire, byzantine[1] ou maligne » : un processus peut être sujet à n'importe quel comportement, y compris de la malveillance (de la part d'un utilisateur) ; par opposition, les défaillances précédentes sont dites « bénignes ». Pour un canal de communication, cela correspond à la perte, la duplication, la corruption (violation de l'intégrité), voire la génération spontanée d'un message.

Ces types de défaillances peuvent être classés en termes de sévérité. La figure ordonne les types de défaillances, des défaillances les moins sévères (arrêt francs) au plus sévères (arbitraires). Un algorithme tolérant les défaillances arbitraires tolère aussi les arrêts francs.

Les types de défaillances présentés ici existent aussi bien dans les systèmes synchrones qu'asynchrones. En outre, dans les systèmes synchrones, les défaillances peuvent aussi être « temporelles ». Un processus sujet à des défaillances temporelles peut défaillir des manières suivantes :

- omission générale ;

1. Le qualificatif « byzantin » vient de l'article célèbre de Lamport, Shostak et Pease « *The Byzantine Generals Problem* » de 1982, ACM Transactions on Programming Languages and Systems, 4(3):382–401, July 1982.

- « défaillance de l'horloge locale » : l'horloge locale dérive au delà de la borne tolérée ;
- « défaillance de performance » : la durée d'exécution d'un traitement dépasse la borne tolérée ou est trop courte. Pour un canal de communication, il s'agit d'une transmission trop rapide ou trop lente.

Dans ce cours, hormis lorsque nous le précisons explicitement dans de rares cas, les algorithmes présentés sont conçus dans le cas de systèmes répartis sans défaillance. En outre, les défaillances arbitraires étant très difficiles à tolérer, le cours ne les aborde pas du tout. La tolérance aux fautes bénignes est étudiée dans les études bibliographiques du module.

1.2 Conventions de codage des algorithmes répartis

9

■ Notation orientée contrôle

```
1  Chaque processus p exécute :
2      var r[q] pour tout q ∈ Voisins init F ;
3      begin
4          while #{q : r[q] = F} > 1 do
5              recevoir⟨JETON⟩ de q
6              r[q] := T
7          émettre⟨JETON⟩ vers q0 avec r[q0] = F
8          recevoir⟨JETON⟩ de q0
9          r[q0] := T
10         decider
11     end
```

■ Notation orientée évènement

```
1  Chaque processus p exécute :
2      var r[q] pour tout q ∈ Voisins init F ;
3      var sp, dp init F ;
4      Sp : {#{q : r[q] = F} = 1 and sp = F}
5          émettre⟨JETON⟩ vers q0 avec r[q0] := F
6          sp := T
7      Rp : {Un message ⟨JETON⟩ est arrivé}
8          recevoir⟨JETON⟩ de q
9          r[q] := T
10     Dp : {#{q : r[q] = F} = 0 et dp = F}
11         decider ; dp := T
```

Dans les deux formes, les opérations de réception `recevoir(m)` ne spécifient pas le processus émetteur du message, mais l'émetteur est connu après la réception, c'est-à-dire dans la portion d'algorithme traitant la réception. La notation « $\#E$ » est utilisée pour signifier la cardinalité de l'ensemble E. Les notations « T » et « F » sont utilisées pour signifier les valeurs booléenes « vrai » et « faux », respectivement. Les deux algorithmes de cette page réalisent le même traitement, mais ce n'est pas ce qui nous intéresse dans cette diapositive.

Les deux orientations (contrôle et évènement) sont possibles pour chaque algorithme, mais dans de nombreux cas, l'une est plus commode que l'autre. L'orientation contrôle d'un algorithme consiste en un algorithme séquentiel par processus avec des actions d'émission et de réception. La structure de l'algorithme est ainsi exprimée explicitement. En revanche, le non-déterminisme est plus facilement exprimé (parce qu'implicite) dans l'orientation évènement. La spécification consiste en une déclaration de variables suivie d'une liste d'actions. Chaque action consiste en une expression booléenne (l'action de garde) et un bloc d'instructions correspondantes. L'action est autorisée ou applicable lorsque la garde est évaluée à vrai. Dans ce cas, les instructions de l'action sont exécutées atomiquement, c'est-à-dire sans interruption dans le bloc. Les actions sont exécutées dans n'importe quel ordre de façon non déterministe.

10

1.3 Relation « arrivé avant » aussi appelée précédence causale, Lamport 1978

- Précédence causale (en anglais, *happened before relation*) entre évènements :
  - La relation « $\rightarrow$ » sur les évènements d'un système est la plus petite relation satisfaisant les conditions suivantes :
    - Si a et b sont des évènements sur un même processus, et a arrive avant b, alors $a \rightarrow b$
    - Si a est l'émission d'un message par un processus et b la réception du même message par un autre processus, alors $a \rightarrow b$
    - Si $a \rightarrow b$ et $b \rightarrow c$, alors $a \rightarrow c$
- La relation « $\rightarrow$ » définit un ordre partiel irréflexif
- $a||b = a \not\rightarrow b \wedge b \not\rightarrow a$ signifie que les deux actions sont concurrentes
- Ensuite, l'horloge logique prend différentes formes
  - Scalaire, vecteur ou matrice + avec estampillage des messages
  - Et lorsque les dépendances causales sont construites à base de vecteurs
    - Dépendances directes : estampilles des messages contenant un scalaire
    - Dépendances indirectes : estampilles des messages contenant un vecteur

La non-unicité de temps impliquant le fait qu'il est impossible de réaliser un observateur qui voit tout le système de manière immédiate, empêche l'utilisation d'une horloge physique exacte pour ordonner les actions d'un algorithme réparti. Nous avons recours à une horloge globale logique. Dans un article de 1978, qui est parmi les plus cités, voire le plus cité, de la littérature des systèmes répartis, Lamport définit la relation binaire appelée « précédence causale » (en anglais, *happened before relation*) entre les évènements[1] d'un système, et notée « $\rightarrow$ », comme étant la plus petite relation satisfaisant les conditions suivantes :

- si a et b sont des évènements sur un même processus, et a arrive avant b, alors $a \rightarrow b$;
- si a est l'émission d'un message par un processus et b la réception du même message par un autre processus, alors $a \rightarrow b$;
- si $a \rightarrow b$ et $b \rightarrow c$, alors $a \rightarrow c$.

Cette relation capture la relation de cause à effets.

Ensuite, le temps logique peut être exprimé par un scalaire ou bien par un vecteur, ou encore par une matrice. Les estampilles sont alors des scalaires ou bien des vecteurs, ou encore des matrices. Pour les estampilles contenant un scalaire, leur taille est très petite et fixe, le calcul de l'horloge rapide, et la précision de l'horloge très faible. À l'opposé, pour les estampilles contenant une matrice, leur taille est grande et fonction du nombre de processus, le calcul de l'horloge long, et la précision de l'horloge très grande. La méthode la plus employée se situe entre ces deux extrêmes et utilise un vecteur pour repérer le temps logique. Dans ce cas, si l'estampille du message m contient le scalaire S (la valeur du vecteur de l'émetteur à l'indice de l'émetteur, c'est-à-dire $V_i[i]$), les dépendances sont dites *directes*. Dans l'autre cas, les messages transportent le vecteur de l'émetteur, c'est-à-dire V_i, et les dépendances sont dites *indirectes*

La diapositive qui suit présente la mise en œuvre de Lamport avec des scalaires et la suivante la mise en œuvre de Fidge avec des vecteurs.

1. Par un petit abus de langage, nous utilisons indistinctement les termes « évènement » et « action » : selon la terminologie utilisée par Lamport dans l'article définissant la précédence causale, « *the processes are algorithms and the events represent certain actions during their execution* ».

1.3.1 Algorithme de calcul des horloges scalaires de Lamport 1978

11

```
1  var H_p : entier init 0 ;
2  I_p : { Une action interne }
3     H_p := H_p + 1
4     changement état
5  S_p : { Une action d'émission }
6     H_p := H_p + 1
7     émettre ⟨message, H_p⟩
8  R_p : { Une action de réception }
9     recevoir ⟨message, H⟩
10    H_p := max(H_p, H) + 1
```

L'*horloge logique* d'un processus est réalisée par un nombre entier, qui est incrémenté à chaque action du processus. L'*horloge globale logique* est réalisée en estampillant les messages inter-processus. Pratiquement, l'action émettre insère l'horloge logique du processus émetteur au message et l'action recevoir met à jour la connaissance de l'horloge logique du processus récepteur en fonction de l'estampille du message reçu.

Dans cet algorithme très simple, l'action I_p est celle qui est exécutée par le processus p pour ces actions internes, l'action S_p est celle qui est exécutée par le processus p pour émettre un message, et l'action R_p est celle qui est exécutée par p pour recevoir un message. Cet algorithme « s'ajoute » donc à l'algorithme applicatif : les opérations émettre sont remplacées par l'action S_p, etc.

Par cet algorithme, Lamport définit une mise en œuvre à base de scalaires. La relation binaire notée « $\prec$ » sur les actions est définie comme suit :

$$\mathsf{a}_i^x \prec \mathsf{a}_j^y \stackrel{\text{def}}{=} \begin{cases} & (i=j) \wedge (y = x+1) \\ \vee & \exists m : (\mathsf{a}_i^x = \texttt{émettre}(P_j, m)) \wedge (\mathsf{a}_j^y = \texttt{recevoir}(m)) \\ \vee & \exists \mathsf{a}_k^z : (\mathsf{a}_i^x \prec \mathsf{a}_k^z) \wedge (\mathsf{a}_k^z \prec \mathsf{a}_j^y) \end{cases}$$

Théorème 1 *Soient deux évènements a et b. L'algorithme de cette diapositive garantit la condition*

$$a \to b \implies a \prec b.$$

Preuve : Par la définition de la relation « $\to$ », il est aisé de voir que l'implication est obtenue avec les deux conditions suivantes :

- si a et b sont des évènements du processus p_i tels que a arrive avant b, alors $a \prec b$;
- si a est l'émission d'un message par le processus p_i et b la réception de ce message par le processus p_j, alors $a \prec b$.

La première condition est vérifiée grâce à l'incrémentation de la variable H_p dans l'action I_p de l'algorithme : la valeur de H_p à l'instant a est inférieure à la valeur de H_p à l'instant b. La seconde condition est vérifiée conjointement par l'incrémentation avant l'émission du message, puis la transmission de l'horloge scalaire de l'émetteur dans le message, et enfin le calcul $(\max(H_p, H) + 1)$ par le processus récepteur : la valeur de H_p à l'instant a est inférieure à la valeur de H_p à l'instant b. □

1.3.2 Algorithme de calcul des horloges vectorielles de Fidge 1991

12

```
1  var H_p[q] : pour tout q ∈ Π init 0 ;
2  I_p : { Une action interne }
3     H_p := H_p + 1_p
4     changement état
5  S_p : { Une action d'émission }
6     H_p := H_p + 1_p
7     émettre ⟨message, H_p⟩
8  R_p : { Une action de réception }
9     recevoir ⟨message, H⟩
10    H_p := max(H_p, H) + 1_p
```

L'horloge à base de scalaires n'a pas la précision suffisante pour construire la relation contraposée : la relation $(a \prec b \implies a \to b)$ n'est pas vraie. En d'autres termes, la relation $\prec$ définit un parmi les ordonnancements possibles des évènements. Pour obtenir l'ordre partiel de façon complète, dans des travaux parallèles, Fidge[1] et Mattern[2] introduisent en 1988 les vecteurs d'horloge.

Nous utilisons la formulation plus générale que Fidge propose en 1991 dans laquelle il utilise un scalaire par processus, avec chaque processus mémorisant un ensemble de paires contenant un identifiant de processus ainsi que la dernière valeur de l'horloge scalaire connue. Les processus du système non encore connus n'ont pas de paires correspondantes et la valeur de leur horloge est par défaut égale à 0. Les processus copient leur ensemble dans les messages qu'ils envoient. Vu autrement, en considérant que l'ensemble des processus Π est connu par chacun d'entre eux au démarrage du système et que les identifiants des processus sont totalement ordonnés, chaque processus gère un vecteur d'horloge et copie la valeur de son vecteur d'horloge dans les messages qu'il émet. C'est cette dernière solution qui est présentée dans l'algorithme de cette diapositive.

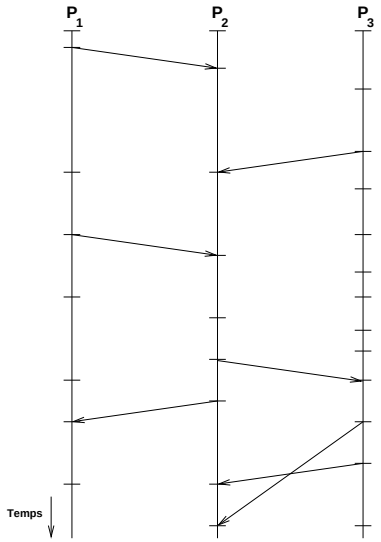
Les définitions suivantes sont ajoutées pour la manipulation des vecteurs d'horloge :

- $V = 1_p \stackrel{\text{def}}{=} V[p] = 1 \wedge (\forall i \neq p, V[i] = 0)$
  - par exemple, $1_1 = [1, 0, 0, 0]$
- $V \leq W \stackrel{\text{def}}{=} \forall i, V[i] \leq W[i]$
  - par exemple, $[0, 1, 0, 1] \leq [1, 1, 0, 1]$, mais $[0, 1, 0, 1] \not\leq [1, 1, 0, 0]$
- $V := \max(W, X) \stackrel{\text{def}}{=} \forall i, V[i] := \max(W[i], X[i])$

1. C.J. Fidge, *Timestamps in Message-Passing Systems that Preserve the Partial Ordering*, In Proc. of the 11th Australian Computer Science Conference, volume 10, pages 56–66, February 1988.
2. F. Mattern, *Virtual Time and Global States of Distributed Systems*, In Proc. of the International Workshop on Parallel and Distributed Algorithms, pages 215–226, Chateau de Bonas, France, October 1988.

1.3.3 Exercices

- Donnez une exécution de l'algorithme de Lamport 1978 avec le diagramme temporel ci-contre
- Donnez une exécution de l'algorithme de Fidge 1991 avec le diagramme temporel ci-contre



13

1.4 Vague et traversée de graphe

14

- Propriétés
  - Terminaison : chaque exécution répartie est finie
  - Décision : chaque exécution répartie contient au moins une action `décider`
  - Dépendance : dans chaque exécution répartie, l'action `décider` est causalement précédée par une action `recevoir` dans chaque processus
- Processus initiateurs démarrant une vague
  - 1 seul initiateur = algorithme centralisé
    - Algorithme de traversée du graphe de processus construisant un arbre couvrant
      - ★ Canal choisi pour l'arbre = celui du premier message de la vague reçu

Parmi les problèmes fondamentaux que nous étudions dans ce cours, beaucoup peuvent s'exprimer à l'aide de sous-tâches génériques comme les vagues ; intuitivement, une vague est l'exécution d'un algorithme telle que (1) un ou plusieurs processus démarrent la vague en envoyant des messages et (2) tous les autres processus reçoivent puis émettent un message de la vague sur tous les canaux sortants. Nous présentons dans cette section le principe des algorithmes de vagues et y ferons régulièrement référence dans la suite du cours.

Un algorithme de vague échange un nombre fini de messages, puis prend une décision qui dépend causalement d'une action dans chaque processus. Une exécution est une collection d'actions partiellement ordonnées par la relation $\preceq$ définie comme suit : $\mathsf{a}_i^x \preceq \mathsf{a}_j^y = (\mathsf{a}_i^x \prec \mathsf{a}_j^y) \vee (\mathsf{a}_i^x = \mathsf{a}_j^y)$, avec $\mathsf{a}_i^x = \mathsf{a}_j^y$ signifiant $x = y \wedge i = j$. Soit les notations suivantes : (1) $\mathcal{W}$ l'exécution d'une vague, (2) $|\mathcal{W}|$ le nombre d'actions dans $\mathcal{W}$, (3) Π l'ensemble des processus, et (4) $\mathcal{W}_p$ l'ensemble des actions exécutées par le processus p dans $\mathcal{W}$. Un algorithme de vague satisfait les trois propriétés suivantes :

1. terminaison : chaque exécution répartie est finie,
   - formellement, $\forall \mathcal{W}, |\mathcal{W}| < \infty$
2. décision : chaque exécution répartie contient au moins une action `décider`,
   - formellement, $\forall \mathcal{W}, \exists a \in \mathcal{W} : a = \texttt{décider}$
3. dépendance : dans chaque exécution répartie, l'action `décider` est causalement précédée par une action `recevoir` dans chaque processus,
   - formellement, $\forall \mathcal{W}, \forall a \in \mathcal{W}, a = \texttt{décider} \implies (\forall q \in \Pi, \exists b \in \mathcal{W}_q : b \preceq a \wedge b = \texttt{recevoir})$.

Dans l'exécution d'une vague, une distinction est faite entre les processus initiateurs et les autres. Un algorithme de vague est dit centralisé s'il ne peut y avoir qu'un initiateur. Il est possible de montrer que toute action `décider` est précédée dans chaque processus par au moins une action `émettre` et dans chaque processus non-initiateur par au moins une action `recevoir`.

L'exécution d'un algorithme de vague avec un seul initiateur construit un arbre couvrant du graphe du réseau de communication :

1. il n'y a qu'un initiateur, ce dernier émet un message ;
2. suite à la réception d'un message, chaque processus dans le graphe soit émet un message soit décide ;
3. l'algorithme se termine chez l'initiateur, et lorsque cela arrive, chaque processus a émis un message au moins une fois.

1.4.1 Algorithme de vague centralisé Écho de Segall, 1983

15

```
1  var rec_p : entier init 0
2  var parent_p : processus ∪{udef} init udef
3  Pour l'initiateur :
4     for all q ∈ Neigh_p do émettre ⟨jeton⟩ vers q
5     while rec_p < #Neigh_p do recevoir ⟨jeton⟩ ; rec_p := rec_p + 1
6     décider()
7  Pour les non-initiateurs :
8     recevoir ⟨jeton⟩ de q ; parent_p := q ; rec_p := rec_p + 1
9     for all q ∈ Neigh_p ∧ q ≠ parent_p do émettre ⟨jeton⟩ vers q
10    while rec_p < #Neigh_p do recevoir ⟨jeton⟩ ; rec_p := rec_p + 1
11    émettre ⟨jeton⟩ vers parent_p
```

Un algorithme peut être adapté à une topologie donnée, être centralisé ou décentralisé. Nous présentons deux algorithmes pour réseaux arbitraires, l'un est centralisé, l'autre décentralisé. Le premier, l'algorithme centralisé Écho de Segall, 1983, inonde le réseau de messages ⟨JETON⟩ en construisant ainsi un arbre couvrant. Les JETONs sont renvoyés en écho via les arêtes de cet arbre comme dans un algorithme de flot dans un arbre.

La notation « $\#E$ » est utilisée pour signifier la cardinalité de l'ensemble E, et $Neigh_p$ avec *Neigh* pour *neighbour* repère l'ensemble des processus voisins de p au sens où il existe un canal de communication bidirectionnel entre p et chaque processus de $Neigh_p$.

Par ailleurs, pour simplifier l'écriture dans cette diapositive, lorsque les boucles for all et while « tiennent » sur une ligne, nous n'avons pas écrit done à la fin de la ligne pour signifier la fin de la boucle.

L'initiateur de l'algorithme Écho émet un message ⟨JETON⟩ vers tous ses voisins, puis entre dans une boucle d'attente de réception d'un message ⟨JETON⟩ de tous ses voisins, et enfin, décide. Sur réception du premier message ⟨JETON⟩, un processus non initiateur fait suivre le message ⟨JETON⟩ vers tous ses voisins, excepté vers celui duquel provient le ⟨JETON⟩. Pour un non-initiateur, le processus émetteur du premier message ⟨JETON⟩ reçu est le père dans l'arbre couvrant. Lorsqu'un processus non initiateur a reçu un message ⟨JETON⟩ de tous ses voisins, le message ⟨JETON⟩ est renvoyé en écho au processus père.

Nous présentons dans la suite de cette section les éléments de preuve de l'algorithme Écho de Segall, 1983.

Lemme 1 *Soit $\mathcal{W}$ une vague ayant comme processus initiateur p. Pour chaque processus non initiateur q, notons $parent_q$ le processus voisin de q duquel q reçoit le premier message dans $\mathcal{W}$. Le graphe $T = (V_T, E_T)$, avec $V_T = \Pi$ l'ensemble des sommets et $E_T = \{qr | q \neq p \wedge r = parent_q\}$ l'ensemble des arêtes, est un arbre couvrant dirigé vers p.*

PREUVE : Comme le nombre de sommets de T est égal au nombre d'arêtes plus un, il suffit de montrer que T ne contient pas de cycle. C'est le cas car, étant donné a_q la première action de q dans $\mathcal{W}$ et a_r la première action de r dans $\mathcal{W}$, a_q est la réception par q du premier message de la vague de $\mathcal{W}$ et $qr \in E_T$ implique $a_r \preceq a_q$, et la relation $\preceq$ définit un ordre partiel. □

Théorème 2 *L'algorithme Écho de Segall, 1983 est un algorithme de vague.*

PREUVE : Remarquons que dans l'algorithme Écho de Segall, 1983 chaque processus émet au maximum un message vers chacun de ses voisins. Le nombre de messages est donc fini, ce qui montre la terminaison de l'algorithme.

Soit ω la configuration terminale atteinte par l'exécution $\mathcal{W}$ ayant pour processus initiateur p_0. Pour cette configuration, définissons le graphe $T = (V_T, E_T)$, avec $V_T = \Pi$ l'ensemble des sommets et $E_T = \{pq | q \neq$

$p \wedge parent_p = q\}$ l'ensemble des arêtes. Pour montrer que T est un arbre, nous devons montrer que le nombre d'arêtes est égal au nombre de sommets moins un. Observons que chaque processus participant à $\mathcal{W}$ émet un message à chacun de ses voisins, excepté, si le processus n'est pas l'initiateur, au voisin duquel il reçoit le premier message. Cela implique que chacun de ses voisins reçoit au moins un message dans $\mathcal{W}$ et participe donc à $\mathcal{W}$. Il s'ensuit que $\forall p \neq p_0, parent_p \neq udef$. Par conséquent, en suivant le même raisonnement que dans le Lemme 1, T ne contient pas de cycle. Nous pouvons donc construire un raisonnement par induction en utilisant T.

L'arbre T est dirigé vers p_0. Notons T_p l'ensemble des sommets du sous-arbre de p : T_p consiste en l'union des arbres T_q sur les enfants q de p et p lui-même. Remarquons aussi que, dans la configuration terminale ω, puisque chaque processus p a émis un message vers ses voisins, excepté vers $parent_p$, les arêtes du réseau qui ne sont pas dans T ont vu passer un message dans les deux directions. Notons b_p (si elle existe) l'action dans laquelle p émet un message vers $parent_p$ et c_p (si elle existe) l'action de réception de ce message par $parent_p$. Par induction sur les sommets de T, nous pouvons montrer que :

- $\forall p \neq p_0, \mathcal{W}$ contient l'action b_p,
- $\forall s \in T_p, \exists d \in \mathcal{W}_s : d \preceq c_p$.

Considérons les deux cas suivants :

- p est une feuille de T. T_p contient uniquement p. p a reçu un message de $parent_p$ et de tous ses autres voisins (puisque les arêtes depuis ses autres voisins ne sont pas dans T). Donc, l'action b_p est applicable, et comme ω est la configuration terminale, b_p a été exécutée. Par conséquent, $b_p \preceq c_p$;
- p n'est pas une feuille de T. p a reçu un message de $parent_p$ et de tous ses voisins qui ne sont pas dans T_p. Par induction, pour chaque enfant p' de p, $\mathcal{W}$ contient l'action $b_{p'}$, et puisque ω est la configuration terminale, $\mathcal{W}$ contient $c_{p'}$. Donc, l'action b_p est applicable, et comme ω est la configuration terminale, b_p a été exécutée. Il est par conséquent possible de montrer par induction que $\forall p' \in T_p : b_{p'} \preceq c_{p'}$.

À partir du raisonnement par induction précédent, il est clair que, dans la configuration terminale ω, p_0 a reçu un message de chacun de ses voisins. L'action `décider` est donc applicable et exécutée. Cela prouve la propriété de décision. Enfin, l'action `décider` est causalement précédée par une action `recevoir` dans chaque processus. Cela prouve la propriété de dépendance. En conclusion, l'algorithme Écho de Segall, 1983 est un algorithme de vague. □

1.4.2 Algorithme de vague décentralisé de Finn, 1979 *

16

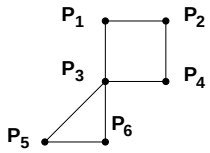
```
1  var Inc_p : ensemble de processus init {p}
2  var NInc_p : ensemble de processus init ∅
3  var rec_p[q] : booléen pour tout q ∈ In_p init false
4  begin
5     if p est initiateur then
6        for all r ∈ Out_p do émettre ⟨Inc_p, NInc_p⟩ vers r
7     while Inc_p ≠ NInc_p do
8        recevoir ⟨Inc, NInc⟩ de q_0
9        Inc_p := Inc_p ∪ Inc ; NInc_p := NInc_p ∪ NInc
10       rec_p[q_0] := true
11       if ∀q ∈ In_p : rec_p[q] = true then NInc_p := NInc_p ∪ {p}
12       if Inc_p ou NInc_p ont changé then
13          for all r ∈ Out_p do émettre ⟨Inc_p, NInc_p⟩ vers r
14    décider
15 end
```

L'algorithme décentralisé de Finn suppose que chaque processus connaît son identité et que les noms sont uniques. En outre, il considère que le graphe du réseau de communication est orienté, d'où l'utilisation de In_p et Out_p. Chaque processus p maintient deux ensembles d'identités de processus : Inc_p et $NInc_p$. Inc_p est l'ensemble des processus j tels qu'une action dans j précède l'action la plus récente dans p. $NInc_p$ est l'ensemble des processus j tels que pour tous les voisins k de j, une action dans k précède l'action la plus récente dans p.

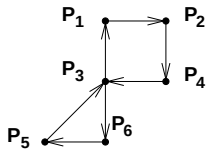
Initialement, $Inc_p = \{p\}$ et $NInc_p = \emptyset$. Le processus p émet un message à tous ses voisins, incluant Inc_p et $NInc_p$, à chaque fois que l'un des ensembles a changé. Lorsque p reçoit un message, il met à jour Inc_p et $NInc_p$ en incluant respectivement les ensembles Inc et $NInc$ contenus dans le message. Lorsque p a reçu un message de tous ses voisins, p est inséré dans $NInc_p$. Enfin, lorsque les deux ensembles Inc_p et $NInc_p$ sont égaux, p décide.

1.4.3 Exercices

17



(a)



(b)

? Dans la topologie (a), donnez une exécution de l'algorithme Écho avec P_1 initiateur
? Dans la topologie (b), donnez une exécution de l'algorithme de Finn avec P_1 initiateur *
? Même dernière question avec plusieurs initiateurs *

La deuxième et la troisième questions sont optionnelles et à faire après l'étude de l'algorithme de Finn.

2 Élection

18

Cette section sur l'élection est reprise de la référence suivante :

- G. Tel, *Chapter 7 : Election Algorithms*, dans Introduction to Distributed Algorithms, Cambridge University Press, pp. 222–263, 1994.

Le problème de l'élection est de partir d'une configuration dans laquelle tous les processus sont dans le même état pour arriver dans une configuration dans laquelle un seul processus est dans l'état « gagnant » et tous les autres dans l'état « perdant ».

Une élection est par exemple nécessaire lorsqu'un algorithme centralisé doit être exécuté alors qu'aucun processus ne peut *a priori* servir d'initiateur. C'est aussi un algorithme très utile lorsque des défaillances surviennent. Par exemple, lors de la perte d'un jeton dans un algorithme d'exclusion mutuelle, les processus corrects élisent l'un des leurs pour regénérer un jeton. Les processus peuvent aussi promouvoir l'un des leurs en tant que coordinateur pour gérer un groupe de diffusion : liste des processus participants, liste des nouveaux entrants, etc.

2.1 Propriétés et vocabulaire

19

1. Chaque processus exécute le même algorithme : symétrie complète
2. L'algorithme est décentralisé : une exécution peut être commencée par un nombre quelconque de processus
3. L'algorithme atteint une configuration terminale dans laquelle il existe exactement un processus « gagnant » et tous les autres processus sont « perdants »

- *dormant* au début
- puis, *candidat* (tous n'ont pas besoin d'être candidats)
- et enfin, *gagnant* ou *perdant*

Un algorithme d'élection est un algorithme qui satisfait les trois propriétés suivantes :

1. chaque processus exécute le même algorithme : symétrie complète ;
2. l'algorithme est décentralisé : une exécution peut être commencée par un nombre quelconque de processus ;
3. l'algorithme atteint une configuration terminale dans laquelle il existe exactement un processus « gagnant » et tous les autres processus sont « perdants ».

La dernière propriété est quelquefois relâchée en : il existe un seul processus gagnant. Le processus gagnant est alors au courant qu'il a gagné l'élection, mais les autres ne savent pas encore qu'ils ont perdu. Dans ce cas, le processus gagnant diffuse le résultat aux autres processus.

Dans les algorithmes que nous étudions, chaque processus possède un nom unique qui peut être comparé ($=$, $>$, $<$, $\geq$ ou $\leq$) avec les identifiants des autres processus du système réparti.

Enfin, chaque processus possède une variable $state_p$ pouvant contenir les valeurs *gagnant* et *perdant*. Certaines fois, la variable $state_p$ est égale à *dormant* avant que le processus n'ait exécuté une étape de l'algorithme et *candidat* lorsque le processus participe à l'algorithme en étant candidat mais ne sait pas encore s'il est gagnant ou perdant.

2.2 Élection dans un anneau, algorithme de Le Lann, 1977

20

```
1  Every process p executes the following :
2     var List_p : set of P init {p} ;
3     var state_p init dormant ;
4     begin
5        if p is initiator then
6           state_p := candidat ; send ⟨JETON, p⟩ to Next_p ; receive ⟨JETON, q⟩ ;
7           while q ≠ p do
8              List_p := List_p ∪ {q} ; send ⟨JETON, q⟩ to Next_p ; receive ⟨JETON, q⟩ ;
9           if p = min(List_p) then state_p := gagnant ; else state_p := perdant ;
10       else
11          state_p := perdant
12          while true do
13             receive ⟨JETON, q⟩ ; send ⟨JETON, q⟩ to Next_p ;
14    end
```

Seul le processus gagnant termine le programme. Il envoie alors un message indiquant à tous qu'il a gagné. Tous alors terminent.

Les premiers algorithmes sont apparus pour des applications réparties dont les processus sont organisés en anneau unidirectionnel. C'est une topologie simple et très utilisée.

Le premier algorithme a été donné par Le Lann en 1977 pour des canaux unidirectionnels FIFO. L'algorithme possède une complexité en nombre de messages en $\mathcal{O}(n^2)$, avec n le nombre de processus. Chaque initiateur calcule l'ensemble des identifiants des processus de l'anneau, l'initiateur avec l'identifiant le plus petit est « gagnant ». Chaque initiateur p émet un message $\langle \text{JETON}, p\rangle$ contenant son identifiant, le message $\langle \text{JETON}, p\rangle$ est propagé sur l'anneau. Les canaux sont supposés FIFO et un initiateur p doit générer son message $\langle \text{JETON}, p\rangle$ avant qu'aucun autre message $\langle \text{JETON}, q\rangle$ ne soit reçu. Lorsqu'un initiateur p reçoit son message $\langle \text{JETON}, p\rangle$, les messages $\langle \text{JETON}, q\rangle$ de tous les autres processus ont visité p. p peut alors connaître l'identifiant du processus « gagnant ». Avant qu'un processus ne reçoive le premier message $\langle \text{JETON}, p\rangle$, s'il veut participer à l'élection en étan candidat, il doit avoir commencé à exécuter l'algorithme en se considérant comme un initiateur.

Théorème 3 *L'algorithme de Le Lann 1977 résoud le problème de l'élection : il est symétrique et décentralisé, et si nous considérons l'algorithme de la diapositive (lignes 1–14) augmenté, comme suggéré par le commentaire en bas de la diapositive, de l'annonce par le gagnant qu'il a gagné, alors l'algorithme se termine.*

Preuve : Clairement, l'algorithme exécuté par tous les processus est le même ; c'est dans cet algorithme que sont distingués les rôles des processus candidats ou non à l'élection.

L'exécution de l'algorithme peut être commencée par n'importe lequel des processus avec la contrainte que les processus initiateurs commencent par émettre leur jeton (c'est-à-dire, avant de recevoir le premier jeton).

Les canaux de communication étant FIFO, l'ordre de transmission des jetons est préservé. Un processus initiateur p émet le message $\langle \text{JETON}, p\rangle$ avant de recevoir les messages $\langle \text{JETON}, q\rangle$ de tous les processus initiateurs, et avant de recevoir le message $\langle \text{JETON}, p\rangle$. Donc, chaque processus initiateur construit l'ensemble $List_p$ contenant tous les initiateurs. Chaque processus initiateur calcule alors l'identité du processus initiateur ayant la plus petite identité. Par conséquent, le processus initiateur avec la plus petite identité est le seul qui est élu et tous les processus connaissent l'identité du processus élu.

Sans l'ajout de l'annonce par le processus gagnant, l'algorithme de Le Lann (lignes 1–14) ne se termine pas. □

La solution de Le Lann a été améliorée par Chang et Roberts en 1979, qui obtiennent un algorithme dont la complexité en nombre de messages est dans le pire des cas en $\mathcal{O}(n^2)$, et en moyenne en $\mathcal{O}(n\ log\ n)$. L'algorithme de Chang et Roberts améliore celui de Le Lann en enlevant de l'anneau tous les messages $\langle \text{JETON}, p\rangle$ des processus p qui perdront.

Voici l'algorithme décentralisé de Chang et Roberts, 1979 :

```
1  Every process p executes the following :
2    var state_p init dormant ;
3    begin
4      if p is initiator then
5        state_p := candidat ; send ⟨JETON, p⟩ to Next_p ;
6        while state_p ≠ gagnant do
7          receive ⟨JETON, q⟩ ;
8          if q = p then state_p := gagnant ;
9          else if q < p then
10               if state_p = candidat then state_p := perdant ;
11               send ⟨JETON, q⟩ to Next_p ;
12     else while true do
13           state_p := perdant ; receive ⟨JETON, q⟩ ; send ⟨JETON, q⟩ to Next_p ;
14   end
```

Dans cet algorithme, un initiateur p ne propage pas les messages $\langle \text{JETON}, q \rangle$ tels que $q > p$. Un initiateur p devient « perdant » lorsqu'un message $\langle \text{JETON}, q \rangle$ avec $q < p$ est reçu, et devient « gagnant » lorsqu'il reçoit son propre message $\langle \text{JETON}, p \rangle$.

2.3 Élection avec l'algorithme de vague Écho de Segall, 1983

21

```
1  Every process p executes the following :
2    var caw, parent, win : P ∪{udef} init udef ; rec, lrec : integer init 0 ; state : dormant ;
3    begin
4      if p is initiator then caw := p ; forall q ∈ Neigh do send ⟨JETON, p⟩ to q ;
5      while lrec < #Neigh do   {Fin de l'algo. = fin de la pseudo-vague de messages ⟨GAGNANT⟩}
6        receive msg from q ;
7        if msg =⟨GAGNANT, r⟩ then
8          if lrec = 0 ∧ r ≠ p then forall q ∈ Neigh do send ⟨GAGNANT, r⟩ to q ;
9          lrec := lrec + 1 ; win := r ;
10       else if msg =⟨JETON, r⟩
11         if caw = udef ∨ r < caw then caw := r ; rec := 0 ; parent := q ; forall s ≠ q ∈ Neigh
           do send⟨JETON, r⟩ to s ;
12         if r = caw then
13           rec := rec + 1 ;
14           if rec = #Neigh then
15             if caw = p then forall s ∈ Neigh do send ⟨GAGNANT, p⟩ to s ;
16             else send ⟨JETON, caw⟩ to parent ;
17     if win = p then state := gagnant else state := perdant ;
18   end
```

Un algorithme d'élection peut être obtenu à partir d'un algorithme de vague centralisé en appliquant le principe de l'extinction comme dans l'algorithme de Chang et Robert pour l'anneau. Chaque processus p candidat est un initiateur et lance une vague ; les messages de la vague de p sont estampillés avec l'identifiant de p. L'algorithme d'élection selon le principe de l'extinction assure que, quelque soit le nombre de vagues lancées, seule l'une d'entre-elles se termine par l'action `décide`. Toutes les autres vagues sont annulées avant qu'une action `décide` ne soit exécutée. Cette action `décide` est sémantiquement égale à « je suis le gagnant », ici appelé le leader, et je le dis aux autres avec une vague (incomplète).

Les variables de l'algorithme sont les suivantes :

- caw : numéro de processus de la vague active (en anglais, *currently active wave*),
- rec : nombre de messages $jeton$ reçus,
- $parent$: parent dans la vague caw,
- $lrec$: nombre de messages $gagnant$ reçus,
- win : identité du processus gagnant.

Chaque processus p est « actif » dans au plus une vague à la fois ; cette vague est repérée par la variable caw, initialisée à p si p est un initiateur, sinon à la valeur par défaut $udef$. Lorsqu'un message $\langle$JETON, $r\rangle$ est reçu par p, si $r > caw$, le message est simplement ignoré, la vague lancée par r disparaît. Si $r = caw$, le message est traité comme dans l'algorithme de vague. Si $r < caw$ ou si $caw = udef$, p rejoint l'exécution de la vague lancée par r et exécute $caw := r$. Lorsque la vague lancée par r exécute une action `décide` (dans l'algorithme Écho, l'action `décide` est exécutée par l'initiateur), r est « gagnant ». r termine l'algorithme d'élection en propageant le message $\langle$GAGNANT, $r\rangle$ dans l'arbre couvrant construit par la vague.

Lemme 2 *Soit $\mathcal{W}$ une vague et $d_p \in \mathcal{W}$ une action* `décider` *du processus p. Alors, nous pouvons dire :* $\forall q \neq p : \exists a_q \in \mathcal{W} : (a_q \preceq d_p \ \wedge \ a_q =$ `émettre`$)$.

Preuve : Comme $\mathcal{W}$ est une vague, il existe une action a_q exécutée par q qui précède d_p. Choisissons a_q comme étant la dernière action de q qui précède d_p. Montrons que a_q est une action d'émission. La propriété de précédence causale implique qu'il existe une séquence d'actions (une chaîne de causalité) $a_q = a_0, a_1, ..., a_k = d_p$, telle que pour chaque $i < k$, a_i et a_{i+1} sont des actions qui se suivent soit sur le même processus soit sur deux processus différents (avec la première action étant une émission de message et la seconde la réception correspondante). Puisque a_q est la dernière action de q précédent d_p, que a_0 est l'action d'initialisation du processus initiateur, et que a_1 est une action sur un processus différent, a_q est une action `émettre`. □

Théorème 4 *L'algorithme d'élection construit sur la base de l'algorithme de vague Écho résoud le problème de l'élection.*

Preuve : Tous les processus exécutent le même algorithme. C'est dans l'algorithme que sont distingués les rôles des processus candidats ou non à l'élection.

L'exécution de l'algorithme peut être commencée par n'importe lequel des processus avec la contrainte que les processus initiateurs commencent par émettre leur jeton (c'est-à-dire, avant de recevoir le premier jeton).

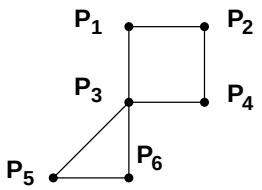
Soit p_0 le processus initiateur possédant la plus petite identité. Selon la condition $caw = udef \vee r < caw$, la vague initiée par p_0 traverse tous les processus autres que p_0 et tous ces mêmes processus terminent la vague car il n'existe pas de vagues éteignant la vague de p_0. Par conséquent, p_0 reçoit un jeton de tous ses voisins, et selon la condition $caw = p$, p_0 sait qu'il est élu. Ensuite, p_0 démarre la seconde vague émettant des messages donnant l'identité du processus élu p_0.

Si p n'est pas un processus initiateur alors aucune vague avec l'identité de p n'est démarrée. Par conséquent, p ne peut pas être élu. Si $p \neq p_0$ est un processus initiateur, une vague avec son identité est démarrée. D'après la propriété de dépendance des algorithmes de vague et le lemme 2, une éventuelle décision dans la vague de p est précédée par une émission par p_0 dans la vague de p. Or, p_0 n'exécute aucune action d'émission dans la vague de p, c'est-à-dire p_0 ne rejoint pas la vague de p. Donc, la vague de p ne se termine pas, aucune décision n'est prise, et p n'est pas élu.

Ainsi, une seule seconde vague donnant l'identité du processus élu p_0 est démarrée. Dans cette seconde vague (ce n'est en fait pas une « vraie » vague, au sens algorithmique du terme, car la fin de p_0 intervient sans qu'il soit nécessaire que la vague passe par tous les processus), la traversée du graphe n'est pas soumise à condition comme l'étaient les premières vagues. Par conséquent, la seconde vague se termine et tous les processus connaissent l'identité du processus élu. En conclusion, l'algorithme d'élection construit sur la base de l'algorithme de vague Écho résoud le problème de l'élection. □

22

2.3.1 Exercice



? Donnez une exécution de l'algorithme d'élection par vagues avec P_1 et P_5 initiateurs
? L'algorithme est-il correct si les canaux de communication ne sont pas FIFO

Donnez une exécution pas à pas de l'algorithme en montrant l'évolution des variables. Par exemple, pour cet algorithme, nous vous conseillons de noter les évolutions des variables *caw*, *parent*, *win*, *rec*, et *lrec*.

3 Diffusion

23

Cette section sur la diffusion ou les communications de groupe est reprise des références suivantes :

- V. Hadzilacos et S. Toueg, *Chapter 5 : Fault-Tolerant Broadcasts and Related Problems*, dans Distributed Systems 2nd Ed., Editeur S.J. Mullender, ACM Press, pp. 97–145, 1993.
- V. Hadzilacos et S. Toueg, *A Modular Approach to Fault-Tolerant Broadcasts and Related Problems*, dans Technical Report TR94-1425, Department of Computer Science, Cornell University, Ithaca (N.Y., USA), 84 pages, May 1994.
- K.P. Birman et T.A. Joseph, *Reliable Communication in the Presence of Failures*, ACM Transactions on Computer Systems, 5(1):47–76, February 1987.
- Guerraoui, R. and Rodrigues, L. *Introduction to Reliable Distributed Programming*, Springer, 2006.
- M.J. Fischer, N.A. Lynch et M.S. Paterson, *Impossibility of Distributed Consensus with One Faulty Process*, dans Journal of the ACM, Volume 32, Numéro 2, pp. 374–382, Avril 1985.

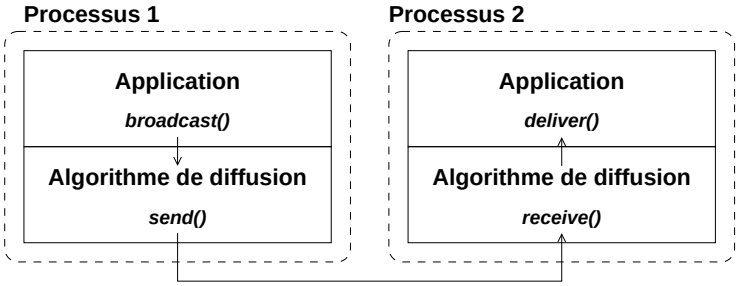
Il existe des cas où un processus doit envoyer le même message ou la même suite de messages à tous les autres processus : c'est la diffusion. Cette section introduit aussi un paradigme fondamental à la base de nombreuses solutions de tolérance aux fautes : le consensus. Le consensus est la faculté d'atteindre un accord entre processus répartis et de décider de façon cohérente : tous les processus corrects, c.-à-d. non défaillants, prennent la même décision. Nous verrons que le consensus utilise des diffusions et qu'il est équivalent à une forme de diffusion.

NB : cette section est celle qui montre quelque peu que la tolérance aux fautes des liens ou des processus est complexe à mettre en œuvre. Pourtant, nous ne considérons dans cette section que les fautes par arrêts francs (en anglais, *crash*).

3.1 Spécification des diffusions

24

- Type de système :
  - Diffusions fiable, FIFO et causale : possibles en systèmes synchrone ou asynchrone
  - Diffusions atomiques : possible uniquement en système synchrone ou partiellement synchrone
- Construction modulaire et hiérarchique :
  - Construction et preuve simplifiées
  - Remplacement et portabilité facilités



Dans les définitions des différents types de diffusion, nous supposons que les fautes sont bénignes. Cela permet non seulement de simplifier les définitions mais aussi rend possible l'explication des définitions de façon incrémentale. La spécification des diffusions ne s'attache pas à la topologie du réseau formé par les processus y participant. Ici, notre étude concerne l'ordre de transmission des messages diffusés : FIFO, causal et total.

Dans cette section, nous montrons que les algorithmes proposés pour les diffusions fiable, FIFO et causale fonctionnent pour les systèmes synchrones et asynchrones. Par contre, la diffusion atomique, et par conséquent les diffusions FIFO atomique et causale atomique, ne possèdent pas de solution algorithmique (déterministe) dans les systèmes asynchrones. Les algorithmes donnés pour ces dernières diffusions ne fonctionnent que pour les systèmes synchrones.

Les algorithmes sont construits d'une façon modulaire, à la manière des relations entre les diffusions. En commençant par la diffusion fiable, nous montrons comment résoudre les diffusions plus difficiles au-dessus de la diffusion fiable. Le premier avantage de cette démarche est de permettre le remplacement de l'un de ces algorithmes par un autre de la littérature et de fournir une panoplie complète d'algorithmes de diffusion. Les autres avantages non négligeables de cette démarche sont que les algorithmes sont plus simples, avec l'inconvénient d'être moins efficients, et les preuves sont facilitées. Un dernier avantage très important de cette démarche est la portabilité en termes de dépendances vis-à-vis des caractéristiques du système de communication utilisé : topologie, synchronisation...

Pour distinguer les différents algorithmes de diffusion, chaque type de diffusion est repéré par un acronyme (« R » pour *reliable*, fiable en anglais, « F » pour FIFO, « C » pour causale, « A » pour atomique, « FA » pour FIFO atomique et « CA » pour causale atomique). Le type de diffusion est inséré en premier paramètre des primitives `broadcast()` et `deliver()`.

25

3.2 Diffusion fiable

- Hypothèses sur le système réparti
  - Validité des communications point à point : si p émet le message m vers q, et le lien bidirectionnel entre p et q est correct, alors q recevra *in fine* m
  - Intégrité des communications point à point : pour tout message m, q reçoit m au plus une fois de p, et seulement si p a précédemment émis m vers q
  - Défaillances des liens et des processus par arrêt franc
  - Pas de partitionnement du système : tous les processus corrects sont reliés deux à deux par au moins un chemin constitué de processus et de liens corrects
- Validité (*validity*) : si un processus correct p diffuse un message m alors p livre ultimement[a] m
- Accord (*agreement*) : si un processus correct livre un message de diffusion m alors tous les processus corrects livrent ultimement m
- Intégrité (*integrity*) : pour chaque message de diffusion m, tout processus correct livre m au plus une fois, et seulement si m a été antérieurement diffusé par l'émetteur de m

a. « *in fine* », « au bout du compte », « dans un certain temps », « à terme »

Avant d'introduire les propriétés de la diffusion fiable, nous présentons le modèle de système réparti. Comme toutes les autres formes de diffusion reposent sur la diffusion fiable, c'est le même modèle de système réparti qui est supposé par la suite. Voici les hypothèses sur le système réparti :

- validité des communications point à point : si p émet le message m vers q, et le lien bidirectionnel entre p et q est correct, alors q recevra *in fine* m ;
- intégrité des communications point à point : pour tout message m, q reçoit m au plus une fois de p, et seulement si p a précédemment émis m vers q ;
- défaillances des liens et des processus par arrêt franc ;
- pas de partitionnement du système : tous les processus corrects sont reliés deux à deux par au moins un chemin constitué de processus et de liens corrects.

Informellement, la diffusion fiable demande que tous les processus corrects livrent le même ensemble de messages (accord), que cet ensemble contienne tous les messages diffusés par les processus corrects (validité) et qu'aucun message ne soit transformé (intégrité).

Un algorithme de diffusion satisfait les trois propriétés suivantes :

- validité (*validity*) : si un processus correct p diffuse un message m alors p livre ultimement m ;
- accord (*agreement*) : si un processus correct livre un message de diffusion m alors tous les processus corrects livrent ultimement m ;
- intégrité (*integrity*) : pour chaque message de diffusion m, tout processus correct livre m au plus une fois, et seulement si m a été antérieurement diffusé par l'émetteur de m.

Tout algorithme de diffusion est un algorithme de vague. En effet, le processus désirant diffuser le message est le processus initiateur. Le message doit visiter tous les processus (comme dans un algorithme de vague) avant la réception de la notification (qui précède l'action `décide` dans un algorithme de vague). Pendant la propagation du message, l'arbre couvrant est construit (s'il ne l'était pas déjà).

Il est important de remarquer que si un processus p défaille pendant la diffusion d'un message par `broadcast()`, la diffusion fiable possède deux alternatives : le message est livré soit par tous les processus corrects, soit par aucun. Par exemple, si p défaille avant le premier appel à `send()`, aucun processus ne peut livrer m puisqu'aucun ne connaît le message. Par contre, si p défaille après plusieurs appels à `send()` alors les processus corrects doivent livrer m, même si certains d'entre eux ne l'ont pas reçu directement de p ; c'est ce qui rend la fiabilité non triviale à atteindre.

26

3.2.1 Algorithme de diffusion fiable

```
1  Every process p executes the following :
2    To execute broadcast(R, m) :
3      tag m with sender(m) and seq#(m) /* These tags make m unique */
4      send(m) to all neighbours ∪ {p}
5    upon receive(m) do
6      if p has not previously executed deliver(R, m) then
7        if sender(m) ≠ p then send(m) to all neighbours
8        deliver(R, m)
```

Pour R-diffuser un message, un processus l'émet à tous ses voisins et à lui-même. Lorsqu'un processus reçoit un message pour la première fois, il le relaie à tous ses voisins et ensuite le livre à l'application. Rappelons qu'il est supposé que le réseau de communication ne partitionne pas l'ensemble des processus corrects. L'algorithme suppose aussi que les primitives `send()` et `receive()` utilisées pour transmettre un message respectent les propriétés de validité et d'intégrité.

Théorème 5 *L'algorithme présenté dans cette diapositive est un algorithme de diffusion fiable.*

Preuve : Validité : selon l'algorithme, lorsqu'un processus correct p diffuse un message m, il l'envoie vers tous ses voisins et à lui-même. Donc, trivialement, p livre m.

Accord : soient p et q deux processus corrects. Supposons que p livre m et montrons que q livre aussi m. Par l'hypothèse de non-partitionnement du système réparti, il existe un chemin entre p et q qui est constitué de liens et de processus corrects. Selon l'hypothèse de validité des communications point à point et selon l'algorithme, un simple raisonnement par récurrence sur le diamètre du graphe du système réparti démontre que tous les processus du chemin entre p et q reçoivent et livrent m, y compris p et q.

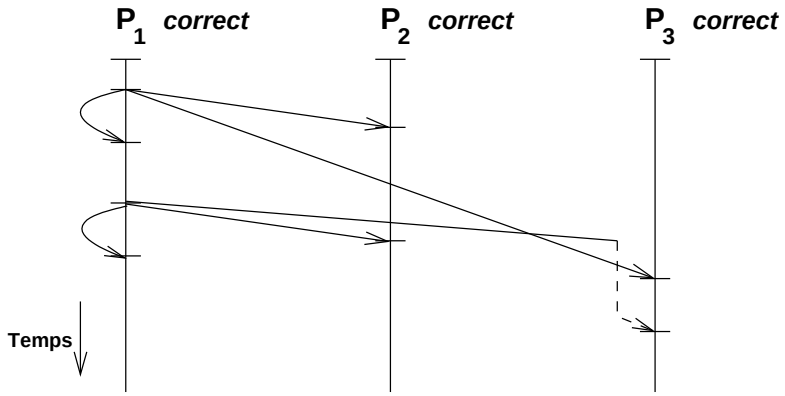
Intégrité : Selon l'algorithme, p livre m uniquement s'il n'a pas précédemment exécuté deliver(R, m). Supposons par ailleurs que q livre m. Par la propriété d'intégrité des communications point à point et par l'utilisation des méta-données ajoutées à m, nommément $sender(m)$ et $seq\#(m)$, q livre m qui a été émis par p. □

27

3.3 Diffusion FIFO

■ Diffusion fiable

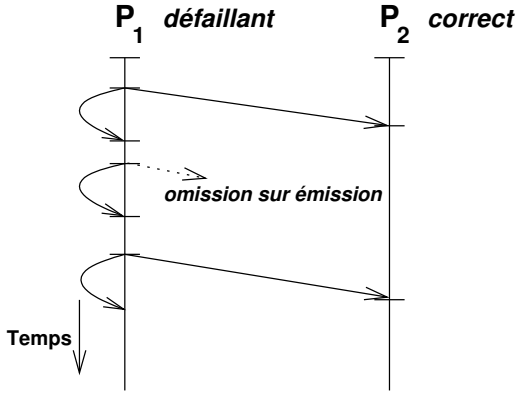
\+ Ordre FIFO : si un processus diffuse un message m_1 avant de diffuser un message m_2, alors aucun processus correct ne livre m_2 à moins qu'il n'ait déjà livré m_1



Dans certains cas d'utilisation, par exemple dans un système de réservation de places d'avions, les messages possèdent un contexte (lié à l'émetteur) en dehors duquel le message peut être mal interprété. Par exemple, une annulation de réservation n'a de sens que si le message contenant la demande de réservation est connu. Une telle application demande que l'algorithme de diffusion respecte la propriété suivante :

- ordre FIFO : si un processus diffuse un message m_1 avant de diffuser un message m_2, alors aucun processus correct ne livre m_2 à moins qu'il n'ait déjà livré m_1.

La définition de la diffusion FIFO est plus subtile qu'elle n'y paraît. Une définition (complète) telle que la suivante est erronée : « tous les messages diffusés par le même processus sont livrés par tous les processus dans l'ordre de leur émission ». Supposons, comme dans la figure suivante, qu'un processus P_1 diffuse les messages m_1, m_2 et m_3 dans cet ordre et qu'un autre processus P_2 livre m_1 puis m_3, mais jamais m_2. Ce scénario peut apparaître si P_1 a souffert d'omission sur émission lors de la diffusion de m_2. Ce scénario est permis par la dernière définition (puisque m_1 et m_3 sont livrés dans l'ordre de leur diffusion et que la définition autorise des « trous » dans la séquence de messages reçus), mais pas par celle que nous retenons.



3.3.1 Algorithme de diffusion FIFO

```
1  Every process p executes the following :
2    Initialisation :
3      msgBag := ∅ /* set of messages that p R-delivered but not yet F-delivered */
4      next[q] := 1 for all q /* sequence number of next message from q that p will F-deliver */
5    To execute broadcast(F, m) :
# 28 6      broadcast(R, m)
7    upon deliver(R, m) do
8      q := sender(m)
9      msgBag := msgBag ∪ {m}
10     while ∃n ∈ msgBag : (sender(n) = q) ∧ (seq#(n) = next[q]) do
11       deliver(F,n)
12       next[q] := next[q] + 1
13       msgBag := msgBag \ {n}
```

Il est aisé de construire un algorithme de diffusion FIFO à partir d'un algorithme de diffusion fiable. Rappelons que les messages sont estampillés avec le numéro de séquence d'émission et l'identifiant de l'émetteur. Chaque processus maintient un vecteur, une entrée par processus connu, mémorisant le numéro de séquence du prochain message de diffusion fiable à recevoir en provenance du processus distant. Lorsqu'un processus reçoit un message de diffusion fiable, il le met dans un ensemble. Ensuite, le processus livre à l'application tous les messages qui suivent, dans l'ordre des numéros de séquence, le dernier message livré en provenance du même émetteur, en s'arrêtant dès qu'un « trou » dans la séquence est observé. L'algorithme de diffusion FIFO n'ajoute pas de contrainte ou limitation en plus de celles de l'algorithme de diffusion fiable.

Lemme 3 *Pour tout processus p (correct ou défaillant), si $next_p[q] = k$ alors la séquence des messages que p a déjà F-livrés est constituée des $(k-1)$ premiers messages F-diffusés par q.*

Preuve : Le lemme est aisément démontré avec un raisonnement par récurrence sur k et en utilisant la propriété d'intégrité de la diffusion fiable. □

Lemme 4 *Supposons qu'un processus correct p R-livre le message m et F-livre tous les messages que l'émetteur de m a F-diffusés avant m. Alors, p F-livre aussi m.*

Preuve : Supposons que p et m satisfont les hypothèses du lemme, et soit q l'émetteur de m (c'est-à-dire $sender(m) = q$) et k le numéro de séquence de m (c'est-à-dire $seq\#(m) = k$). Par hypothèse, p F-livre les $k-1$ messages que q a F-diffusés avant m. Puisque p incrémente $next_p[q]$ lors de chaque F-livraison, à terme nous avons $next_p[q] \geq k$. Deux cas sont alors à considérer. Dans le premier cas, $next_p[q]$ atteint ultimement la valeur $k+1$. Selon le lemme 3, p F-livre les k premiers messages F-diffusés par q, y compris m. Dans le second cas, $next_p[q]$ atteint ultimement la valeur k et ensuite reste sur cette valeur. Soit m' le message dont la F-livraison par p amène à la situation $next_p[q] = k$. Par hypothèse, p R-livre m. Si, lorsque cela arrive, m' a déjà été livré, alors nous avons $next_p[q] = k$ et donc p peut F-livrer m. Sinon, p insère m dans $msgBag_p$ et quand plus tard p F-livre m' et exécute $next_p[q] := k$, p F-livre aussi m. Par conséquent, p F-livre m dans tous les cas. □

Théorème 6 *L'algorithme présenté dans cette diapositive transforme tout algorithme de diffusion fiable en un algorithme de diffusion FIFO.*

Preuve : Validité : procédons par contradiction. Supposons qu'un processus p F-diffuse un message qu'il ne F-livre jamais. Soit m ce premier message que p F-diffuse. Selon l'algorithme, puisque p F-diffuse m, p R-diffuse m. Selon la propriété de validité de la diffusion fiable, p R-livre m *in fine*. Par hypothèse sur m,

p F-livre tous les messages qu'il a F-diffusés avant m. Selon le lemme 4, q F-livre m, contredisant ainsi la définition de m. Donc, un processus correct F-livre tous les messages qu'il F-diffuse.

Ordre FIFO : supposons qu'un processus p F-livre un message m avec $sender(m) = q$ et $seq\#(m) = k$. Selon l'algorithme, juste avant de F-livrer m, $next_p[q] = k$. Par le lemme 3, p a déjà F-livré les $(k-1)$ messages que q a F-diffusés avant m, comme requis.

Accord : supposons, par contradiction, que la propriété d'accord n'est pas respectée. D'où, il existe deux processus p et q tels que p F-livre un message que q ne F-livre pas. Soit m ce message de plus petit numéro de séquence. Puisque p F-livre m, alors il a précédemment R-livré m. Selon la propriété d'accord de la diffusion fiable, q R-livre aussi m. Par ailleurs, par la propriété d'ordre FIFO respecté par l'algorithme, p a précédemment F-livré tous les messages que $sender(m)$ a F-diffusés avant m. Par le choix de m, q F-livre aussi tous ces messages. Par le lemme 4, q F-livre m, qui est une contradiction.

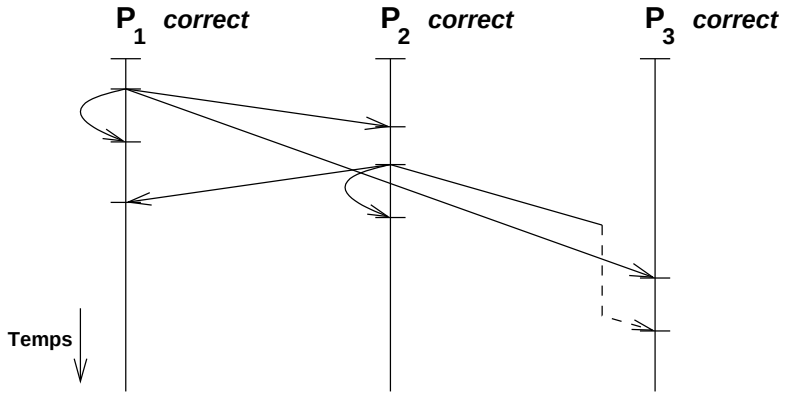
Intégrité : supposons que p F-livre m avec $sender(m) = q$ et $seq\#(m) = k$. Selon l'algorithme, p a précédemment R-livré m. Par la propriété d'intégrité de la diffusion fiable, q a R-diffusé m. Par conséquent, q doit avoir aussi F-diffusé m. Par ailleurs, quand p F-livre m, il incrémente $next_p[q]$ de k à $k+1$. Puisque $next_p[q]$ ne décroît jamais et doit être égal à k pour que m soit F-livré, m n'est pas F-livré à nouveau. □

29

3.4 Diffusion causale

■ Diffusion fiable

\+ Ordre causal : avant qu'un processus p ne livre un message m_2, p doit livrer tous les messages m_1 causalement précédents ($m_1 \rightarrow m_2$)



■ Diffusion causale inclut diffusion FIFO

La diffusion FIFO est pertinente et suffisante lorsque le contexte du message m concerne uniquement des messages précédemment diffusés par l'émetteur p (p_1 dans la figure) avant la diffusion de m. Cependant, la diffusion d'un message et son contenu peuvent aussi dépendre des messages précédemment livrés par l'émetteur p (p_2 dans la figure) avant la diffusion de m. Dans ce cas, l'ordre FIFO sur les diffusions n'est plus suffisant. Prenons un exemple. Dans une application de diffusion de nouvelles, si le lecteur de nouvelles livre les articles dans l'ordre FIFO, le scénario indésirable qui suit peut survenir. L'utilisateur U_1 diffuse l'article m_1. L'utilisateur U_2 lit l'article m_1 livré par son lecteur respectant l'ordre FIFO, puis répond à m_1 en diffusant l'article m_2. L'article m_2 ne peut être compris qu'après réception de l'article m_1. Le lecteur de l'utilisateur U_3 peut très bien, toujours en respectant l'ordre FIFO, livrer l'article m_2 en provenance de U_2 avant l'article m_1 en provenance de U_1 : U_3 ne comprend pas le message m_2 sans le contexte, qui est représenté ici par le message m_1, qui n'a pas encore été livré. La diffusion causale renforce l'ordre FIFO en empêchant ce scénario par la généralisation de la notion de dépendance.

Avant de définir la propriété d'ordre causale de la diffusion, précisons la propriété de précédence causale entre messages[1] :

- soit m_1 et m_2 deux messages, $m_1 \rightarrow m_2$ (m_1 précède causalement m_2) si et seulement si :

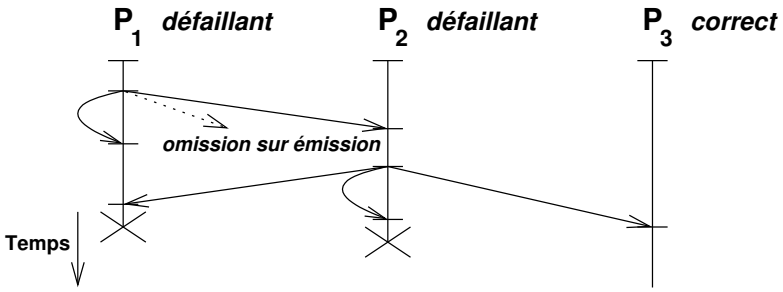
  un processus p diffuse m_1 avant de diffuser m_2,
  $\vee$ un processus p livre m_1 avant de diffuser m_2,
  $\vee$ il existe un message m_3 tel que $m_1 \rightarrow m_3 \wedge m_3 \rightarrow m_2$.

La diffusion causale est une diffusion fiable qui satisfait la propriété d'ordre causal suivante :

- ordre causal : avant qu'un processus p ne livre un message m_2, p doit livrer tous les messages m_1 causalement précédents (selon la relation de précédence causale entre messages $\rightarrow$).

Par analogie avec la diffusion FIFO, la définition de la diffusion causale est plus subtile qu'elle n'y paraît. Dans un système réparti sujet à défaillances, une définition (complète) telle que la suivante est erronée : « si la diffusion du message m_1 précède causalement la diffusion du message m_2 alors tout processus correct qui livre les deux messages doit livrer m_1 avant m_2 ». En fait, cette dernière définition autorise le même scénario que précédemment dans lequel un processus défaillant diffuse m_1, m_2 puis m_3, et un processus correct livre m_1 puis m_3. Cette définition autorise aussi le scénario indésirable présenté dans la figure qui suit. Un processus défaillant P_1 diffuse m_1. Un processus défaillant P_2, qui est le seul à livrer m_1, diffuse une réponse m_2. P_1 et P_2 défaillent par arrêt franc. Un processus correct P_3 livre m_2 sans livrer m_1. Le problème vient de ce que m_1 disparaît avec la défaillance par arrêt franc de P_1 et P_2.

1. La définition de la précédence causale donnée dans l'introduction concernait tous les évènements (internes [à un processus], et émissions et réceptions de message). Dans cette section, la définition considère uniquement les échanges de messages.


P1 défaillant
P2 défaillant
P3 correct
omission sur émission
Temps

30

3.4.1 Algorithme de diffusion causale construit à partir d'un algorithme de diffusion FIFO

```
1  Every process p executes the following :
2    Initialisation :
3      prevDlvrs := ∅ /* sequence of messages that p C-delivered since its previous C-broadcast */
4    To execute broadcast(C, m) :
5      broadcast(F, ⟨prevDlvrs + m⟩)
6      prevDlvrs := ∅
7    upon deliver(F, ⟨m_1, m_2, ..., m_l⟩) for some l do
8      forall i from 1 to l do
9        if p has not previously executed deliver(C, m_i) then
10         deliver(C,m_i)
11         prevDlvrs := prevDlvrs + m_i
```

Pour diffuser causalement un message, un processus utilise la diffusion FIFO en insérant, au lieu du seul message habituel, la liste des messages causalement livrés depuis la dernière diffusion causale. L'opérateur « + » est l'opérateur de concaténation de messages dans la séquence de messages $prevDlvrs$. Lorsqu'un processus F-livre une telle séquence de messages, il C-livre tous les messages de la séquence qu'il n'a pas encore C-livrés. Le principe de l'algorithme consiste donc à ajouter au message que le processus diffuse tout l'historique qui précède causalement ce message, et à livrer les messages reçus en parcourant la séquence complète pour éviter les « trous » de causalité.

Lemme 5 *Supposons qu'un processus p F-diffuse la séquence $prevDlvrs' + m'$, et que soit q a précédemment F-livré la séquence $prevDlvrs + m$ soit m appartient à la séquence $prevDlvrs'$. Alors, aucun processus (correct ou défaillant) ne C-livre m' à moins qu'il n'ait précédemment C-livré m.*

Preuve : La preuve est par contradiction. Supposons que les hypothèses sont vérifiées et qu'un processus C-livre m' et ne C-livre pas m avant m'. Soit p ce premier processus. Deux cas sont à considérer quant à la raison pour laquelle p C-livre m' :

1. p F-livre la séquence $prevDlvrs' + m'$. Selon les hypothèses, deux sous-cas sont à considérer. Dans le premier sous-cas, le processus q F-livre la séquence $prevDlvrs + m$ avant la séquence $prevDlvrs' + m'$. Par la propriété d'ordre FIFO de la diffusion FIFO, p doit avoir F-livré la séquence $prevDlvrs + m$ avant la séquence $prevDlvrs' + m'$. Dans le second sous-cas, m appartient à la séquence $prevDlvrs'$. Dans les deux sous-cas, selon l'algorithme, p a C-livré m avant m', la contradiction.
2. p F-livre un message $prevDlvrs'' + m''$ tel que m' appartient à la séquence $prevDlvrs''$ et m n'est pas placé avant m' dans la séquence $prevDlvrs''$. Soit $q = sender(\langle prevDlvrs'' + m'' \rangle)$. Puisque m' appartient à la séquence $prevDlvrs''$, q a C-livré m' avant de F-diffuser la séquence $prevDlvrs'' + m''$. Par la propriété d'uniformité de la diffusion FIFO, p a F-livré tous les messages précédemment F-diffusés par q. Puisque p n'a pas C-livré m avant m', m n'était pas inclus dans ces messages précédemment F-diffusés. Par ailleurs, comme m n'est pas placé avant m' dans la séquence $prevDlvrs''$, lorsque q C-livre m', q n'a pas précédemment C-livré m. Alors, puisque q C-livre m' avant m, cela contredit la définition de p.

□

Théorème 7 *L'algorithme présenté dans cette diapositive transforme tout algorithme de diffusion FIFO en un algorithme de diffusion causale.*

Preuve : Validité : supposons que p est correct et C-diffuse m. Selon l'algorithme, p F-diffuse un message contenant la séquence $prevDlvrs + m$. Par la propriété de validité de la diffusion FIFO, p à terme F-livre la séquence $prevDlvrs + m$. En conséquence, selon l'algorithme, p C-livre m.

Accord : supposons que p est correct et C-livre m. Selon l'algorithme, p F-livre un message contenant la séquence $\langle m_1, m_2, ..., m_l \rangle$ qui contient m. Par la propriété d'accord de la diffusion FIFO, tous les processus corrects F-livrent cette séquence *in fine*, et donc à terme, C-livrent m.

Intégrité : selon l'algorithme, un processus C-livre un message m seulement s'il n'a pas précédemment executé deliver(C,m). Ainsi, un processus C-livre m au plus une fois. Considérons le premier processus qui C-livre m. Il doit avoir F-livré la séquence $prevDlvrs + m$. Par la propriété d'intégrité de la diffusion FIFO, un processus p doit avoir F-diffusé la séquence $prevDlvrs + m$. Cela implique aussi que p a C-livré m.

Ordre causal : premièrement, supposons qu'un processus q C-diffuse m avant m'. Selon l'algorithme, q F-diffuse la séquence $prevDlvrs + m$ avant la séquence $prevDlvrs' + m'$. Par le lemme 5, aucun processus ne C-livre m' avant d'avoir C-livré m. Deuxièmement, supposons qu'un processus q C-livre m avant de C-diffuser m', et que p C-livre m'. Nous devons montrer que p C-livre m avant m'. Soit m'' le premier message que q C-diffuse après avoir C-livré m (m'' peut être m'). Quand q C-diffuse m'', il F-diffuse la séquence $prevDlvrs'' + m''$. Selon la définition de m'' et l'algorithme, la séquence $prevDlvrs''$ contient m. D'où, par le lemme 5, p C-livre m avant m''. Si les C-diffusions de m'' et m' sont les mêmes, nous obtenons l'ordre causal : q C-livre m avant de C-diffuser m' et p C-livre m avant m'. Sinon, q C-diffuse m'' avant m'. Par la propriété d'ordre FIFO montrée en premier dans le paragraphe, p C-livre m'' avant m'. Donc, p C-livre m avant m', comme requis. □

3.4.2 Algorithme de diffusion causale à base d'horloge vectorielle de Birman et Joseph, 1987

31

```
1  Every process p executes the following :
2    Initialisation :
3      msgBag := ∅ ; ∀i ∈ Π, v_p[i] = 0 /* bag of messages not already C-delivered ; vector clock */
4    To execute broadcast(C, m) :
5      broadcast(R, ⟨m, v_p⟩)
6      deliver(C, m)
7      v_p := v_p + 1_p
8    upon deliver(R, ⟨m, v⟩) do
9      if sender(m) ≠ p then
10       msgBag := msgBag ∪ {⟨m, v⟩}
11       while ∃⟨m_s, v_s⟩ ∈ msgBag : (∀k, v_p[k] ≥ v_s[k]) do
12         deliver(C, m_s)
13         v_p := v_p + 1_s ;
14         msgBag := msgBag \ {⟨m_s, v_s⟩}
```

Comme les processus maintiennent une horloge vectorielle, deux mises en œuvre des dépendances causales à base de vecteurs sont possibles :

- dépendances directes : estampilles des messages contenant un scalaire ;
- dépendances indirectes : estampilles des messages contenant un vecteur. C'est cette mise en œuvre qui est utilisée dans l'algorithme.

Dans l'algorithme de Birman et Joseph, 1987, remarquez que l'horloge d'un processus n'est incrémentée qu'après soit la diffusion fiable soit la livraison causale. D'où, l'entrée $v_p[k]$ représente l'horloge scalaire du prochain message attendu par p en provenance de k. Ainsi, la condition $\forall k, v_p[k] \geq v_s[k]$ signifie que le processus p attend le $v_p[0]^{\text{è}}$ message de p_0 ou le $v_p[1]^{\text{è}}$ message de p_1, ou encore le $v_p[2]^{\text{è}}$ message de p_2, etc.

Théorème 8 *L'algorithme présenté dans cette diapositive est un algorithme de diffusion causale.*

Preuve : Validité : supposons que p est correct et C-diffuse m. Selon l'algorithme, p R-diffuse $\langle m, v\rangle$ et ensuite R-livre immédiatement $\langle m, v\rangle$.

Accord : supposons, par contradiction, que la propriété de l'accord n'est pas respectée. Il existe alors deux processus corrects p et q tels que p C-livre un message que q ne C-livre pas. Considérons la séquence de messages que p C-livre, dans l'ordre dans lequel ils sont C-livrés. Soit m le premier message de cette séquence que q ne C-livre pas. Selon l'algorithme, puisque p C-livre m, il a précédemment R-livré $\langle m, v\rangle$ avec un certain vecteur v. Par la propriété d'accord de la diffusion fiable, q R-livre aussi à terme $\langle m, v\rangle$. Si cette R-livraison intervient quand la condition $(\forall k, v_q[k] \geq v[k])$ est vérifiée alors q C-livre immédiatement m, une contradiction selon le choix de m. Donc, nous pouvons supposer que, lorsque q R-livre $\langle m, v\rangle$, nous avons la condition $(\exists k, v_q[k] \not\geq v[k])$ et que $\langle m, v\rangle$ est (et reste) dans l'ensemble $msgBag$ dans l'attente d'une future C-livraison. Soit v_p la valeur du vecteur d'horloge de p lorsque p a C-livré m. Selon l'algorithme, nous avons $(\forall k, v_p[k] \geq v[k])$. Clairement, lorsque p C-livre m, il a précédemment C-livré exactement $v_p[r]$ messages qui ont été C-diffusés par r, et ce pour tous les processus r. Par la définition de m, q à terme C-livre ces messages. Par conséquent, la condition $(\forall k, v_q[k] \geq v_p[k])$ est ultimement vérifiée. Considérons le message m' dont la C-livraison par q provoque la première vérification de la condition $(\forall k, v_q[k] \geq v_p[k])$. Quand q C-livre m', $\langle m, v\rangle$ est déjà dans $msgBag$. Immédiatement après la C-livraison de m', nous avons $(\forall k, v_q[k] \geq v_p[k] \geq v[k])$, et donc q C-livre m à la prochaine itération du while de la ligne 11 de l'algorithme, ce qui contredit la définition de m.

Intégrité : par la propriété d'intégrité de la diffusion fiable, il est aisé de voir qu'un processus C-livre un message m seulement si $sender(m)$ a précédemment C-livré m. Il reste à démontrer qu'aucun processus p ne C-livre m deux fois. Pour ce faire, nous montrons que p n'insère pas les messages $\langle m, v\rangle$ et $\langle m, v'\rangle$, pour $v \neq v'$, dans $msgBag$. Trivialement, par la propriété d'intégrité de la diffusion fiable, ces deux messages

auraient été R-diffusés par le même processus $q = sender(m)$ à deux instants différents. D'où, q aurait C-diffusé m deux fois, ce qui contredit la définition de m (chaque message est identifié de manière unique par l'identité de l'émetteur et par un numéro de séquence).

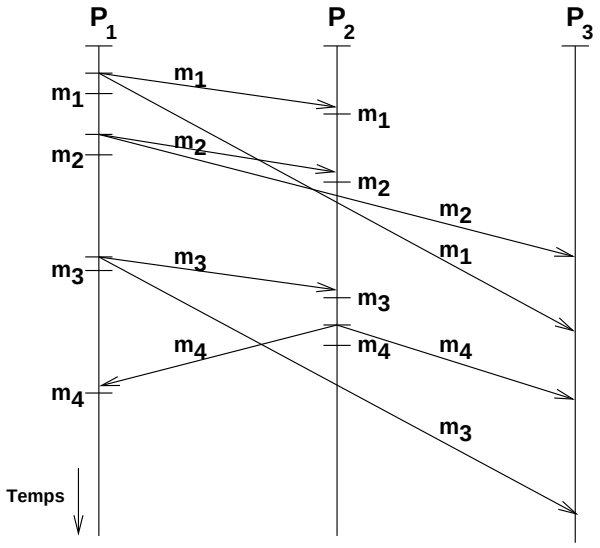
Ordre causal : selon la propriété de précédence causale entre messages, trois cas sont à considérer :

1. un processus q diffuse m avant de diffuser m' : montrons que p C-livre m avant m'. Selon l'algorithme, q R-diffuse $\langle m, v\rangle$ avant de R-livrer $\langle m', v'\rangle$ avec $v'[q] = v[q] + 1$. Toujours selon l'algorithme, p doit avoir R-livré $\langle m', v'\rangle$ avec $(\forall k, v'[k] \geq v[k])$. Par la propriété d'accord, p doit avoir R-livré $\langle m, v\rangle$ et $\langle m', v'\rangle$ avec $(\forall k, v'[k] \geq v[k])$. Selon l'algorithme, si p C-livre m sans le bloquer (c.-à-d. sans mettre $\langle m, v\rangle$ et le laisser dans $msgBag_p$) alors p C-livre m avant m'. Si p laisse $\langle m, v\rangle$ dans $msgBag_p$ avant de R-livrer $\langle m', v'\rangle$, deux cas sont à considérer. Dans le premier cas, quand p R-livre $\langle m', v'\rangle$, il bloque m' en insérant et laissant $\langle m', v'\rangle$ dans $msgBag_p$. Puisque, premièrement, p C-livre tous les messages de $msgBag_p$ en provenance du même processus q dans l'ordre de $v_q[p]$, deuxièmement, $\langle m, v\rangle$ est avant $\langle m', v'\rangle$ selon cet ordre, troisièmement, m' est par définition C-livrable, et quatrièmement, m' est par définition C-livré, nous concluons que m est C-livrable et qu'il est C-livré avant m'. Dans le second cas, quand p R-livre $\langle m', v'\rangle$, il C-livre m' immédiatement. D'où, quand cette R-livraison intervient, nous avons $(\forall k, v_p[k] \geq v'[k])$. Considérons le message m'' dont la C-livraison par p est la première à valider la condition $(\forall k, v_p[k] \geq v'[k])$. Selon le scénario dans lequel nous sommes (p a bloqué m ou n'a pas bloqué m'), la C-livraison de m'' intervient après que p R-livre $\langle m, v\rangle$ et avant que p R-livre $\langle m', v'\rangle$. Par conséquent, quand p C-livre m'', $\langle m, v\rangle$ est déjà dans $msgBag_p$. Puisque la C-livraison de m'' signifie que nous avons $(\forall k, v_p[k] \geq v'[k]) \wedge (\forall k, v'[k] \geq v[k])$, selon l'algorithme, p C-livre m lors de l'analyse de $msgBag_p$ immédiatement après la C-livraison de m''. Donc, p C-livre m avant m'. Cette première partie est par la suite appelée la propriété d'ordre FIFO de l'algorithme ;
2. un processus p livre m_1 avant de diffuser m_2 : supposons que q C-livre m avant de C-diffuser m'. Par la propriété d'intégrité démontrée ci-avant, m a été C-diffusé par un processus s. Supposons que m était le $k^{\text{è}}$ message C-diffusé par s. Par la propriété d'ordre FIFO de l'algorithme démontrée ci-avant, quand q C-livre m, c'est le $k^{\text{è}}$ message de s que q a C-livré. Donc, immédiatement après la C-livraison de m, nous avons $v_q[s] = k$. Aussi, quand q plus tard C-diffuse m', il R-diffuse $\langle m', v'\rangle$ avec $v'[s] \geq k$. Supposons que le processus p C-livre m'. Cela arrive après que p ait R-livré $\langle m', v'\rangle$ avec $(\forall k, v_p[k] \geq v'[k])$. Donc, quand p C-livre m', nous avons $v_p[s] \geq v'[s] \geq k$, et par conséquence, p a déjà C-livré au moins k messages C-diffusés par s. Par la propriété d'ordre FIFO de l'algorithme démontrée ci-avant, la séquence inclut les k premiers messages C-diffusés par s, qui incluent m. Donc, p C-livre m avant m' ;
3. il existe un message m_3 tel que $m_1 \rightarrow m_3 \wedge m_3 \rightarrow m_2$: la preuve de ce troisième scénario s'établit aisément à partir des deux premiers.

□

3.4.3 Exercice

- Donnez une exécution de l'algorithme de Birman et Joseph 1987



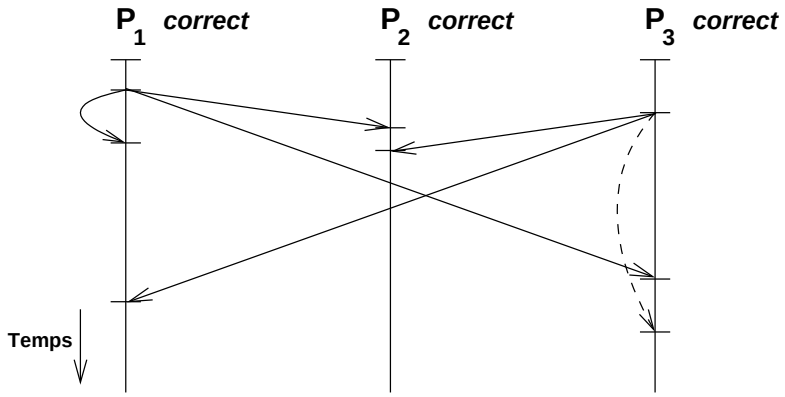
32

3.5 Diffusion atomique (ou totale)

33

■ Diffusion fiable

\+ Ordre total : Soient P_1 et P_2 deux processus qui livrent un message m_2. Si P_1 livre un message m_1 avant m_2, alors P_2 livre aussi m_1 avant m_2



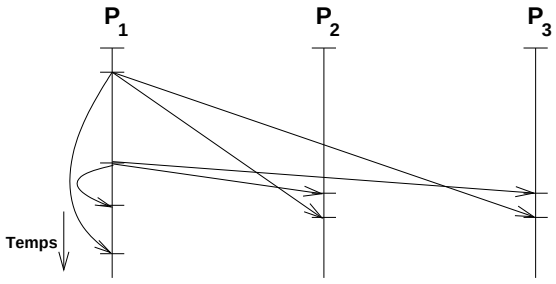
La diffusion causale « n'améliore » pas l'ordre de livraison des messages qui ne sont pas reliés causalement. Par conséquent, deux processus peuvent livrer deux messages non reliés causalement dans des ordres différents. Cette situation peut créer le problème suivant. Considérons une application possédant une base de données répliquée. Soit le compte bancaire représenté par la variable x. Initialement, x est égal à 100. Un utilisateur dépose 20 euros, provoquant une diffusion du message m_1 « ajouter 20 à x » vers les deux instances de la base de données. Au même moment, à un nœud différent du système réparti, la banque calcule les intérêts du compte x et diffuse le message m_2 « ajouter 10% d'intérêt à x ». Parce que ces deux diffusions ne sont pas reliées causalement, la diffusion causale permet que les deux copies de x soient mises à jour différemment, créant ainsi une incohérence dans la base de données : soit $(100 + 20) \times 1,1 = 132$, soit $(100 \times 1,1) + 20 = 130$.

Pour éviter de tels problèmes, la diffusion atomique requiert que tous les processus corrects livrent tous les messages dans le même ordre. Cet ordre total assure que tous les processus corrects possèdent la même vue du système réparti ; ainsi, ils peuvent agir de manière cohérente sans communication supplémentaire. Formellement, une diffusion totale est une diffusion fiable qui satisfait en plus l'ordre total :

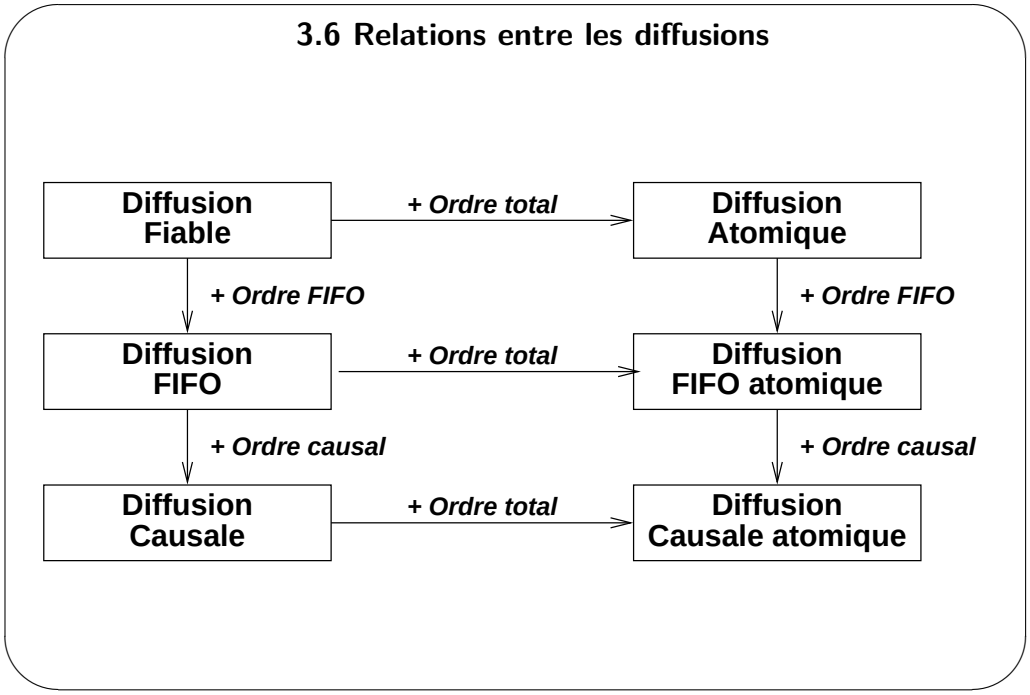
- ordre total : soient P_1 et P_2 deux processus qui livrent un message m_2. Si P_1 livre un message m_1 avant m_2, alors P_2 livre aussi m_1 avant m_2.

Les propriétés d'accord et de validité de la diffusion fiable impliquent que les processus corrects livrent ultimement la même séquence de messages.

La diffusion atomique n'impose pas que les messages soient livrés dans un ordre FIFO. Par exemple, dans la figure qui suit, si un processus P_1 est sujet à des réordonnancements locaux de messages, un message m_1 diffusé par P_1 peut ne pas être livré par les processus corrects avant le message m_2 aussi diffusé par P_1, mais après m_1. La figure qui suit montre un tel exemple de diffusion atomique (tous les processus livrent la même séquence de messages) non FIFO.



34



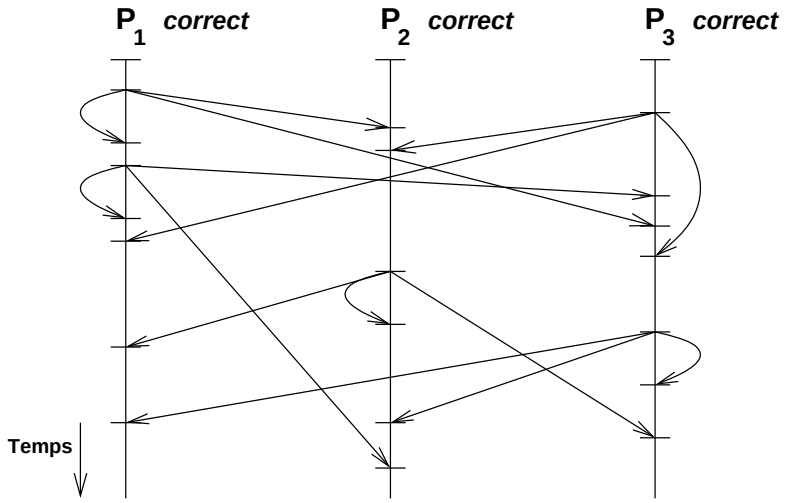
Toutes les diffusions étudiées dans cette section possèdent les propriétés de validité, accord et intégrité. Elles ne diffèrent que par l'ordre de livraison des messages : FIFO, causal et total. Nous obtenons le shéma présenté dans cette diapositive.

35

3.7 Exercice

? Vérifiez, et si besoin, transformez l'exécution suivante pour qu'elle respecte successivement les ordres FIFO, causal et total

♦ Contrainte : on ne peut que retarder la livraison d'un message



Comme pour les autres exercices demandant l'exécution d'un algorithme réparti, cet exercice possède de nombreuses solutions. Vous devez modifier le diagramme temporel avec la contrainte de ne pouvoir que retarder la réception des messages. Nous vous proposons de commencer par modifier le diagramme pour respecter l'ordre FIFO, puis de continuer avec le respect de l'ordre causal, et enfin, de terminer avec l'ordre total. Lorsque vous ajoutez l'ordre causal après l'ordre FIFO, vous continuez de respecter l'ordre FIFO, puisque l'ordre causal implique l'ordre FIFO. Par contre, lorsque vous ajoutez l'ordre total, faites attention à ne pas « casser » l'ordre causal.

36

3.8 Diffusion atomique et consensus *

- Consensus :
  - Ensemble de processus devant s'accorder pour prendre une décision
    - Terminaison : tous les processus corrects décident ultimement d'une valeur
    - Validité : si tous les processus qui proposent une valeur proposent v, tous les processus corrects décident ultimement v
    - Accord : si un processus décide v, tous les processus corrects décident ultimement v
- Diffusion atomique : rappel :
  - Diffusion fiable
  - \+ Ordre total : Si deux processus corrects P_1 et P_2 livrent tous les deux m_1 et m_2, alors P_1 livre m_1 avant m_2 si et seulement si P_2 livre m_1 avant m_2
- Équivalence entre les deux problèmes :
  - Consensus : diffusion atomique de v pour proposer une valeur et par exemple prendre v du premier message
  - Diffusion atomique : diffusion fiable d'un message et consensus sur l'ordre de réception

Il existe des cas où un ensemble de processus doivent s'accorder pour prendre une décision : c'est le consensus. Tous les processus doivent être au courant de la décision ou, autrement dit, doivent décider. Le consensus peut alors être vu comme la propagation d'une demande de décision, puis, la diffusion des informations de décision, et enfin, la décision.

Un algorithme de consensus satisfait les quatre propriétés suivantes :

- terminaison : tous les processus corrects décident ultimement d'une valeur ;
- validité : si tous les processus qui proposent une valeur proposent v, tous les processus corrects décident ultimement v ;
- accord : si un processus décide v, tous les processus corrects décident ultimement v.

Tout algorithme de consensus est un algorithme de diffusion. En effet, les processus prêts à s'accorder démarrent l'exécution de l'algorithme : il y a donc potentiellement plusieurs initiateurs. La demande de décision doit visiter tous les processus (comme dans l'algorithme de diffusion). Toutes les actions `décide()` sont précédées de la réception d'une demande de décision et de la réception des informations de décision.

3.8.1 Résultat d'impossibilité du consensus *

37

- Système (complètement) asynchrone
- Canaux de communication fiables et défaillances par arrêts francs possibles
- Pas d'ordre FIFO
- Pas de mécanisme pour détecter les fautes de processus : impossibilité de savoir si transmission très lente ou défaillance par arrêt franc du processus
- Preuves : tous les algorithmes possèdent une « fenêtre de vulnérabilité » — un intervalle de temps durant l'exécution pendant lequel le délai ou l'inaccessibilité d'un processus peut causer l'attente indéfinie de l'algorithme
- Cela ne signifie pas que le problème n'a pas de solution en pratique, mais :
  - ∨ Nécessité d'être plus réaliste que le modèle asynchrone : faire des hypothèses sur les délais
  - ∨ Nécessité de relâcher les propriétés du consensus, par exemple, avec les algorithmes non déterministes
- Corollaire : résultat d'impossibilité de la diffusion atomique

Dans un système réparti asynchrone, il est impossible d'obtenir un consensus entre processus répartis dès qu'un seul d'entre eux est défaillant par arrêt franc (en anglais, *crash*). Cette impossibilité résulte de la difficulté de dire si le processus distant est juste lent ou s'il est défaillant. Pourtant, beaucoup de mécanismes de tolérance aux fautes nécessitent de telles décisions, ne serait-ce que pour distinguer de façon cohérente l'ensemble des processus corrects de ceux qui sont défaillants à un instant donné.

Pour pallier ce résultat d'impossibilité, des solutions de tolérance aux fautes utilisent des détecteurs de défaillances non fiables ou des algorithmes aléatoires (non déterministes). La première solution consiste à faire l'hypothèse optimiste qu'une fenêtre favorable (un intervalle de temps) existera à terme telle que les processus détermineront de manière sûre l'ensemble des processus défaillants et les processus corrects ne se suspecteront pas (plus), mais décideront et donc qu'un consensus sera atteint. La seconde solution estime une probabilité d'obtention de la solution ou une probabilité d'attente d'une solution correcte.

Le consensus étant théoriquement équivalent (en terme de difficulté) à la diffusion atomique, le résultat d'impossibilité est aussi valable pour la diffusion atomique : il est impossible d'obtenir une diffusion atomique dès qu'un seul d'entre eux est défaillant par arrêt franc.

3.8.2 Algorithmes de diffusion temporisée *

38

- Exemple de la diffusion fiable temporisée
  - En ajoutant les hypothèses suivantes :
    - f processus au plus peuvent défaillir par arrêt franc
    - Graphe fortement connecté et diamètre (entre processus corrects) d
    - Borne des durées de transmission entre deux nœuds δ
    - Durée d'exécution d'une action interne nulle
    - Système synchronisé : horloges physiques locales synchronisées à ϵ près, et dérive par rapport au temps réel borné à ρ
  - $\Rightarrow$ Algorithme de diffusion fiable :
    - Aucun processus correct ne livre un message m après $\Delta = (f + d)\delta(1 + \rho) + (f + 1)\epsilon$
- Sur le même principe, diffusions atomique, FIFO atomique et causale atomique temporisées

La diffusion FIFO atomique est définie comme étant une diffusion fiable qui satisfait en plus les propriétés de l'ordre FIFO et de l'ordre total. Voici un algorithme de diffusion atomique temporisée construit sur le principe selon lequel « temporisé » signifie aussi « livraison contrainte dans le temps » :

```
1  Every process p executes the following :
2    To execute broadcast(A^Δ, m) :
3      broadcast(R^Δ, m)
4    deliver(A^Δ, m) occurs as follows :
5      upon deliver(R^Δ, m) do
6        schedule deliver(A^Δ, m) at time ts(m) + Δ
```

Toujours en suivant le même principe, nous pouvons concevoir des algorithmes de diffusion FIFO atomique (ou FIFO totale) temporisée et de diffusion causale atomique (ou causale totale) temporisée.

```
1  Every process p executes the following :
2    Initialisation :
3      msgBag := ∅ /* set of messages that p A^Δ-delivered but not yet FA-delivered */
4      next[q] := 1 for all q /* sequence number of next message from q that p will FA-deliver */
5    To execute broadcast(FA, m) :
6      broadcast(A^Δ, m)
7    deliver(FA, m) occurs as follows :
8      upon deliver(A^Δ, m) do
9        q := sender(m)
10       msgBag := msgBag ∪ {m}
11       while ∃n ∈ msgBag : (sender(n) = q) ∧ (seq#(n) = next[q]) do
12         deliver(FA,n)
13         next[q] := next[q] + 1
14         msgBag := msgBag − {n}
```

La diffusion causale atomique est définie comme étant une diffusion fiable qui satisfait en plus les propriétés de l'ordre causal et de l'ordre total. La diffusion causale atomique inclut la diffusion FIFO atomique. C'est ce type de diffusion qui est le plus utilisé dans les approches de tolérance aux fautes par réplication à base de machines à états. Voici un algorithme de diffusion causale atomique (ou causale totale) temporisée :

```
1  Every process p executes the following :
2    Initialisation :
3      prevDlvrs := ∅ /* set of messages that p CA-delivered since its previous CA-broadcast */
4      suspects := ∅ /* processes that p suspects to be faulty */
5    To execute broadcast(CA, m) :
6      broadcast(FA, ⟨m, prevDlvrs⟩)
7      prevDlvrs := ∅
8    deliver(CA, m) occurs as follows :
9      upon deliver(FA, ⟨m, D⟩) do
10       if sender(m) ∉ suspects ∧ p has previously executed deliver(CA, m′) for all m′ ∈ D then
11         deliver(CA,m)
12         prevDlvrs := prevDlvrs ∪ {m}
13       else /* either p or sender(m) is faulty */
14         discard(m)
15         suspects := suspects ∪ {sender(m)}
```

39

3.9 Propriété d'uniformité *

Qu'en est-il des processus défaillants ?

- Accord uniforme : si un processus (***correct ou non***) livre un message de diffusion m alors tous les processus corrects livrent ultimement m
- Intégrité uniforme : pour chaque message de diffusion m, tout processus (***correct ou non***) livre m au plus une fois, et seulement si m a été antérieurement diffusé par l'émetteur de m
- FIFO uniforme : si un processus diffuse un message m_1 avant de diffuser un message m_2, alors aucun processus (***correct ou non***) ne livre m_2 à moins qu'il n'ait déjà livré m_1
- Causal uniforme : si la diffusion d'un message m_1 précède causalement la diffusion d'un message m_2 alors aucun processus (***correct ou non***) ne livre m_2 avant d'avoir livré m_1
- Total uniforme : Si deux processus (***corrects ou non***) P_1 et P_2 livrent tous les deux m_1 et m_2, alors P_1 livre m_1 avant m_2 si et seulement si P_2 livre m_1 avant m_2

Les propriétés d'accord, d'intégrité et d'ordre définies jusqu'ici n'imposent aucune restriction dans les livraisons de messages par les processus défaillants. Par exemple, la propriété d'accord autorise un processus défaillant à livrer un message qui n'est livré par aucun processus correct.

Dans le cas des fautes bénignes, par exemple les arrêts francs, il est possible de définir pour chaque diffusion une version dite « uniforme » en remplaçant les propriétés initiales par leur forme « uniforme » :

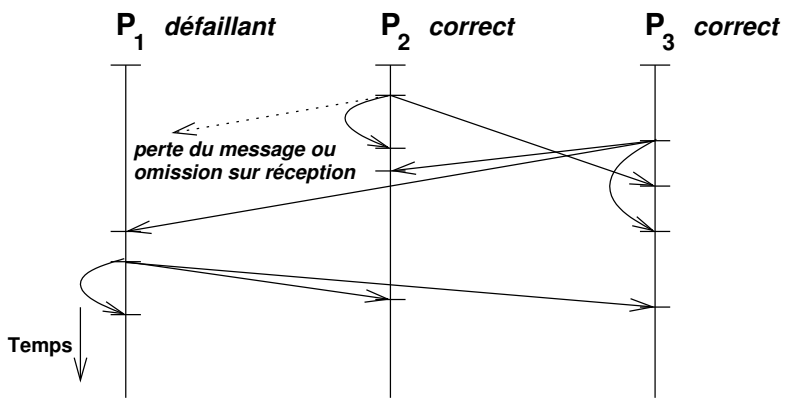
- accord uniforme : si un processus (*correct ou non*) livre un message de diffusion m alors tous les processus corrects livrent ultimement m ;
- intégrité uniforme : pour chaque message de diffusion m, tout processus (*correct ou non*) livre m au plus une fois, et seulement si m a été antérieurement diffusé par l'émetteur de m ;
- FIFO uniforme : si un processus diffuse un message m_1 avant de diffuser un message m_2, alors aucun processus (*correct ou non*) ne livre m_2 à moins qu'il n'ait déjà livré m_1 ;
- causal uniforme : si la diffusion d'un message m_1 précède causalement la diffusion d'un message m_2 alors aucun processus (*correct ou non*) ne livre m_2 avant d'avoir livré m_1 ;
- total uniforme : Si deux processus (*corrects ou non*) P_1 et P_2 livrent tous les deux m_1 et m_2, alors P_1 livre m_1 avant m_2 si et seulement si P_2 livre m_1 avant m_2.

Avec les diffusions normales (non uniformes), la sémantique de l'accord requiert uniquement que les processus corrects livrent le même ensemble de messages, quelque soit les messages reçus par les processus défaillants. En particulier, un processus peut livrer un message et défaillir juste après, sans qu'aucun autre processus ne livre ce message (tous « décident » de ne pas livrer le message, pour une raison quelconque). L'uniformité est typiquement importante lorsque les processus interagissent avec le monde extérieur, par exemple, en affichant un résultat interprété par des utilisateurs ou en actionnant des vannes dans une usine. Dans ce cas, le fait que le processus défaillant reçoive un message est important, même si la défaillance intervient juste après, ceci parce que juste après la réception du message et avant la défaillance, le processus peut avoir communiqué avec le monde extérieur. Les processus corrects doivent aussi être au courant que ce dernier message reçu avant la défaillance a effectivement été reçu, et que l'interaction avec le monde extérieur a été effectuée.

Remarquons que l'uniformité n'empêche pas un processus de ne pas livrer un message, mais s'il le livre, cette dernière propriété précise qu'il respecte tel ou tel ordre. En outre, dans l'approche modulaire organisant les dépendances entre les différents types de diffusion, les transformations préservent la propriété d'uniformité. Ainsi, si l'algorithme de diffusion de base, la diffusion fiable, est uniforme, les autres le sont aussi.

3.10 Inconsistance et contamination *

40



Cas de la diffusion atomique (uniforme ou non)

Supposons qu'un processus P_1 est défaillant par omission sur réception. À la suite d'une faute dans la livraison d'un message m_1, l'état de P_1 est inconsistant. Si P_1 continue à s'exécuter et livre un message m_2 que tous les processus corrects livrent, le contenu de m_2 est corrompu puisqu'il a été généré dans un état inconsistant. Par conséquent, la livraison de m_2 contamine tous les processus corrects.

Même si les propriétés de la diffusion sont uniformes, la contamination est possible. Prenons un exemple avec la diffusion atomique uniforme. Une variable x répliquée sur trois processus est initialement égale à 5. Le processus P_2 diffuse une instruction pour incrémenter x et P_3 une instruction pour doubler x. Les processus P_2 et P_3 sont corrects et livrent dans l'ordre les instructions pour incrémenter puis doubler x. Leur valeur de x est maintenant 12. Cependant, le processus P_1 est défaillant et omet l'instruction d'incrémentation ; P_1 est inconsistant et la valeur de x est 10. Puisque P_1 est défaillant, cette exécution satisfait la spécification de la diffusion atomique uniforme. Une fois que P_1 est inconsistant, il peut diffuser un message basé sur son état inconsistant et contaminer tous les processus corrects. Par exemple, P_1 diffuse la valeur d'une variable y calculée à partir de x. Les processus corrects mettent alors à jour leur variable y et deviennent inconsistants.

Clairement, éviter que les processus défaillants deviennent inconsistants, ou au moins, qu'ils ne contaminent pas les processus corrects, est désirable dans de nombreuses situations. Heureusement, c'est possible avec toutes les diffusions présentées ci-avant pour toutes les fautes bénignes. Intuitivement, un processus peut éviter d'être contaminé en refusant de livrer les messages pour lesquelles les livraisons ne sont pas compatibles avec les siennes. La quantité d'information que chaque message doit transporter, de telle façon que tout processus détermine s'il peut livrer ce message, dépend du type de diffusion et des types de fautes. Ce travail de détection est cependant difficile et coûteux en terme de quantité d'information transportée. Pour faire court, cela requiert des techniques pour qu'un processus défaillant détecte s'il est en train de commettre une erreur en essayant de livrer un message, et si c'est le cas, le processus doit immédiatement s'arrêter. Les techniques de détection du respect des ordres FIFO, causal et total, sont fondées entre autres sur les horloges logiques.

4 Exclusion mutuelle

41

Cette section sur la gestion de l'exclusion mutuelle est reprise des références suivantes :

- L. Lamport, *Time, clocks and the ordering of events in a distributed system*, Communications of the ACM, 21(7), July 1978.
- G. Ricart et A.K. Agrawala, *An Optimal Algorithm for Mutual Exlusion in Computer Networks*, Communications of the ACM, Volume 24, Numéro 1, pp. 9–17, Janvier 1981.
- O.S.F. Carvalho et G. Roucairol, *On mutual exclusion in computer networks*, Communications of the ACM, Volume 26, Numéro 2, pp. 145–147, Février 1983.
- G. Ricart et A.K. Agrawala, *Authors' response to "On mutual exclusion in computer networks"*, Communications of the ACM, Volume 26, Numéro 2, pp. 147–148, Février 1983.
- I. Suzuki and T. Kasami, *A distributed mutual exclusion algorithm*, ACM Transactions on Computer Systems, 3(4):344–349, November 1985.
- B.A. Sanders, *The Information Structure of Distributed Mutual Exclusion Algorithms*, ACM Transactions on Computer Systems, Volume 5, Numéro 3, pp. 284–299, Août 1987.
- M. Raynal, *A simple taxonomy for distributed mutual exclusion algorithms*, ACM SIGOPS Operating Systems Review, Volume 25 Numéro 2, pp. 47–50, Avril 1991.

Il existe de nombreuses façons de synchroniser des processus. Le premier concept, qui est aussi le plus utilisé, est l'exclusion mutuelle : à tout instant, il peut y avoir 0 ou 1 processus qui accède à une ressource ; nous disons, pour celui qui accède à la ressource, qu'il est entré en section critique. Ensuite, nous trouvons des concepts dérivés tels que les sémaphores qui sont une généralisation du concept d'exclusion mutuelle.

Dans d'autres cas, l'exclusion mutuelle est trop stricte ; c'est pourquoi nous autorisons plusieurs processus à entrer en section critique simultanément s'ils ne font que consulter la ressource sans la modifier. Dès qu'une demande de modification est effectuée, nous revenons dans le cadre de l'exclusion mutuelle au sens strict initial du terme. Ce mécanisme s'appelle la synchronisation avec plusieurs lecteurs / un écrivain (paradigme aussi appelé producteur / consommateurs).

NB : nous profitons de cette section pour montrer comment une structure informationnelle[1] permet de concevoir ce que nous appellons dans cette section un « algorithme générique » et que nous pouvons aussi voir comme un « méta-algorithme », c'est-à-dire un algorithme à instancier ou décliner en une multitude de cas particuliers. Cette approche est à mettre en regard par exemple de l'approche qui consiste à décliner un principe de conception sur des topologies particulières (par exemple, anneau ou arbre) pour exprimer une solution.

1. Dans le cas de l'exclusion mutuelle, les informations dont nous avons besoin sont celles qui proviennent de la réponse aux deux questions suivantes : à qui dois-je demander l'autorisation pour entrer en section critique ? et, qui dois-je informer lorsque je sors de section critique ?

4.1 Propriétés

42

1. Sûreté (en anglais, *safety*) : à tout instant, il ne peut y avoir au plus qu'un seul processus en section critique
2. Vivacité (en anglais, *liveness*) : tout processus désirant entrer en section critique y arrive en un temps fini

Les problèmes d'exclusion mutuelle sont légions dans la vie courante. Prenons un exemple très simple. Quand nous créditons notre compte en banque, ce crédit ne doit pas être perdu parce qu'en parallèle la banque débite un chèque. Dans le cours sur les concepts des systèmes d'exploitation que vous suivez, vous apprenez à formuler ce problème comme ceci : « une ressource ne doit être accessible que par une entité à un instant donné ». Nous disons aussi que la ressource doit être accédée dans une « section critique ». Dans ce cas, la ressource est une zone mémoire contenant le solde d'un compte. Dans ce cours, la formulation est la même, mais les demandeurs ne s'exécutent pas sur la même machine : l'algorithme doit être réparti.

Un algorithme d'exclusion mutuelle réparti est un algorithme qui satisfait les deux propriétés suivantes :

1. sûreté (en anglais, *safety*) : à tout instant, il ne peut y avoir au plus qu'un seul processus qui est en section critique ;
2. vivacité (en anglais, *liveness*) : tout processus désirant entrer en section critique y arrive en un temps fini.

Les deux grandes familles de solutions sont les algorithmes à base de permissions et les algorithmes à base de jeton. Les deux familles se rejoignent lorsqu'il y a un coordinateur qui centralise les livraisons des permissions ou du jeton.

4.2 Algorithmes à base de permissions

43

Le principe des algorithmes à base de permissions est le suivant. Lorsqu'un processus désire entrer en section critique, il demande à une partie des processus leur permission et attend que toutes les permissions demandées arrivent. Lorsqu'un processus reçoit une demande de permission, s'il n'est pas intéressé par la section critique, il envoie une permission au processus demandeur. En outre, un mécanisme permet d'établir un ordre de priorité pour entrer dans la section critique. Dans la littérature, la priorité est établie soit à l'aide d'un estampillage (« à la » Lamport) des demandes, soit en organisant un graphe acyclique direct des demandes avec les processus étant les arcs et les demandes les sommets.

La propriété de sûreté (l'exclusion mutuelle n'est jamais violée) est assurée en obtenant un nombre suffisant de permissions. La propriété de vivacité (chaque demande est satisfaite) est assurée en ordonnant (totalement) les demandes.

4.2.1 Structure informationnelle générique de Sanders, 1987

44

- État d'un processus : $dans\_sc$, $hors\_sc$, en_att pour « dans », « hors de », « en attente à l'entrée de » la section critique
- I_p : ensemble des processus que le processus p doit « informer » lors de la transition de $hors\_sc$ à en_att, et de la transition de $dans\_sc$ à $hors\_sc$
- P_p : ensemble des processus à qui le processus p doit demander les permissions
  - Pour qu'un algorithme soit correct, $I_p \subseteq P_p$
- F_p : file d'attente ordonnée des processus en attente de permissions et pour lesquels p a reçu une demande
- id_sc : identité du processus dans la section critique, $\perp$ si aucun
- M_p : processus pour lequel le processus courant sait quand ils sont en section critique
  - $p \in I_q \wedge q$ entre en section critique $\Rightarrow$ à terme (après un temps fini) $q \in M_p$

Les processus de l'application sont hors de la section critique, en attente pour entrer dans la section critique ou dans la section critique. La conception d'un algorithme d'exclusion mutuelle consiste donc à définir le protocole de coordination pour entrer dans la section critique. Ce protocole contient les variables maintenues par chaque processus, le code exécuté avant l'entrée de la section critique, le code exécuté lors de la sortie, et enfin, le code exécuté lors d'une demande d'entrée par un autre processus. Ce sont ces informations qui définissent la structure informationnelle de l'algorithme. Ainsi, ce sont les informations maintenues par chaque processus qui varient d'une solution à une autre : les processus connaissent plus ou moins l'état des autres processus suite à l'échange de plus ou moins de messages. Voic la structure informationnelle utilisée par tous les algorithmes :

- l'état d'un processus peut être $dans\_sc$, $hors\_sc$, en_att pour, respectivement, « dans la section critique », « hors de la section critique », « en attente à l'entrée de la section critique » ;
- l'ensemble I_p des processus que le processus p doit « informer » lors la demande d'entrée (de $hors\_sc$ à en_att) et lors de la sortie de section critique (de $dans\_sc$ à $hors\_sc$) ;
- l'ensemble P_p des processus à qui le processus p doit demander les « permissions » pour entrer en section critique ;
- la file d'attente ordonnée F_p des processus en attente de permissions et pour lesquels p a reçu une demande ;
- l'identité id_sc du processus dans la section critique : $\perp$ lorsqu'aucun processus n'est en section critique.
- l'ensemble M_p des processus pour lequel le processus courant sait quand ils sont entrés en section critique. Nous avons donc $p \in I_q \wedge q$ entre en section critique $\Rightarrow$ à terme (après un temps fini) $q \in M_p$. En effet, par la définition de l'ensemble P_p et de la relation $I_p \subseteq P_p$, la relation $q \in M_p$ est vraie à terme lors de l'entrée en section critique de p, et par la définition de I_p, la même relation est vraie à terme lors de la sortie de section critique de p.

Dans la suite, la politique d'entrée dans la section critique est « premier arrivé, premier servi », basée sur la date de la demande. Pour résoudre les conflits, puisqu'il n'y a pas d'horloge physique globale, chaque processus maintient une horloge logique de Lamport. Les conflits sont résolus en choisissant le processus avec l'horloge logique la plus petite, puis le processus de plus petit identifiant. Ce sont ces règles qui sont utilisées pour ordonner F_p.

45

4.2.2 Algorithme générique de Sanders, 1987

- À l'entrée :
  1. p choisit une estampille et envoie des demandes à tous les processus dans P_p avec l'estampille
  2. p attend les permissions, puis entre en section critique
- En sortie :
  1. p émet un message de sortie à tous les processus de I_p
- Lors de la réception d'une demande de permission de q :
  1. q placé dans F_p et mise à jour de l'estampille de p
  2. Si $id\_sc = \perp$
     (a) Envoi d'une permission au « premier » processus α de F_p et retrait de α de F_p
        i. Si $\alpha \in M_p$ alors $id\_sc = \alpha$
- Lors de la réception d'un message de sortie de q :
  1. $id\_sc := \perp$
  2. Tant que $id\_sc = \perp \vee F_p \neq \emptyset$
     (a) Envoi d'une permission au « premier » processus α de F_p et retrait de α de F_p
        i. Si $\alpha \in M_p$ alors $id\_sc = \alpha$

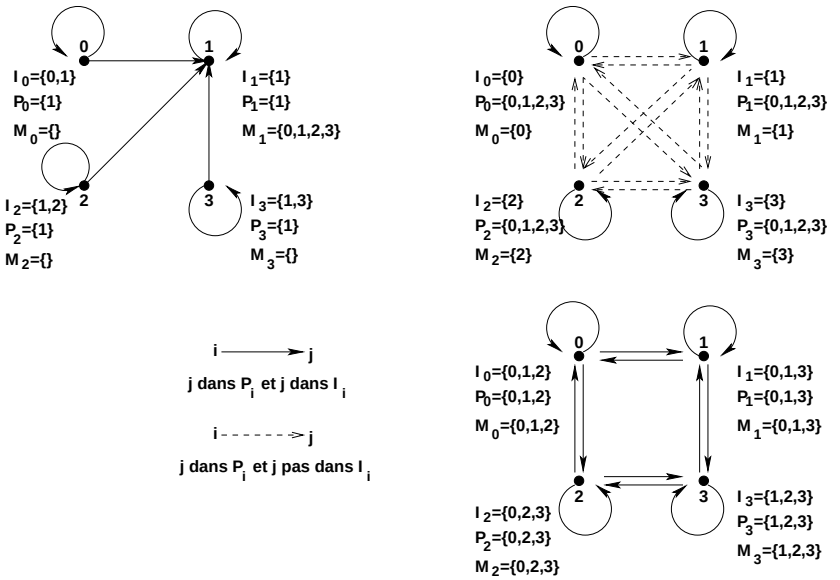
L'algorithme générique de Sanders, 1987 suppose que les communications sont fiables (aucun message n'est perdu) et respectent l'ordre FIFO (les messages en provenance d'un même émetteur et à destination d'un même récepteur sont reçus dans l'ordre de leur émission)[1]. Si ces hypothèses ne sont pas respectées, l'algorithme réparti peut se bloquer et empêcher de rentrer en section critique.

Les demandes sont estampillées avec le numéro de processus et une horloge logique scalaire. Chaque processus enregistre l'horloge logique la plus grande connue (des demandes reçues et ses propres demandes) et choisit une estampille pour une nouvelle demande qui est calculée comme la plus grande connue plus un. Lors de la réception d'une demande de permission, le « premier » processus de F_p est celui qui a l'horloge logique la plus petite, avec l'utilisation de l'identifiant du processus le plus petit en cas d'égalité.

1. Par construction, les canaux de communication TCP respectent l'ordre FIFO, contrairement aux canaux de communication UDP.

4.2.3 Quelques algorithmes (dérivés de l'algorithme générique)

46



La figure illustre trois algorithmes de la littérature en termes de structure informationnelle. Une flèche pleine de i vers j représente la relation $j \in P_i \wedge j \in I_i$ (i doit demander la permission à j et i doit informer j de ses changements d'états) alors qu'une flèche pointillée de i vers j représente la relation $j \in P_i \wedge j \notin I_i$ (i doit demander la permission à j et i n'informe pas j de ses changements d'états). Le premier algorithme est centralisé : le processus 1 arbitre les demandes concurrentes et distribue les permissions. Dans le deuxième schéma, l'algorithme est complètement réparti : chaque processus demande la permission à tous les autres processus avant de rentrer dans la section critique. Enfin, le dernier algorithme est construit de façon à ce que chaque processus demande la permission à un sous-ensemble des processus tel que, pour toute paire de processus i et j, les ensembles I et P soient égaux et $I_i \cap I_j \neq \emptyset$.

Si les algorithmes présentés dans cette page sont corrects, il est à noter que toutes les configurations ne donnent pas une solution correcte. En fait, pour qu'un algorithme soit correct, les deux conditions suivantes doivent être respectées pour tous les processus :

1. $p \in I_p$ et $I_p \subseteq P_p$;
2. $\forall p, q\ I_p \cap I_q \neq \emptyset$ ou $p \in P_q \wedge q \in P_p$.

La première condition requiert qu'un processus p demande une permission à tous les processus de I_p avant de rentrer en section critique. La conséquence est que tous les processus dans I_p sont toujours au courant des changements d'états de p. La première partie de la seconde condition indique qu'il existe un processus r tel que p et q informent r à la sortie de la section critique, et avec la première condition, p et q demandent aussi à r une permission avant de rentrer en section critique. La seconde partie de la seconde condition signifie que p et q se demandent mutuellement une permission avant de rentrer en section critique. Enfin, ces conditions sont nécessaires et suffisantes.

Lemme 6 *Si $q \in I_p$ alors q exécute l'affectation id_sc := p avant que p n'entre en section critique et exécute l'affectation id_sc := $\perp$ lorsque p quitte la section critique.*

Preuve : Provient directement de la description de l'algorithme et de la relation $I_p \subseteq P_p$. □

Théorème 9 *Conditions suffisantes.*

Preuve : Considérons une paire de processus p et q. La première partie de la seconde condition indique qu'il existe un processus r tel que p et q informent r à la sortie de la section critique, et avec la première condition, p et q demandent aussi à r une permission avant de rentrer en section critique. Par conséquent, selon le lemme 6, *id_sc* indiquera toujours que q est dans la section critique quand q est effectivement en section critique, et donc l'exclusion mutuelle est garantie.

La seconde partie de la seconde condition signifie que p et q se demandent mutuellement une permission avant de rentrer en section critique. Une concurrence entre p et q peut survenir de deux manières différentes. Dans le premier cas, p et q choisissent une estampille avant de recevoir la requête de l'autre processus. Le processus à plus faible priorité, c'est-à-dire dont l'estampille est la plus grande, enverra une permission au processus possédant la plus forte priorité, c'est-à-dire dont l'estampille est la plus petite. Le processus à plus forte priorité n'enverra pas de permission tant qu'il ne sera pas entré puis sortie de la section critique. Dans le second cas, supposons que q décide d'entrer en section critique après avoir reçu une demande de p puis envoyé une permission à p. L'estampille choisie par q sera toujours supérieure à celle de p. Donc, p n'enverra pas de permission à q avant sa sortie de section critique. Ainsi, dans les deux cas, l'exclusion mutuelle est garantie. □

Théorème 10 *Conditions nécessaires.*

Preuve : La nécessité de la propriété 1 est évidente dans le contexte de l'algorithme générique : I_p contient les processus informés de l'entrée et de la sortie de section critique, p connaît son état et donc appartient à I_p ; un processus p demande une permission à tous les processus de I_p avant d'entrer en section critique, avec pour conséquence que tous les processus dans I_p sont toujours au courant des changements d'états de p.

La nécessité de la propriété 2 est montrée en considérant toutes les situations possibles non incluses dans la propriété et en démontrant que l'exclusion mutuelle peut être violée dans ces cas. Étant donné les relations $I_p \subseteq P_p$, $I_q \subseteq P_q$ et $I_p \cap I_q = \emptyset$, les relations suivantes sont mutuellement exclusives et couvrent toutes les situations possibles :

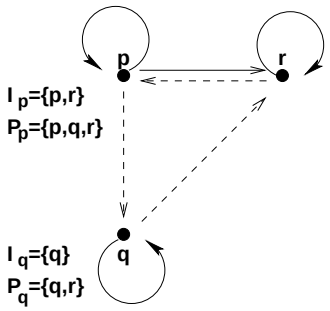
1. $P_p \cap P_q = \emptyset$;
2. $P_p \cap I_q = \emptyset$ et $I_p \cap P_q = \emptyset$, mais $P_p \cap P_q \neq \emptyset$;
3. $P_p \cap I_q = \emptyset$, mais $I_p \cap P_q \neq \emptyset$;
4. $P_q \cap I_p \neq \emptyset$ et $P_p \cap I_q \neq \emptyset$. Ce cas peut être décomposé en trois sous-cas :
   (a) $p \in P_q \cap I_p$ et $q \in P_p \cap I_q$;
   (b) $r \in P_q \cap I_p$ et $r \in P_p \cap I_q$, avec $r \neq p$ et $r \neq q$ (et la situation symétrique obtenue en permutant p et q) ;
   (c) $r \in P_q \cap I_p$ et $s \in P_p \cap I_q$, avec $r \neq s$ et tous les deux différents de p et q.

La situation 4a satisfait la propriété 2 comme indiqué dans le théorème. Pour les situations 4b et 4c, nous donnons des contre-exemples montrant que l'exclusion mutuelle peut être violée. Quant aux situations 1 à 3, nous montrons que l'exclusion mutuelle peut être violée.

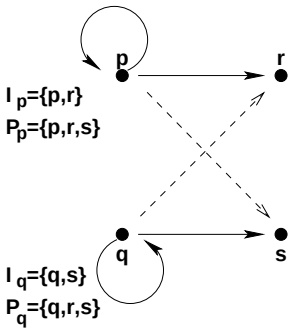
La situation 4b est montrée dans la figure (a) ci-dessous. Les ensembles I_p et I_q sont respectivement égaux à $\{p, r\}$ et $\{q\}$. Les ensembles P_p et P_q sont respectivement égaux à $\{p, q, r\}$ et $\{q, r\}$, avec $r \neq p$ et $r \neq q$. Les informations de q à propos de l'état de p sont obtenues de manière indirecte via le processus r, tandis que les informations de p sur q sont obtenues directement. Supposons qu'aucun processus n'est dans la section critique, mais qu'approximativement au même instant, p obtient une estampille de valeur 1 et que q obtient une estampille de valeur 2. Ensuite, supposons la séquence d'actions suivantes. Une demande de q arrive à r, qui transmet aussitôt une permission à q. Une requête de p arrive à q, qui transmet une permission puisque p possède une estampille plus faible. La demande de p arrive à r, qui répond aussitôt par une permission. Lorsque la permission de p arrive à q, q entre en section critique. Au bout d'un certain temps, les permissions émises par q et r arriveront à p, et p entrera en section critique, violant ainsi l'exclusion mutuelle.

La situation 4c est montrée dans la figure (b) ci-dessous. Dans cette configuration, $I_p = \{p, r\}$, $I_q = \{q, s\}$, $P_p = \{p, r, s\}$ et $P_q = \{q, r, s\}$, avec $r \neq s$, et r et s différents de p et de q. L'exclusion mutuelle sera violée lorsque p et q demande à entrer en section critique à peu près au même instant, et la demande de p arrive à s avant la demande de q, et la demande de q arrive à r avant la demande de p.

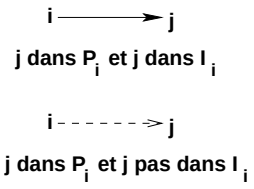
Dans les situations 1–3, les risques de violation de l'exclusion mutuelle sont encore plus importants. Dans la situation 3, l'exclusion mutuelle peut être violée si p essaie d'entrer en section critique alors que q est en section critique. Dans la situation 2, si p ou q sont en section critique, l'autre processus peut entrer aussi en section critique. Dans la situation 1, il n'existe aucune intersection entre les ensembles des processus avec lesquels p et q communiquent avant d'entrer en section critique. □


p
r
l p={p,r}
P p={p,q,r}
l q={q}
P q={q,r}
q

(a)


p
r
l p={p,r}
P p={p,r,s}
l q={q,s}
P q={q,r,s}
q
s

(b)


i
j
j dans P i et j dans l i
i
j
j dans P i et j pas dans l i

4.3 Algorithme à base de jeton de Ricart et Agrawala 1983, et de Suzuki et Kasami, 1985

47

```
1  À l'entrée de la section critique de p
2    if p n'a pas le jeton then          {sinon p déjà en section critique avec le jeton}
3      ns_p := ns_p + 1 ;
4      diffuser la demande du jeton avec ns_p et attendre le jeton ;
5  Lors de la réception d'une demande du jeton de q avec l'estampille ns_q
6    dem_p[q] := max(dem_p[q], ns_q) ;
7    if p a le jeton ∧ p n'est pas en section critique then
8      exécuter « À la sortie de la section critique de p » ;
9  À la sortie de la section critique de p
10   jet[p] := ns_p ;
11   p n'est plus en section critique ;
12   forall q de p+1, ..., n, 1, ..., p-1 do
13     if dem_p[q] > jet[q] ∧ p a le jeton then
14       envoyer le jeton (c.-à-d. le tableau jet) à q et p n'a plus le jeton ;
```

La deuxième approche pour résoudre le problème de l'exclusion mutuelle répartie se base sur la notion de privilège. Celui-ci est symbolisé par la possession d'un objet unique particulier que nous appelons un jeton. Il n'existe qu'un seul jeton dans tout le système. L'obtention de ce jeton garantit l'exclusivité de l'accès à la ressource. La propriété de vivacité est assurée en faisant en sorte que le jeton puisse se déplacer de processus en processus et parcourir tous les processus. En ce qui concerne la vivacité, deux possibilités existent : le jeton passe de lui-même de processus en processus en dessinant un anneau logique incluant tous les processus (le processus qui désire entrer en section critique garde le jeton, entre en section critique et fait suivre le jeton à sa sortie de section critique) ; le processus qui désire entrer en section critique demande le jeton et attend de le posséder avant d'entrer effectivement en section critique.

En 1983, Ricart et Agrawala ont esquissé le premier algorithme suivant cette dernière approche : G. Ricart et A.K. Agrawala, *Authors' response to "On mutual exclusion in computer networks"*, Communications of the ACM, Volume 26, Numéro 2, pp. 147–148, Février 1983. En parallèle, en 1981, Suzuki et Kasami ont proposé un algorithme proche, qui a été effectivement publié en 1985 : I. Suzuki and T. Kasami. *A distributed mutual exclusion algorithm*, ACM Transactions on Computer Systems 3(4):344–349, November 1985. La version que nous proposons dans cette diapositive est très proche de ces deux versions.

Chaque processus gère une horloge logique locale et tous les messages sont estampillés par la valeur de l'horloge au moment de l'émission. Un jeton est un message dont le type est un tableau d'estampilles jet de taille n (n étant le nombre de processus participant à l'algorithme réparti). Chaque fois qu'un processus transmet le jeton, il place la valeur de son horloge dans l'entrée qui lui correspond. Ainsi, chaque entrée du tableau du jeton contient la date du dernier passage du jeton chez le processus correspondant.

Chaque processus gère un tableau dem_p de taille n, dans lequel il range les estampilles des dernières demandes reçues. Lorsque p possédant le jeton sort de section critique, il recherche dans ce tableau, dans l'ordre $p+1, \ldots, n, 1, \ldots, p-1$, le premier processus q ayant fait une nouvelle demande, en comparant les entrées correspondantes de dem_p et de jet.

Cet algorithme est efficace car il évite les famines et réalise l'exclusion mutuelle. La famine est évitée car tous les processus parcourt le jeton dans le même ordre, et par conséquent, une fois leur demande du jeton diffusée et reçue par tous les autres processus, tous les demandeurs auront le jeton après au plus $n-1$ sections critiques. Bien que la taille du jeton puisse être élevée, cet algorithme est efficient car il nécessite relativement peu de messages : soit 0, soit $(n-1)$ messages pour la diffusion de la demande + 1 message pour la transmission du jeton.

Lemme 7 *À tout moment, il n'existe qu'un jeton, qui est soit possédé par un processus soit en transit vers un processus.*

Preuve : Au début de l'algorithme, un des processus possède le jeton. Toujours selon l'algorithme, le jeton est transmis dans deux situations : lors de la réception d'une demande du jeton par un processus p qui le

possède alors que p n'est pas en section critique, et lorsqu'un processus p sort de section critique. Ces deux situations correspondent à l'exécution des mêmes lignes 9–14 de l'algorithme. Selon la condition de la ligne 13 et selon la ligne 14, le jeton n'est transmis qu'à un seul processus. Ainsi, par les hypothèses implicites de l'algorithme selon lesquelles, au début de l'exécution, un seul processus possède le jeton et aucun jeton n'est généré pendant l'exécution, et par le fait qu'au plus un seul jeton est transmis dans la section de l'algorithme intitulée « à la sortie de la section critique », le jeton est transmis à un seul processus et le processus qui le transmet ne le possède plus. Grâce à la fiabilité des communications point à point, un seul processus reçoit le jeton. L'exécution est donc formée, pour ce qui concerne le jeton, d'une séquence de paires d'évènements $(\text{émettre}(jet), \text{recevoir}(jet))$. Dans chacune de ces configurations, il n'y a qu'un jeton, et il est soit possédé par un processus soit en transit. □

Lemme 8 *Soit d une demande d'entrée en section critique envoyée par un processus p. À terme, il existe un évènement d'envoi du jeton qui est causalement précédé par l'envoi de d.*

Preuve : Soit ns la valeur ns_p insérée dans la demande d du jeton. Selon l'algorithme, p envoie la demande d alors qu'il ne possède pas le jeton. Grâce à la fiabilité des communications point à point, la demande est reçue par tous les autres processus du système, et selon l'algorithme, il est évident que nous avons $(\forall q \neq p, dem_q[p] = ns)$ ainsi que $(jet[p] = ns - 1)$. Par le lemme 7, soit la demande de p est reçue par un processus q possèdant le jeton, soit aucun processus ne possède le jeton lors de la réception de la demande de p. Dans le premier cas, q exécute à terme (selon qu'il est ou non en section critique) la section de code « à la sortie de la section critique » dans laquelle q envoie le jeton à un demandeur, qui peut être p car $(dem_q[p] > jet[p])$. Dans le second cas, puisque les communications point à point sont fiables, le jeton est reçu à terme, et selon l'algorithme, le processus q qui a reçu le jeton entre en section critique et à terme en sort avec l'exécution de la section de code « à la sortie de la section critique » dans laquelle il envoie le jeton à un demandeur, qui peut être p car $(dem_q[p] > jet[p])$. □

Théorème 11 *L'algorithme proposé dans cette diapositive réalise une exclusion mutuelle.*

Preuve : Sûreté : par le lemme 7, il y au plus un jeton dans le système. En outre, un processus est en section critique lorsqu'il possède le jeton. Par conséquent, à tout instant, il ne peut y avoir au plus qu'un seul processus qui est en section critique.

Vivacité : procédons par contradiction. Supposons qu'un processus p demande à entrer en section critique et n'y entre jamais. Soit ns la plus petite valeur de ns_p pour laquelle p exécute la section de code « à l'entrée de la section critique » sans obtenir à terme le jeton. Par la définition de ns, pour les valeurs de ns_p inférieure à ns, p a obtenu le jeton. Comme les communications point à point sont fiables, toutes les demandes jusqu'à la valeur $(ns-1)$ incluse ont été reçues par tous les autres processus du système et ont été satisfaites. Selon l'algorithme, nous avons donc $(\forall q \neq p, dem_q[p] = ns - 1)$ ainsi que $(jet[p] = ns - 1)$. Selon l'algorithme, la demande ns de p est envoyée alors que p ne possède pas le jeton. À l'aide d'un raisonnement par récurrence sur le nombre n de processus dans le système, en utilisant le lemme 8 et selon l'algorithme, il est simple de montrer qu'il existe un chemin de causalité de taille fini contenant la séquence d'évènements émettre($demande$)...émettre(jet) et tel que p reçoive le jeton à terme, la contradiction. □

4.4 Exercice

48

- Un seul exemplaire de la ressource R
- Début de l'exécution de tous les processus à 8 h 55
- Chaque processus effectue un travail (est réllement actif) pendant une certaine durée
- P_2 a besoin de R pendant toute la durée de son exécution, les autres processus pendant seulement la seconde moitié
- Au début de l'exécution, c'est le processus P_1 qui possède le jeton

Proc.	**Démarrage**	**Démarrage du travail effectif**	**Durée du travail effectif**
P_1	8 h 55	8 h 57	6 mn
P_2	8 h 55	8 h 59	5 mn
P_3	8 h 55	9 h 08	4 mn
P_4	8 h 55	8 h 55	14 mn
P_5	8 h 55	9 h 08	2 mn

? Dessinez le diagramme temporel de l'algorithme d'exclusion mutuelle de Ricart et Agrawala, 1983, et de Suzuki et Kasami, 1985

♦ Montrez les instants d'acquisition de la ressource R et les messages transmis

À titre d'exercice, soit cinq processus dont les instants de démarrage et les durées d'exécution sont donnés dans le tableau ci-dessus. On suppose qu'il n'existe qu'un seul exemplaire de la ressource R. Le processus P_2 a besoin de R pendant toute la durée de son exécution et les autres processus pendant seulement la seconde moitié. Donnez une exécution de l'algorithme d'exclusion mutuelle de Ricart et Agrawala, 1983, et de Suzuki et Kasami, 1985. Au début de l'exécution, on suppose que *1)* c'est le processus P_1 qui possède le jeton, *2)* les horloges logiques scalaires des processus sont toutes égales à 0 et *3)* le jeton est égal au vecteur $[0, 0, 0, 0, 0]$. Montrez entre autres les instants d'acquisition de la ressource R et les messages transmis.

5 Interblocage

Cette section sur la gestion des interblocages est reprise des références suivantes :

- G. Tel, *Chapter 10 : Snapshots*, dans Introduction to Distributed Algorithms, Cambridge University Press, pp. 331–351, 1994.
- J. Brzeziński, J.-M. Hélary and M. Raynal, *A General Definition of Deadlocks for Distributed Systems*, dans Proc. 1st IEEE International Conference on Algorithms and Architectures for Parallel Processing, pp. 324–327, April 1995.
- S.J. Rosenkrantz, R.E. Stearns, and P.M. Lewis, *System Level Concurrency Control for Distributed Database Systems*, ACM Transactions on Database Systems, Vol. 3, N. 2, june 1978.
- K.M. Chandy and L. Lamport, *Distributed Snapshots : Determining Global States of Distributed Systems*, ACM Transactions on Computer Systems, 3(1) :63–75, February 1985.

Dans un système réparti, le fait que des processus de différentes machines accèdent concurremment à des ressources partagées est une cause possible de blocage. Un interblocage se produit lorsqu'un ensemble de processus est tel que chacun d'eux tient au moins une ressource, et pour poursuivre sa progression, est en attente d'une ressource tenue par l'un des autres.

Dans un certain sens, ce problème est une extension de l'exclusion mutuelle : comme dans une exclusion mutuelle, il faut éviter les famines et l'accès à une ressource simultanément par plusieurs processus. L'algorithme est réparti non seulement parce que les clients sont répartis, mais aussi parce que les serveurs de ressources sont répartis.

Les exemples d'interblocage montrés dans cette section sont volontairement simplifiés pour plus de clarté. Plus précisément, nous considérons des ressources accédées en exclusion mutuelles par des processus ; chaque ressource est contrôlée par un gestionnaire de ressource ; et chaque gestionnaire de ressource contrôle une seule ressource. Aussi, dans cette section, par abus de langage, mais sans que cela porte à confusion, nous employons le terme « ressource » aussi pour « gestionnaire de ressource ». En outre, même si nous distinguons les gestionnaires de ressources des processus, nous disons qu'un processus p dépend d'un processus q via la ressource R si p demande l'accès (en exclusion mutuelle) à R alors que R est tenue par q (qui y accède en exclusion mutuelle) : nous notons cette dépendance $p \rightarrow_R q$ et nous disons que p attend une autorisation d'accès (à R) de la part de q.

5.1 Principaux modèles d'interblocage

50

- Blocage en attente d'autorisations de la part d'autres processus
- Si pas de traitement correct, danger sérieux pour les performances, voire pour l'utilité de l'application répartie
  - Verrouillage et donc indisponibilité des ressources impliquées

- Modèle d'interblocage ET
- Modèle d'interblocage OU[a]
- Modèle d'interblocage OU–ET

a. Cf. détection de terminaison

Dans certaines applications réparties, les processus sont temporairement bloqués en attente de messages ou d'autorisations de la part d'autres processus : par exemple, des transactions dans les bases de données réparties ou des processus en compétition pour accéder à des ressources. Lorsque les interblocages surviennent et ne sont pas traités correctement, ils mettent sérieusement en danger les performances, voire l'utilité de ces applications : les ressources impliquées dans l'interblocage sont verrouillées et donc indisponibles.

Le problème de l'interblocage est le plus souvent étudié dans des modèles « simplifiés » dans lesquels les interactions entre processus sont simplifiées. Les modèles les plus courants sont :

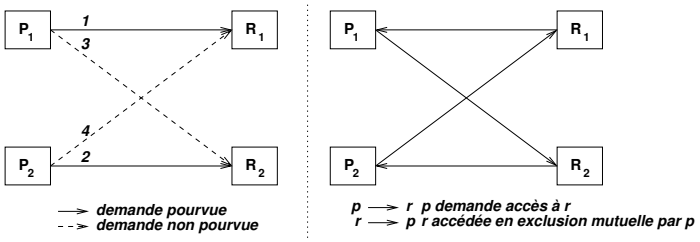
- Modèle ET dans lequel un processus doit obtenir l'autorisation de tous les autres processus ;
- Modèle OU (étudié dans la section 6) ;
- Modèle OU–ET dans lequel un processus peut choisir entre plusieurs ressources répliquées (en plusieurs exemplaires) (le « OU »), et pour la ressource choisie, il doit obtenir l'autorisation de tous les autres processus (le « ET »). Il est démontré dans la littérature que tous les modèles possibles sont compris dans le modèle OU–ET.

Nous commençons par développer les deux principaux modèles d'interblocage.

5.1.1 Modèle d'interblocage ET

51

- Hypothèse : premier arrivé, premier servi



1. P_1 demande l'accès à R_1 et accède à R_1 (c.-à-d. verrouille R_1)
2. P_2 demande l'accès à R_2 et accède à R_2
3. P_1 demande l'accès à R_2, qui est verrouillée par P_2
   - Donc, $P_1 \rightarrow_{R_2} P_2$
     « P_1 attend une autorisation d'accès à R_2 de la part de P_2 »
4. P_2 demande l'accès à R_1, qui est verrouillée par P_1
   - Donc, $P_1 \rightarrow_{R_2} P_2 \rightarrow_{R_1} P_1$
     - ♦ D'où présence d'un cycle, et par conséquent situation d'interblocage

Le fait que des processus de différentes machines accèdent concurremment à des ressources partagées est une cause possible de blocage. Par exemple, une base de données répartie consiste en une collection de fichiers dispersés sur de nombreuses machines. L'interface de la base de données permet d'accéder aux données soit pour les lire soit pour les modifier. L'accès aux données est structuré dans ce que nous appelons une transaction. La transaction nécessite l'accès à un certain nombre de données. Pour éviter que les mêmes données soient accédées par une autre transaction en même temps, la transaction protège l'accès aux données avec un verrou. Bien sûr, si une donnée est déjà accédée par une autre transaction, la première doit attendre que la dernière libère la donnée.

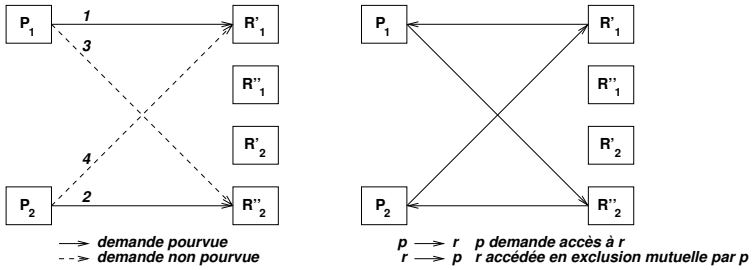
Pour être active, la transaction doit donc obtenir les verrous de toutes les données qu'elle demande. Ce modèle dans lequel les accès à toutes les ressources doivent être autorisés avant que la transaction soit active, est le modèle ET. Dans ce modèle, un interblocage se produit lorsqu'un ensemble de processus est tel que chacun d'eux tient au moins une ressource et est en attente, pour poursuivre sa progression, d'une ressource tenue par l'un des autres.

Dans la figure à gauche de la diapositive, le processus P_1 demande et obtient l'accès en exclusion mutuelle à la ressource R_1. Puis, le processus P_2 demande et obtient l'accès en exclusion mutuelle à la ressource R_2. Ensuite, P_1 demande et n'obtient pas l'accès à la ressource R_2 car elle est tenue par P_2, ce que nous notons $P_1 \rightarrow_{R_2} P_2$. De même, P_2 demande et n'obtient pas l'accès à la ressource R_1 car elle est tenue par P_1, ce que nous notons $P_1 \rightarrow_{R_2} P_2$. En conséquence, nous obtenons le cycle $P_1 \rightarrow_{R_2} P_2 \rightarrow_{R_1} P_1$. Visuellement, en changeant la syntaxe des arcs comme indiqué dans la figure à droite, nous dessinons un cycle.

52

5.1.2 Modèle d'interblocage OU–ET

- Hypothèse : premier arrivé, premier servi



1. P_1 demande l'accès à R'_1 et accède à R'_1
2. P_2 demande l'accès à R''_2 et accède à R''_2
3. $P_1 \rightarrow_{R''_2} P_2$
4. $P_2 \rightarrow_{R'_1} P_1 \Longrightarrow P_1 \rightarrow_{R''_2} P_2 \rightarrow_{R'_1} P_1 \Longrightarrow$ cycle $\Longrightarrow$ interblocage

Considérons maintenant que la base de données est répliquée ou que les données de la base de données sont répliquées. La réplication permet une meilleure disponibilité des données et incidemment une meilleure performance des transactions (une meilleure latence d'accès). Si les deux transactions ont des chances d'accéder à des tables différentes des bases de données, avec la réplication des mêmes bases de données, les transactions ont moins de risque d'entrer en concurrence pour l'accès aux données. Toutefois, même dans ces conditions optimistes, les interblocages ne sont pas toujours évités si les transactions ne se coordonnent pas comme c'est le cas dans la figure de la diapositive. Pour être active, chaque transaction peut accéder à une des répliques (le « OU ») et doit accéder à une réplique des deux données (le « ET »).

53

5.2 Condition de déblocage

- En considérant que les processus fournissent des autorisations pour les ressources qu'ils tiennent
- Soit le prédicat générique $\mathit{fulfilled}_p$ qui caractérise la condition de déblocage de p, c.-à-d. qui caractérise l'ensemble des autorisations à recevoir pour débloquer p
  - P.ex., dans le modèle ET : toutes les ressources R_i doivent être accessibles par p (non verrouillées) $\Longrightarrow$ une autorisation de tous les processus de B
  - P.ex., dans le modèle OU–ET : un sous-ensemble des ressources R_i doivent être accessibles par p $\Longrightarrow$ B est décomposé en sous-ensembles b_i d'autorisations pour des ressources « équivalentes » (OU), et une autorisation pour chaque ensemble b_i (ET)
- $\mathit{fulfilled}_p(B)$ = prédicat générique établissant si p peut être débloqué avec B un sous-ensemble de gestionnaires de ressources R_i
  - $\mathit{fulfilled}_p(B)$ doit vérifier :
    1. étant donné B l'ensemble des autorisations attendues, $(\neg \mathit{fulfilled}_p(\emptyset)) \wedge (B \neq \emptyset \Rightarrow \mathit{fulfilled}_p(B))$
    2. $\forall B, C$ ensembles d'autorisations, $(\mathit{fulfilled}_p(B) \wedge B \subseteq C) \Rightarrow \mathit{fulfilled}_p(C)$

Pour être débloqué, un processus p doit recevoir un message (une autorisation) d'utilisation en provenance d'un certain nombre de gestionnaires de ressources B.

Le prédicat $\mathit{fulfilled}_p$ est calculé lorsque p pense qu'il est bloqué ; dans ce cas, p fournit l'ensemble des ressources utilisées pour ce calcul. Dans le modèle ET, $\mathit{fulfilled}_p$ est de la forme « toutes les ressources R_i doivent être accessibles par p (non verrouillées) ». Dans le modèle OU–ET, $\mathit{fulfilled}_p$ est de la forme « un sous-ensemble des ressources R_i doivent être accessibles par p ».

Le prédicat $\mathit{fulfilled}_p$ est générique et caractérise la condition de déblocage de p. Il doit satisfaire les deux propriétés suivantes :

1. étant donné B l'ensemble des autorisations attendues, $(\neg \mathit{fulfilled}_p(\emptyset)) \wedge (B \neq \emptyset \Rightarrow \mathit{fulfilled}_p(B))$
2. $\forall B, C$ ensembles d'autorisation, $(\mathit{fulfilled}_p(B) \wedge B \subseteq C) \Rightarrow \mathit{fulfilled}_p(C)$

La première partie du premier prédicat de $\mathit{fulfilled}_p$ signifie : 1) que l'interblocage est une propriété stable impliquant qu'un processus en interblocage ne peut pas en sortir sans recevoir de permissions, et 2) qu'il faut au moins une autorisation pour débloquer p ($B \neq \emptyset$). La seconde partie du premier prédicat indique que si tous les gestionnaires de ressources de B donnent leur autorisation alors p ne peut pas rester bloqué. Le second prédicat exprime la monotonie de la propriété : p est toujours débloqué s'il reçoit plus d'autorisations que celles de l'ensemble suffisant B ($B \subseteq C$) ; autrement dit, les nouvelles autorisations ne remettent pas en cause la décision de débloquer p.

54

5.3 Définition de l'interblocage

- Définitions préléminaires :
  - Π : ensemble des processus
  - Les messages transmis sont des autorisations d'accès (des relâchements de ressources)
  - B : ensemble des processus possiblement « interbloqués » et sur lesquels la condition d'interblocage est testée, avec $B \subseteq \Pi$ et $B \neq \emptyset$
  - $passive_p = true \equiv p$ en attente de ressources
  - ET_p (En Transit) : ensemble des processus q tels qu'il existe des messages en transit de q vers p, c'est-à-dire tous les messages en provenance de q ne sont pas disponibles sur le site de p
  - ARR_p (ARRivé) : ensemble des processus q tels que des messages reçus de q sont disponibles, c'est-à-dire arrivés, mais non encore traités par p
  - Pour rappel, $fulfilled_p$: prédicat générique établissant si p peut être débloqué
    - Pour être débloqué, p doit recevoir une autorisation d'utilisation d'un certain nombre de processus
- Définition :
  - $deadlock(B) \stackrel{\text{def}}{=} \forall p \in B : passive_p \wedge \neg fulfilled_p(ET_p \cup ARR_p \cup (\Pi \setminus B))$

En connaissant le prédicat $fulfilled_p$, il est possible de définir de manière abstraite l'interblocage. En conséquence, il est possible de concevoir une solution valable pour tous les modèles. La détection d'interblocage avec un algorithme de coupure cohérente (voir plus loin à la fin de cette section) est un exemple d'algorithme générique utilisant cette définition de l'interblocage.

Les définitions préliminaires sont nécessaires car le système est asynchrone. Des messages (autorisations) peuvent être en transit entre les processus : l'ensemble ET_p est l'ensemble des processus en provenance desquels des autorisations sont en transit vers p. D'autres peuvent être arrivés sur le site du récepteur sans avoir été traités par ce dernier : l'ensemble ARR_p est l'ensemble des processus q pour lesquels l'autorisation est arrivée sur le site, mais n'est pas encore traitée par p. Par conséquent, un ensemble de processus B est en interblocage si tous les processus de B sont en attente de ressources et si la condition de déblocage est fausse. La condition de déblocage est évaluée sur l'ensemble des ressources pour lesquelles toutes les autorisations ne sont pas reçues ou arrivées, et sur ceux n'appartenant pas à B : aucun processus appartenant à B ne peut être activé même s'il reçoit tous les messages en transit ou non encore traité en provenance des processus de B ou des messages en provenance des autres processus du système ($\Pi \setminus B$), et ceci car ces messages ne sont pas suffisants pour le débloquer ($fulfilled_p$ est faux).

Pratiquement, la formulation de la détection d'un problème d'interblocage peut s'exprimer selon les trois formes suivantes :

- détection d'un interblocage : est-ce que la condition $deadlock(\Pi)$ est vraie ?
- détection d'un processus en interblocage : étant donné un processus p, est-ce que p est en interblocage, c'est-à-dire existe-t-il un ensemble B tel que $p \in B \wedge B \subseteq \Pi \wedge deadlock(B) = true$?
- détection d'un ensemble de processus en interblocage : existe-t-il un ensemble $B \subseteq \Pi$ tel que $deadlock(\Pi) = true$?

5.4 Trois stratégies contre l'interblocage

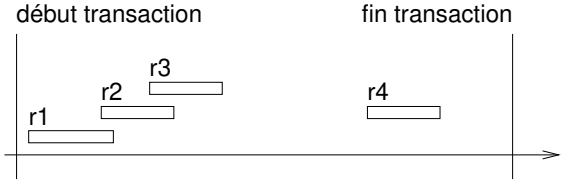
55

- Prévention : contraintes sur la manière de demander l'accès aux ressources :
  - Schéma 1 : réservations de toutes les ressources avant de commencer
  - Schéma 2 : relâchement de toutes les ressources avant d'accéder à de nouvelles
  - Schéma 3 : ordonnancement des ressources et accès dans leur ordre
- Évitement :
  - Contrôle pas à pas de la progression des processus
    Avec mise en œuvre d'une possibilité d'échapper à un interblocage
    Donc, solution dépendante des algorithmes des utilisateurs ⇒ non étudié ici
- Détection puis résolution :
  - Interblocage est une propriété stable ⇒ algorithme de détection
    1. Non-interférence : ne pas influencer l'exécution de l'algorithme de l'utilisateur
    2. Vivacité (en anglais, *liveness*) : si interblocage, le détecter en un temps fini
    3. Sûreté (en anglais, *safety*) : détecter un interblocage seulement lorsqu'il en existe un
  - Résolution : dépendant des algorithmes des utilisateurs ⇒ non étudiée ici

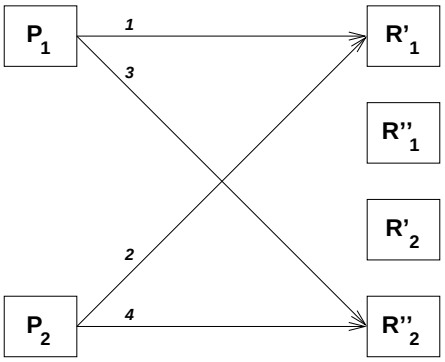
Il existe trois méthodes pour gérer l'interblocage : la prévention, l'évitement et la détection. En outre, il ne faut pas oublier que les solutions doivent éviter les famines.

La **prévention** des interblocages répond à la question « comment empêcher l'occurrence ou l'introduction d'un interblocage ». Elle prend beaucoup de formes, la plupart d'entre-elles imposant des contraintes sur la manière de demander l'accès aux ressources. Il existe trois schémas de base :

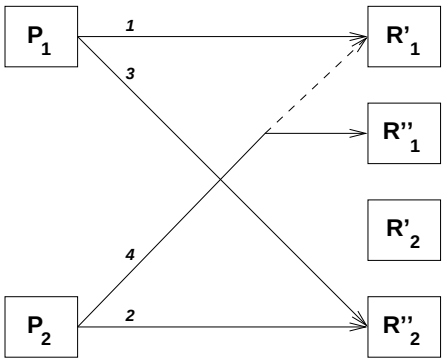
1. toutes les ressources doivent être réservées avant de commencer la transaction. Cf. par exemple l'algorithme de Rosenkrantz, Stearns et Lewis dans les pages suivantes pour le modèle ET. La difficulté réside dans le fait qu'il est souvent difficile de déterminer a priori cet ensemble à cause des dépendances pas toujours très bien connues entre les données.



   Dans la figure précédente, les ressources accédées pendant la transaction (r_1, r_2, r_3 et r_4) sont demandées avant de commencer la transaction et sont bloquées pendant toute la durée de la transaction. L'une des difficultés de cette méthode est de délimiter les début et fin d'une transaction. Par conséquent, cette solution est souvent impraticable (hors programmation avec des transactions, par exemple SQL).
2. un processus doit relâcher toutes ses ressources avant d'accéder à de nouvelles. Cela implique que les ressources devant être accédées de façon concomitante doivent être demandées ensemble. Dans la situation de la figure précédente, il est possible de réserver les trois premières ressources (r_1, r_2 et r_3), puis de relâcher leur réservation, avant de réserver r_4. Globalement, le problème consistant à définir la séquence de réservations des ensembles de ressources, est pratiquement aussi difficile à résoudre que celui consistant à délimiter les début et fin d'une transaction ;
3. les ressources sont ordonnées. Les processus doivent les accéder dans l'ordre croissant des identifiants de ressources. Ainsi, pour le cas étudié ci-avant et dont la figure est remise ci-après, deux processus accédent à deux ressources partagées et il est possible de construire l'exécution comme suit. P_1 et P_2 demandent l'accès d'abord à R'_1 ou R''_1 puis à R'_2 ou R''_2. P_1 obtient le verrou sur R'_1, P_2 est bloqué en attente de la même ressource. Ainsi, P_1 peut obtenir l'accès à R''_2 et travailler. Lorsque P_1 relâche ses deux ressources, P_2 peut accéder à R''_2. Remarquez que si les files d'attente des ressources respectent l'ordre FIFO, cette solution évite la famine.



Les méthodes d'**évitement**, en contrôlant pas à pas la progression des processus, font en sorte qu'à chaque étape d'attribution de ressources, il reste une possibilité au système d'échapper à un interblocage. Cela implique la maintenance d'une quantité importante d'informations sur les processus et les ressources sous la forme d'un graphe. Un algorithme d'évitement des interblocages détermine à partir du graphe la séquence d'actions qui permet à chaque processus de se terminer. Les processus peuvent alors s'exécuter jusqu'à ce que des ressources soient indisponibles, auxquels cas ils se mettent en attente des ressources. Dans les systèmes repartis, le graphe est réparti et la collecte des informations est très coûteuse. Par conséquent, cette méthode n'est généralement pas utilisée. Si nous reprenons l'exemple des deux processus accédant à deux ressources partagées, avant de rentrer en interblocage, P_2 pourrait décider de ne pas accéder à R'_1, mais à R''_1, la difficulté étant de déterminer comment il le sait. Lorsque P_2 termine son travail et relâche ses ressources, P_1 est débloqué et travaille à son tour.



Dans un système réparti utilisant la **détection d'interblocage**, les processus sont autorisés à attendre indéfiniment l'acquisition d'une ressource. L'interblocage est une propriété stable : si une configuration s est bloquée[1], toutes les configurations atteignables à partir de s sont bloquées (si on ne résoud pas l'interblocage). Il faut donc exécuter (en arrière plan) un algorithme de détection d'interblocage, et lorsqu'un interblocage est détecté, exécuter un algorithme de résolution.

Un algorithme de détection d'interblocage doit satisfaire les trois propriétés suivantes :

1. non-interférence : il ne doit pas influencer l'exécution de l'algorithme de l'utilisateur ;
2. vivacité (en anglais, *liveness*) : si un interblocage survient, il doit être détecté en un temps fini ;
3. sûreté (en anglais, *safety*) : il doit détecter un interblocage seulement lorsqu'il en existe un.

Quand aux algorithmes de résolution, ils sont très dépendants des algorithmes des utilisateurs. Par conséquent, nous n'en présentons pas dans la suite de l'exposé.

1. Attention ! Ne pas confondre plus tard une configuration bloquée avec une configuration terminale.

5.5 Prévention dans le modèle ET avec l'algorithme de Rosenkrantz, Stearns et Lewis, 1978

- Attribution unique d'un numéro par demande d'accès pour éviter la famine
  - Ci-dessous et dans l'exercice : date d de (première) création du processus
    $\Longrightarrow$ Maximum un accès par processus

56

1. « Attendre ou mourir » :
   - p demande une ressource tenue par q
   - $d_p < d_q \rightarrow$ attente de la libération de cette ressource
   - $d_p \geq d_q \rightarrow p$ effectue un retour au début de l'exécution (mort), libère toutes ses ressources et recommence son exécution à partir du début
2. « Blesser ou attendre » (solution duale de « Attendre ou mourir ») :
   - p demande une ressource tenue par q
   - $d_p \leq d_q \rightarrow p$ provoque le retour en arrière (mort) de q et q libére toutes ses ressources pour recommencer son exécution à partir du début
   - $d_p > d_q \rightarrow$ attente de la libération de cette ressource

L'interblocage implique une attente circulaire. Si les règles d'acquisition des ressources par les processus font qu'une attente circulaire ne peut jamais apparaître, il est certain qu'il ne se produira jamais d'interblocage.

Rosenkrantz, Stearns et Lewis, 1978 attribuent un numéro de priorité, non pas aux ressources, mais aux processus, en supposant un modèle dans lequel un processus ne peut exécuter qu'une seule section critique. Ci-dessous, c'est la date de première création du processus qui est utilisée, avec « première création » à différencier des autres créations suites aux retours en arrière. En outre, lorsque plusieurs processus sont en concurrence et exécutent l'algorithme au même instant, nous utilisons la date de création pour établir l'ordre de « priorité » pour déterminer celui qui gagne l'accès à ses ressources (demandées).

Dans la première solution, lorsqu'un processus p demande une ressource tenue par q, p n'attend la libération de cette ressource que si la première création de p est antérieure à celle de q. Sinon, p n'est pas autorisé à attendre : il effectue un retour en arrière, libère les ressources qu'il détient et recommence son exécution à partir du début. Cette solution reçoit le nom de « attendre ou mourir ». Le risque de famine est évité sous la condition qu'un processus victime d'un retour en arrière conserve son estampille : c'est le cas avec l'utilisation de la date de première création. Après plusieurs retours en arrière, un processus devient le plus ancien processus en exécution. Le problème de cette solution est que p effectue des retours en arrière tant que q tient la ressource.

La seconde solution, provoquant en moyenne moins de retours en arrière, applique une règle duale. Lorsqu'un processus p demande une ressource tenue par q, p attend la libération de cette ressource si p est plus récent que q. Autrement, p provoque le retour en arrière de q et q doit libérer toutes ses ressources puis recommencer son exécution à partir du début. Cette solution est connue sous le nom de « blesser ou attendre ». Cette solution prévient également tout risque de famine car tout processus devient à terme le plus ancien et n'est ensuite plus interrompu jusqu'à la fin de son exécution.

57

5.5.1 Exercice

- Un seul exemplaire des ressources R_1, R_2, R_3 et R_4
- Une unique section critique par processus, avec besoin de deux ressources
- P_2 a besoin de ses ressources pendant toute la durée de son exécution, les autres processus pendant seulement la seconde moitié

Processus	Première création	Durée	Resources consommées
P_1	8 h 55	14 mn	R_1, R_2
P_2	8 h 57	6 mn	R_2, R_3
P_3	9 h 02	10 mn	R_1, R_3
P_4	9 h 10	4 mn	R_3, R_4
P_5	9 h 11	2 mn	R_2, R_4

? Donnez les exécutions en montrant les attributions des ressources dans les méthodes « attendre ou mourir » et « blesser ou attendre »

? Comptez le nombre de retours en arrière

À titre d'exercice, soit cinq processus qui chacun s'exécute sur un processeur particulier. Le tableau de la diapositive donne l'instant de démarrage (soit la date de première création), la durée des calculs à effectuer pour chaque processus et les ressources requises pour le traitement. Supposons qu'il n'existe qu'un seul exemplaire des ressources R_1, R_2, R_3 et R_4. Le processus P_2 a besoin de ses ressources pendant toute la durée de son exécution et les autres processus pendant seulement la seconde moitié. Indiquez comment s'effectue l'attribution des ressources avec les techniques « attendre et mourir » et « blesser ou attendre » et comparez le nombre de retours en arrière provoqués par les deux méthodes. Rappelons qu'un retour en arrière ramène tout au début de l'exécution, mais sans changement de date de la première création.

5.6 Détection d'interblocage

58

- Évaluation du prédicat $fulfilled_p$ dans une configuration
- Interblocage est une propriété stable
  - = une configuration bloquée reste bloquée tant que non-résolution par un algorithme de contrôle
- Détection de propriété stable
  - ♦ Effectuée sur une configuration
  - ♦ Construction d'une configuration
    - ► Algorithme de coupure dite « cohérente »
      - ★ Configuration ayant « réellement pu exister » durant l'exécution
        - = L'action de réception d'un message ne peut pas être observée si son action d'émission ne l'est pas aussi

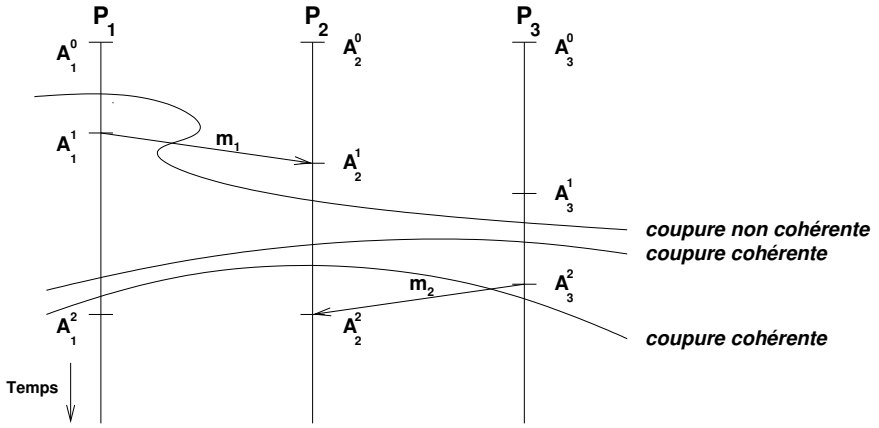
Un algorithme de détection d'interblocage évalue le prédicat $fulfilled_p$ dans une configuration. L'interblocage est une propriété stable d'une exécution répartie : une configuration bloquée reste bloquée tant que l'interblocage n'est pas résolu par un algorithme de contrôle.

La détection de propriété stable s'effectue sur une configuration de l'exécution répartie. Les configurations d'une exécution répartie qui nous intéressent sont celles qui ont « réellement pu exister ». La cohérence s'exprime simplement par le fait que l'action de réception d'un message ne peut pas être observée si son action d'émission ne l'est pas aussi.

59

5.6.1 Coupure cohérente

- Soit l'état global $C = \{s_1, s_2, ..., s_n\}$ des processus $\{p_1, p_2, ..., p_n\}$,
- $cohérent(C) \stackrel{\text{def}}{=} \forall p \neq q,\ \neg(s_p \prec s_q)$



La cohérence s'exprime simplement par le fait que l'action de réception d'un message ne peut pas être observée si son action d'émission ne l'est pas aussi.

La construction d'une configuration est réalisée par un algorithme de coupure cohérente qui enregistre un état local par processus de l'ensemble des processus participant au système. La propriété s'exprime comme suit (l'opérateur $\prec$ est l'opérateur de précédence causale) :

$$C = \{s_1, s_2, ..., s_n\} : \forall p \neq q,\ \neg(s_p \prec s_q)$$

Dans l'expression, s_p signifie « état du processus p » (« s » pour *state*). La propriété de cohérence vérifie qu'aucun message n'est vu reçu dans la coupure sans être vu émis, c.-à-d. aucun état local de la coupure ne dépend causalement d'un autre état local de la même coupure.

5.6.2 Algorithme centralisé de construction de coupure cohérente de Chandy et Lamport, 1985

- Les canaux de communication sont supposés FIFO

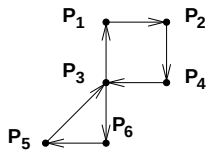
60

```
1  var taken_P : boolean init false ;
2  To initiate the algorithm :
3     record the local state ; taken_P := true ;
4     forall q ∈ Neighbours_P do send ⟨MARKER⟩ to q
5  When a MARKER arrives :
6     receive ⟨MARKER⟩ ;
7     if ¬taken_P then
8        record the local state ; taken_P := true ;
9        forall q ∈ Neighbours_P do send ⟨MARKER⟩ to q
```

L'algorithme de Chandy et Lamport, 1985 est un algorithme centralisé : un processus initiateur démarre l'algorithme pour construire une coupure cohérente qui passe par son état local (l'état local du moment du démarrage de l'algorithme). Par ailleurs, il suppose que les canaux de communication sont FIFO. Les processus s'informent mutuellement de la construction d'une coupure en émettant un message ⟨MARQUEUR⟩ sur chacun des canaux de communication. La réception d'un message ⟨MARQUEUR⟩ par un processus qui n'a pas encore enregistré son état local provoque la saisie de l'état du processus et l'émission des messages ⟨MARQUEUR⟩.

5.6.3 Exercice

61



? Donnez une exécution de l'algorithme de Chandy et Lamport, 1985 avec P_1 initiateur

? Donnez un algorithme valable si les canaux de communication ne sont pas FIFO

? * Donnez un algorithme de détection d'interblocage dans le modèle ET *

6 Détection de terminaison

 # 62

Cette section sur la détection de terminaison d'algorithme « utilisateur » est reprise de la référence suivante :

- G. Tel, *Chapter 8 : Termination Detection*, dans Introduction to Distributed Algorithms, Cambridge University Press, pp. 264–302, 1994.

Dans la section traitant précédemment de l'interblocage, nous avons implicitement introduit le problème de la détection de terminaison. La détection de terminaison est équivalente au problème de la détection d'interblocage dans le modèle OU, dit modèle de communication. Aussi, il est possible d'utiliser des algorithmes génériques de coupure cohérente. Cette section présente le modèle OU d'interblocage avec plus de détails et propose d'autres algorithmes plus spécifiques et plus performants dans une topologie quelconque, mais pour des calculs dits « diffusants », puis dans la topologie de l'anneau.

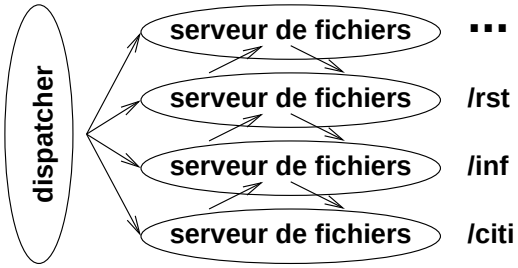
6.1 Modèle OU de l'interblocage

63

- Interblocage (non fin explicite) lorsque la configuration est bloquée
- Modèle OU : $fulfilled_p$ de la forme « une permission doit être reçue d'au moins un processus de l'ensemble des fournisseurs possibles de permissions »
- Définitions, dans une configuration S
  - Processus passif (en attente de réception d'un message) *vs.* actif (non passif)
  - Processus vivant : s'il existe une configuration T, atteignable à partir de S, dans laquelle il est actif
  - Processus mort : sinon
  - Configuration bloquée : s'il existe des processus morts

Un processus coopérant avec d'autres processus dans un système réparti peut entrer dans un état passif, dans lequel la seule action possible est la réception d'un message. Ce processus attend un message en provenance d'un ensemble non vide de processus. L'émission d'un seul message par un seul des processus de cet ensemble active le processus qui était passif. Pour être actif, le processus doit donc obtenir au moins un message en provenance d'au moins un processus d'un ensemble de processus. Ce modèle est le modèle OU de l'interblocage. Dans le modèle OU, $fulfilled_p$ est donc de la forme « une permission doit être reçue d'au moins un processus de l'ensemble des fournisseurs possibles de permissions ».

La figure qui suit montre l'exemple d'un service réparti, ici un système de fichiers réparti. Chaque système de fichiers exécute une boucle infinie de réception/traitement de requêtes. Un processus dédié reçoit toutes les requêtes des clients : c'est le point d'entrée du service, et distribue les requêtes sur les serveurs de fichiers selon les espaces de nommage gérés par chacun d'entre eux. Les serveurs de fichiers échangent des requêtes lorsqu'ils parcourent des liens symboliques, faisant passer les traitements d'un serveur de fichiers à un autre. Imaginons que les administrateurs système souhaitent arrêter le service réparti pour une opération de maintenance. Pour ce faire, ils bloquent les requêtes des clients. La fin du traitement des requêtes en cours est détectée lorsque tous les serveurs sont en attente de requêtes et qu'aucune requête entre serveur n'est en transit. Lorsque la fin du traitement des requêtes est atteinte, les administrateurs peuvent sans risque arrêter les serveurs et opérer la maintenance.



Pour la suite de la discussion, voici quelques définitions. Un processus est dit « passif » s'il est arrêté sur la réception d'un message. Sinon, il est dit « actif » : en cours d'exécution d'une action interne ou d'une émission de message. Un processus p est « vivant » dans une configuration S s'il existe une configuration T, atteignable à partir de S, dans laquelle p est actif. p est « mort » s'il n'est pas vivant. Une configuration S est une configuration « bloquée » s'il existe des processus morts dans S.

6.2 Configurations terminale et finale

64

- Configuration terminale : aucune action supplémentaire ne peut être exécutée =
  - $\wedge$ Tous les processus s'arrêtent sur une réception de message
  - $\wedge$ Tous les canaux de communication sont vides
  - ♦ C'est une terminaison implicite ou dite « de message »
- $\neq$ Configuration finale : aucune action n'est applicable
  - ♦ Tous les processus atteignent l'état final
  - ♦ C'est une terminaison explicite ou dite « de processus »

- Plus facile (en général) de concevoir un algorithme réparti à terminaison implicite
  - ♦ Possibilité d'ignorer tous les aspects relatifs à la terminaison
  - ♦ Comportement des processus simplifié (pas de message spécifique)
  - ♦ Seulement bornage du nombre de messages

- Transformation d'une terminaison implicite en une terminaison explicite
  - ♦ Détection de la terminaison
  - ♦ Diffusion d'un message de terminaison

L'exécution d'un algorithme réparti se termine lorsque l'algorithme atteint une configuration terminale : une configuration dans laquelle aucune action supplémentaire ne peut être exécutée. Dans une configuration terminale, tous les processus ne sont pas forcément dans un état final : dans un état dans lequel aucune action n'est applicable. Considérons une configuration dans laquelle tous les processus s'arrêtent sur une réception de message et tous les canaux de communication sont vides. Une telle configuration est terminale mais les processus n'ont pas connaissance que c'est une configuration terminale : la terminaison est dite « implicite ». Une terminaison est dite « explicite » dans un processus si l'état du processus atteint dans la configuration terminale est l'état final. La terminaison implicite est aussi appelée la « terminaison de message » (en anglais, « *message termination* »), la terminaison explicite la « terminaison de processus » (en anglais, « *process termination* »).

Il est en général plus facile de concevoir un algorithme réparti à terminaison implicite qu'un algorithme réparti à terminaison explicite. Un exemple que nous avons déjà rencontré est l'algorithme de coupure cohérente de Chandy et Lamport, 1985 : durant la conception, tous les aspects relatifs à la terminaison peuvent être ignorés, le concepteur s'occupant seulement de borner le nombre de messages.

Dans la suite de cette section, nous étudions des méthodes générales transformant des algorithmes à terminaison implicite en algorithmes à terminaison explicite. La méthode consiste en deux algorithmes. Le premier algorithme détecte la terminaison et lance l'exécution du deuxième qui diffuse un message de terminaison pour que tous les processus atteignent un état final. La partie la plus difficile est évidemment la détection de terminaison.

65

6.3 États actif et passif, et algorithme de contrôle

- État actif : si une action interne ou l'action `émettre()` est applicable
- État passif sinon

- Distinguer algorithme de détection de terminaison et algorithme de l'utilisateur :
  - Algorithme de détection de terminaison = algorithme de contrôle
    - Algorithme de contrôle : exécution dite sous-jacente
    - Messages de contrôle ne rendant pas actifs les processus passifs

- Transition de l'état actif à l'état passif
  - Avant `recevoir()`, action fictive de actif vers passif
- Transition de l'état passif à l'état actif
  - Après `recevoir()`, action fictive de passif vers actif
- Messages de contrôle émis lorsque les processus sont passifs

- Détection
  - Par calcul du graphe d'exécution (Dijkstra et Scholten, 1980)
  - Par vagues (Safra, 1987)

L'ensemble des états d'un processus p_i est divisé en deux sous-ensembles : celui des états passifs et celui des états actifs. Un état s_i^x est *actif* si une action interne ou l'action `émettre()` est applicable dans s_i^x, autrement, il est *passif*. Dans un état passif, seule l'action `recevoir()` est applicable ou aucune action n'est applicable, auquel cas l'état est final (`exit` ou fin du `main`).

Afin de distinguer l'exécution de l'algorithme de détection de terminaison de l'exécution de l'algorithme de l'utilisateur, la première exécution est dite « sous-jacente », le premier algorithme est dit « de contrôle », et ses messages sont dits « de contrôle ». Les messages de contrôle sont émis lorsque les processus sont passifs, et ils ne rendent pas actifs les processus passifs lorsqu'ils sont reçus.

Dans ce qui suit, toute action de réception de messages est précédée d'une action fictive qui fait passer le processus à l'état passif. De la même façon, toute action de réception est suivie d'une action fictive qui fait passer le processus à l'état actif.

Nous présentons deux catégories d'algorithmes de détection de terminaison dans cette section. Dans la première catégorie, la détection par calcul du graphe d'une exécution, le graphe d'une exécution est un graphe dans lequel les nœuds sont les processus actifs et les messages « en transit ». L'algorithme de Dijkstra et Scholten est effectif pour les algorithmes de l'utilisateur du type centralisé : un initiateur émet le premier message ; les autres processus sont passifs dans l'attente de calculs ; l'algorithme de détection de terminaison construit un arbre de l'exécution. La terminaison implicite est détectée lorsque le graphe est vide. Le principe des algorithmes de la seconde catégorie est l'exécution répétitive de vagues à la fin desquelles soit la terminaison est détectée, soit une nouvelle vague est lancée. Nous présentons un tel algorithme pour la topologie en anneau (Safra, 1987).

6.4 Algorithmes de détection de terminaison

66

- Prédicat `term` défini comme suit :

$$\texttt{term} \Leftrightarrow \begin{cases} & \forall p : state_p = passif \\ \wedge & \forall p, q : canal_{pq} = \emptyset \end{cases}$$

- Propriété d'un algorithme de détection de terminaison :
  1. Non-interférence : non-influence de l'exécution de l'algorithme de l'utilisateur
  2. Vivacité : `term` $\Rightarrow$ appel de l'algorithme de terminaison dans un temps fini
  3. Sûreté : algorithme de terminaison appelé $\Rightarrow$ `term`
- Deux cas selon que l'algorithme de détection est démarré avant ou après l'algorithme de l'utilisateur
  1. Après : les canaux de communication doivent être FIFO
  2. Avant : pas de contrainte d'ordre (toujours notre cas dans la suite)

Le prédicat `term` est défini comme suit :

$$\texttt{term} \Leftrightarrow \begin{cases} & \forall p : state_p = passif \\ \wedge & \forall p, q : canal_{pq} = \emptyset \end{cases}$$

Un algorithme de détection de terminaison doit satisfaire les trois propriétés suivantes :

1. non-interférence : il ne doit pas influencer l'exécution de l'algorithme de l'utilisateur ;
2. vivacité (en anglais, *liveness*) : si le prédicat `term` est vérifié, la configuration est terminale et l'algorithme de terminaison doit être appelé dans un temps fini pour que la terminaison soit effective ;
3. Sûreté (en anglais, *safety*) : si l'algorithme de terminaison est appelé, la configuration doit satisfaire le prédicat `term`.

L'exécution de l'algorithme de détection peut commencer avant l'algorithme de l'utilisateur ou après. Dans le second cas (après l'algorithme de l'utilisateur), les canaux de communication doivent être FIFO. Pour notre part, nous supposons qu'aucune action de l'algorithme de l'utilisateur n'est exécutée avant que l'exécution de l'algorithme de contrôle n'ait été commencée.

6.5 Détection par calcul du graphe d'exécution

67

- Cas de l'algorithme de l'utilisateur « centralisé » : un seul initiateur r, calcul dit « diffusant »
- Arbre d'exécution $T = (V_T, E_T)$ avec les propriétés suivantes :
  1. soit T vide, soit T arbre orienté (racine $= r$)
  2. V_T inclut processus actifs et messages de l'algorithme de l'utilisateur « en transit »
- $p \notin T$ devient actif sur réception d'un message de q : ($p \in T \ \wedge$ q père de p)
- r lance l'algorithme de terminaison lorsque $r \notin V_T$
  1. Cela signifie : T est vide $\wedge$ *term*

L'algorithme de contrôle maintient dynamiquement l'arbre d'exécution $T = (V_T, E_T)$ avec les propriétés suivantes :

1. soit T est vide, soit T est un arbre orienté dont la racine est l'initiateur r ;
2. l'ensemble E_T inclut les relations « parent–enfant », avec les parents qui sont des processus et les enfants qui sont des processus ou des messages ;
3. l'ensemble V_T inclut tous les processus actifs et tous les messages de l'algorithme de l'utilisateur « en transit ».

Lorsqu'un processus p émet un message m pour l'algorithme de l'utilisateur, le message m est ajouté à T, p est le parent du message m. Lorsqu'un processus p qui n'était pas dans T devient actif par la réception d'un message en provenance de q, q devient le parent de p.

L'initiateur de l'algorithme de l'utilisateur, r, lance l'exécution de l'algorithme de terminaison lorsqu'il n'est plus dans l'arbre ($r \notin V_T$). En effet, dans ce cas, par la première propriété, T est vide, et par la seconde propriété, le prédicat *term* est vérifié.

6.5.1 Algorithme de Dijkstra et Scholten, 1980

68

```
1  var state_p : (active, passive) init if p is initiator then active else passive ; /* only one initiator */
2  var sc_p : integer init 0 ; /* number of sons */
3  var parent_p : process init if p is initiator then p else undef
4  S_p : { state_p = active ∧ call to send⟨m⟩ from the user application}
5     send⟨m, p⟩ ; sc_p := sc_p + 1
6  R_p : { call to receive⟨m⟩ from the user application ∧ a message ⟨m, q⟩ has arrived at p }
7     receive⟨m, q⟩ ; state_p := active ;
8     if parent_p = undef then parent_p := q else send⟨sig, q⟩ to q
9  I_p : { state_p = active ∧ (before receive ∨ at the end of main) }
10    state_p := passive ;
11    if sc_p = 0 then /* delete p from T */
12       if parent_p = p then announce else send⟨sig, parent_p⟩ to parent_p
13       parent_p := undef
14 A_p : { a signal ⟨sig, p⟩ has arrived at p }
15    receive⟨sig, p⟩ ; sc_p := sc_p − 1 ;
16    if sc_p = 0 ∧ state_p = passive then
17       if parent_p = p then announce else send⟨sig, parent_p⟩ to parent_p ;
18       parent_p := undef
```

Les actions S_p et R_p englobent les actions d'émission et de réception de l'algorithme de l'utilisateur, c'est-à-dire que les appels aux primitives `send` et `receive` de l'algorithme de l'utilisateur sont délégués aux actions S_p et R_p, respectivement.

Aux deux actions d'émission S_p et de réception R_p des messages de l'algorithme de l'utilisateur, sont ajoutées deux autres actions :

1. avant de se mettre en attente de réception d'un message, un processus exécute l'action I_p qui le rend passif ;
2. les processus émettent des signaux dans les actions R_p et I_p. Ces signaux sont traités dans l'action spécifique A_p.

Les messages de l'utilisateur sont utilisés pour construire l'arbre : les actions d'émission `send` de l'utilisateur sont remplacées par l'action S_p de l'algorithme de Dijkstra et Scholten, 1980. Les messages de contrôle ajoutés sont les signaux qui permettent d'acquitter la réception des messages applicatifs.

Les messages sont des feuilles de T. Les processus maintiennent une variable sc comptant le nombre d'enfants dans T. La suppression d'un enfant du processus p_i s'effectue dans un autre processus p_j. La suppression correspond soit à la réception du message $\langle m, p_i\rangle$ par p_j, soit à la fin du processus p_j. Lorsque cela arrive, p_j émet un message $\langle sig, p_i\rangle$ vers p_i pour que p_i décrémente son compteur sc_{p_i}.

La détection de la terminaison est effectuée lors du passage de l'état actif vers l'état passif et lors de la réception d'un signal de suppression d'un élément de l'arbre. L'algorithme de terminaison est lancé par la primitive `announce()`.

Pour la preuve de l'algorithme, il faut montrer que l'arbre T devient vide après la terminaison de l'algorithme de l'utilisateur. Plus précisément, dans toutes les configurations S de l'algorithme de Dijkstra et Scholten, 1980, l'arbre T est construit tel que :

- $V_T = \{p : parent_p \neq udef\} \cup \{\langle m, p\rangle : \langle m, p\rangle \text{ is in transit}\} \cup \{\langle sig, p\rangle : \langle sig, p\rangle \text{ is in transit}\}$
- $E_T = \{(p, parent_p) : parent_p \neq udef \wedge parent_p \neq p\} \cup \{(\langle m, p\rangle, p) : \langle m, p\rangle \text{ is in transit}\} \cup \{(\langle sig, p\rangle, p) : \langle sig, p\rangle \text{ is in transit}\}$.

Pour prouver la sûreté de l'algorithme, nous devons démontrer l'invariant P qui suit. Autrement dit, nous montrons que P est un invariant de l'algorithme de Dijkstra et Scholten, 1980, puis nous utilisons l'invariant P pour montrer que l'algorithme de Dijkstra et Scholten, 1980 détecte la terminaison de l'algorithme de l'utilisateur.

$$
\begin{aligned}
P \stackrel{\text{def}}{=} \quad & state_p = active \implies p \in V_T && (1)\\
\wedge\ & sc_p = \#\{v : (v,p) \in E_T\} && (2)\\
\wedge\ & V_T \neq \emptyset \implies T \text{ is a tree with root } r && (3)\\
\wedge\ & (state_p = passive \wedge sc_p = 0) \implies p \notin V_T && (4)
\end{aligned}
$$

La signification de l'invariant est la suivante :

1. l'arbre T inclut tous les processus actifs,
2. le compteur du nombre d'enfants du processus p est sc_p,
3. T est un arbre dont la racine est le processus r, initiateur de l'algorithme de l'utilisateur,
4. l'arbre T est vide lorsque l'algorithme de l'utilisateur se termine.

Lemme 9 *P est un invariant de l'algorithme de Dijkstra et Scholten, 1980.*

Preuve : Pour montrer que P est invariant, nous passons en revue chacune des actions, en commençant par l'état initial obtenu après création et initialisation des variables de l'algorithme.

$\forall p \neq r, state_p = passive$ et $parent_r \neq udef$. Cela respecte (1). En outre, $\forall p \in \Pi, sc_p = 0$, ce qui respecte (2). $V_T = \{r\}$ et $E_T = \emptyset$, donc T est un arbre ayant pour racine r, ce qui respecte (3). r est le seul processus appartenant à V_T et r est actif, ce qui respecte (4).

Lors de l'exécution de l'action S_p, aucun processus ne devient actif et aucun processus n'est retiré de V_T. Donc, (1) est préservé. Lors de l'exécution de l'action S_p, le sommet $\langle m, p\rangle$ est inséré dans V_T et l'arête $(\langle m, p\rangle, p)$ est insérée dans E_T, ce qui implique que T reste un arbre ayant pour racine r. Par ailleurs, sc_p est correctement incrémenté. Donc, (2) et (3) sont donc préservés. Enfin, aucun processus ne devient passif et aucun processus n'est ajouté à V_T. Donc, (4) est préservé.

Lors de l'exécution de l'action R_p, soit p était déjà dans V_T (cas $parent_p \neq udef$), soit p devient actif et est inséré dans V_T. Donc, (1) est préservé. Le nombre d'enfants de q ne change pas, car l'enfant $\langle m, q\rangle$ de q est remplacé soit par l'enfant p soit par l'enfant $\langle sig, q\rangle$. Le compteur sc_q reste donc correct, ce qui préserve (2). Par ailleurs, la structure du graphe ne change pas, donc (3) est préservé. À la fin de l'exécution de l'action R_p, dans tous les cas, p est dans V_T. Donc, (4) est préservé.

Dans l'action I_p, le fait de rendre le processus p passif préserve (1), (2), et (3). Par ailleurs, si $sc_p = 0$ alors p est retiré de V_T, ce qui préserve (4). Nous devons maintenant examiner le reste de l'action I_p. Considérons ce qui se passe lorsque p est retiré de T, c'est-à-dire lorsque p était une feuille de l'arbre et que p devient passif. Si un signal est émis par p, le parent du signal est le dernier parent de p, le dernier parent de p étant dans V_T. Le signal remplace l'enfant p de $parent_p$. D'où, sc_{parent_p} reste correct. Donc, (2) est préservé. Par ailleurs, la structure du graphe n'est pas modifiée et (3) est préservé. Dans le cas où $parent_p = p$ est vrai, $p = r$, et p devenant une feuille implique que p était le seul sommet de T. Donc, la suppression de p de T rend T vide. Donc, (3) reste vrai.

Dans l'action A_p, la réception du signal décrémente le nombre d'enfants de p de un. (2) est donc préservé. Le fait que p était le parent du signal implique que p est dans V_T. Si p est passif et que le compteur sc_p est passé à 0, alors p est retiré de V_T. Donc, (4) est préservé. Si p est retiré de V_T, l'invariant est préservé par les mêmes arguments que ceux exprimés dans l'action I_p. □

Théorème 12 *L'algorithme de Dijkstra et Scholten, 1980 est un algorithme de détection de terminaison.*

Preuve : Nous utilisons les termes de l'invariant démontré dans le lemme 9.

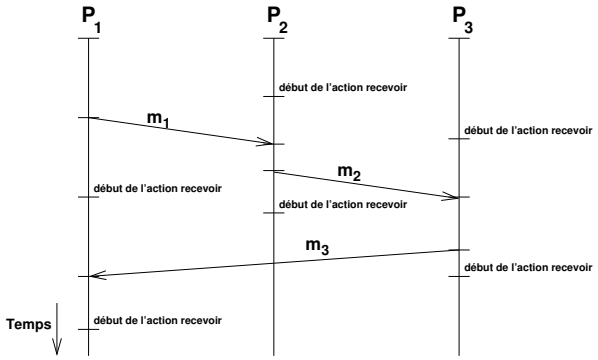
Pour démontrer la sûreté de l'algorithme, remarquons que r est le seul processus qui appelle la méthode `announce`. L'appel intervient quand r est lui-même retiré de l'arbre T. Par (4), T devient vide, ce qui implique que le prédicat `term` est vrai.

Pour démontrer la vivacité de l'algorithme, supposons que l'algorithme de l'utilisateur est terminé. Après la terminaison, seule l'action A_p est applicable. Notons S la somme de tous les compteurs, c'est-à-dire $S = \sum_{p \in P} sc_p$. Clairement, S décroît d'une unité à chaque exécution de l'action A_p. D'où, l'algorithme atteint une configuration terminale dans laquelle V_T ne contient aucun message. Aussi, selon (4), V_T ne contient aucun processus passif. Par conséquent, T ne contient aucune feuille. L'arbre devient vide dès que r est retiré de l'arbre. Dans ce cas, selon l'algorithme, r appelle la méthode `announce`. □

69

6.5.2 Exercice

? Donner une exécution de l'algorithme de Dijkstra et Scholten, 1980 avec l'algorithme de l'utilisateur suivant



? Donner la complexité (en nombre de messages) de l'algorithme

? Donner un algorithme de terminaison

? * Donner une amélioration de l'algorithme de Dijkstra et Scholten, 1980 dans le cas où plusieurs processus de l'algorithme de l'utilisateur émettent avant de recevoir *

6.6 Détection par vagues dans un anneau

70

- Hypothèse : existence d'un anneau de tous les processus
- Communications éventuellement mono-directionnelles
- Anneau utilisé uniquement pour les messages de contrôle
- Communications synchrones :

$$\texttt{term} \Leftrightarrow \forall p_i : state_{p_i} = passive$$

- Communications asynchrones : en plus, comptage des messages en transit

$$\texttt{term} \Leftrightarrow (\forall p_i : state_{p_i} = passive) \wedge (nbMsg = 0)$$

Dans de nombreux systèmes répartis, l'ensemble des processus est connu et peut donc être organisé pour former un anneau. Cela ne signifie pas que l'algorithme de l'utilisateur utilise l'anneau, mais que des algorithmes de contrôle tels que les algorithmes de détection de terminaison peuvent l'utiliser. Les communications dans cet anneau peuvent être mono-directionnelles ; nous verrons que l'algorithme de Safra, 1987, lance des vagues dans l'anneau toujours dans le même sens.

Dans le cas de communications synchrones, pour détecter la terminaison, il faut vérifier que tous les processus sont passifs :

$$\texttt{term} \Leftrightarrow \forall p_i : state_{p_i} = passive$$

Dans le cas de communications asynchrones, il faut en plus compter les messages en transit :

$$\texttt{term} \Leftrightarrow (\forall p_i : state_{p_i} = passive) \wedge (nbMsg = 0)$$

6.6.1 Algorithme de Safra, 1987

71

```
1  var state_p : (active, passive) init active ; color_p : (white, black) ; mc_p : integer init 0 ;
2  S_p : { state_p = active ∧ call to send⟨m⟩ from the user application }
3     send⟨m⟩ ; mc_p := mc_p + 1 /* Rule M */
4  R_p : { call to receive⟨m⟩ from the user application ∧ a message ⟨m⟩ has arrived at p }
5     receive⟨m⟩ ; state_p := active ; mc_p := mc_p − 1 ; color_p := black /* Rules M + 2 */
6  I_p : { state_p = active }
7     state_p := passive ;
8  Start the detection, executed once by p_0 :
9     send⟨tok, white, 0⟩ to Next_p
10 T_p : { state_p = passive ∧ token ⟨tok, c, q⟩ has arrived } /* Rule 1 */
11    if p = p_0 then
12       if (c = white) ∧ (color_p = white) ∧ (mc_p + q = 0)
13          then announce else send⟨tok, white, 0⟩ to Next_p ; /* Rule 4 */
14    else
15       if (color_p = white) /* Rules 1 + 3 */
16          then send⟨tok, c, q + mc_p⟩ to Next_p else send⟨tok, black, q + mc_p⟩ to Next_p
17    color_p := white ; /* Rule 5 */
```

Comme dans l'algorithme de Dijkstra et Scholten, les actions S_p et R_p englobent les actions d'émission et de réception de l'algorithme de l'utilisateur, c'est-à-dire que les appels aux primitives `send` et `receive` de l'algorithme de l'utilisateur sont délégués aux actions S_p et R_p, respectivement.

Comme dans l'algorithme de Dijkstra et Scholten, deux actions sont ajoutées à l'algorithme de l'utilisateur : I_p précède la réception d'un message et T_p traite la réception d'un jeton.

L'algorithme de contrôle est démarré avant le début de l'algorithme de l'utilisateur. La détection de terminaison est démarrée par p_0. L'action de démarrage n'est effectuée qu'une seule fois. Ensuite, le jeton circule jusqu'à la détection de la terminaison.

L'algorithme compte dans chaque processus le nombre de messages émis et reçus. En comparant les compteurs, il peut être certifié que les canaux de communication sont vides. Pour opérer le comptage des messages émis et reçus, chaque processus maintient une variable mc égale au nombre de messages émis moins le nombre de messages reçus et possède une couleur $color$. Les processus font circuler des messages $\langle jeton, c, q\rangle$ avec c la couleur du jeton et q le compteur de messages en transit.

L'algorithme est défini par la série de règles qui suit :

- règle M : lorsqu'un processus émet un message $\langle m\rangle$, il incrémente son compteur de messages ; lorsqu'il en reçoit un, il décrémente le compteur ;
- règle 1 : un processus reçoit un message $\langle jeton, c, q\rangle$ uniquement lorsqu'il est passif, et quand il émet un message $\langle jeton, c, q\rangle$, il ajoute la valeur de son compteur à q ;
- règle 2 : un processus recevant un message $\langle m\rangle$ de l'algorithme de l'utilisateur devient noir ;
- règle 3 : lorsqu'un processus noir fait suivre un message $\langle jeton, c, q\rangle$, c devient noir ;
- règle 4 : une vague ne se termine pas avec succès si, lorsque le message $\langle jeton, c, q\rangle$ revient à l'initiateur p_0, c est égale à noir ou p_0 est noir, ou encore $mc_{p_0} + q \neq 0$. Dans ce cas, p_0 lance l'exécution d'une nouvelle vague ;
- règle 5 : chaque processus devient blanc juste après l'émission d'un message $\langle jeton, c, q\rangle$.

En considérant que le jeton tourne dans le sens $p_0 \to p_{|\Pi|} \to p_{|\Pi|-1} \to \ldots$, cette série de règles permet d'obtenir l'invariant P suivant, t étant le numéro du processus qui détient le jeton ou le numéro du processus vers lequel le jeton est en transit :

$$P \stackrel{\text{def}}{=} P_m \wedge (P_0 \vee P_1 \vee P_2 \vee P_3)$$

avec :

$$
\begin{aligned}
P_m &\stackrel{\text{def}}{=} NbMsg = \sum_{p \in \Pi} mc_p \\
P_0 &\stackrel{\text{def}}{=} (\forall i : |\Pi| > i > t : state_{p_i} = passive) \wedge (q = \sum_{|\Pi|>i>t} mc_{p_i}) \\
P_1 &\stackrel{\text{def}}{=} (\sum_{i \leq t} mc_{p_i} + q) > 0 \\
P_2 &\stackrel{\text{def}}{=} \exists j : t \geq j \geq 0 : color_{p_j} = black \\
P_3 &\stackrel{\text{def}}{=} \text{the token is black}
\end{aligned}
$$

Le prédicat P_m est l'invariant de comptage des messages qui permet de vérifier que tous les messages applicatifs émis ont été reçus. Ce premier prédicat traduit la règle M.

Le prédicat P_0 complète le prédicat P_m pour indiquer que, lorsque le processus p_0 reçoit le jeton, il peut décider si la terminaison est effective : tous les processus sont-ils passifs et tous les messages applicatifs émis ont-il été reçus ?

Pour préserver P_0, les processus doivent traiter le jeton uniquement lorsqu'ils sont passifs. C'est le rôle de la règle 1. Remarquez que, comme les messages applicatifs ne peuvent pas être reçus avant d'avoir été émis, la somme des compteurs des processus lorsqu'ils sont passifs et traitent la réception du jeton plus la valeur du compteur q dans le jeton qui revient à p_0 est toujours positive ou égale à 0. Cette somme supérieure à 0 exprime les cas de réception de messages applicatifs par les processus $i > t$ qui falsifient P_0. Cela s'exprime dans l'invariant par le OU entre P_0 et P_1. Notez aussi que la détection de terminaison reste effective lorsque P_0 est vrai et P_1 est faux. Il reste cependant des cas falsifiant $P_0 \vee P_1$.

P_1 est falsifié dans les cas de réception de messages applicatifs par les processus $i \leq t$. C'est pour cela que l'algorithme de Safra introduit le coloriage des processus. Les processus recevant un message applicatif après le passage du jeton deviennent noirs et le jeton devient noir lorsqu'il passe chez un processus noir : ce sont les règles 2 et 3. Cela correspond aux cas non couverts par le prédicat $P_m \wedge (P_0 \vee P_1)$. Les règles 4 et 5 servent à compléter l'algorithme de Safra pour la détection effective de la terminaison après l'ajout du coloriage des processus et du jeton : le jeton doit être blanc lorsqu'il est traité par p_0 et les processus deviennent blancs après l'émission du jeton. Par ailleurs, les prédicats P_2 et P_3 expriment les conditions falsifiant $P_m \wedge (P_0 \vee P_1)$.

Théorème 13 *L'algorithme de Safra, 1987 est un algorithme de détection de terminaison.*

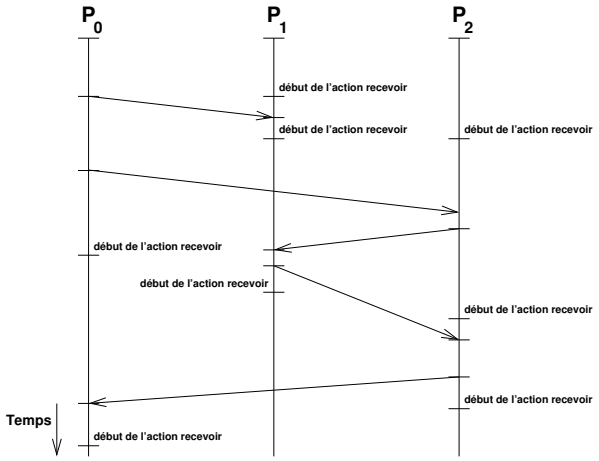
Preuve : Le prédicat P est un invariant de l'algorithme de Safra, 1987.

Pour montrer la sûreté de l'algorithme, il faut observer que la terminaison est détectée par le processus p_0 ($t = 0$) avec $state_{p_0} = passive$, $color_{p_0} = white$, $c = white$, et $mc_{p_0} + q = 0$. Ces conditions impliquent $\neg P_1 \wedge \neg P_2 \wedge \neg P_3$, donc $P_m \wedge P_0$, qui implique `term`.

Pour montrer la vivacité de l'algorithme, il faut observer qu'une fois l'algorithme de l'utilisateur terminé, les compteurs ne changent pas de valeur, et que leur somme est à égale à 0. Une vague qui est lancée dans une telle configuration reviendra au processus p_0 avec $mc_{p_0} + q = 0$ et tous les processus coloriés à blanc, après quoi la vague qui suit est garantie de revenir en activant la ligne 13 de l'algorithme : les conditions des lignes 11 et 12 sont vraies. □

72

6.6.2 Exercice



? Donner une exécution de l'algorithme de Safra avec l'algorithme de l'utilisateur précédent

? Combien de tour(s) faut-il au minimum pour détecter la terminaison ?

Bibliographie

- Éléments introductifs :
  - G. Tel, *Chapter 2 : The Model*, dans *Introduction to Distributed Algorithms*, Cambridge University Press, pp. 43–72, 1994.
  - C. Fidge. *Logical Time in Distributed Computing Systems*, dans *IEEE Computer*, pages 28–33, August 1991.
  - G. Tel, *Chapter 6 : Wave and traversal algorithms*, dans *Introduction to Distributed Algorithms*, Cambridge University Press, pp. 177–221, 1994.
  - H. Attiya et J. Welch, *Chapter 2 : Basic Algorithms in Message-Passing Systems*, dans *Distributed Computing : Fundamentals, simulation, and advanced topics*, Wiley, pp. 9–30, 2004.
  - J.H. Saltzer, M.F. Kaashoek, *Principles of Computer System Design : An Introduction*, Morgan Kaufmann, 2009.
- Élection :
  - G. Tel, *Chapter 7 : Election Algorithms*, dans Introduction to Distributed Algorithms, Cambridge University Press, pp. 222–263, 1994.
- Diffusion, communication de groupe :
  - V. Hadzilacos et S. Toueg, *Chapter 5 : Fault-Tolerant Broadcasts and Related Problems*, dans Distributed Systems 2nd Ed., Editeur S.J. Mullender, ACM Press, pp. 97–145, 1993.
  - V. Hadzilacos et S. Toueg, *A Modular Approach to Fault-Tolerant Broadcasts and Related Problems*, dans Technical Report TR94-1425, Department of Computer Science, Cornell University, Ithaca (N.Y., USA), 84 pages, May 1994.
  - K.P. Birman et T.A. Joseph, *Reliable Communication in the Presence of Failures*, ACM Transactions on Computer Systems, 5(1):47–76, February 1987.
  - Guerraoui, R. and Rodrigues, L. *Introduction to Reliable Distributed Programming*, Springer, 2006.
  - M.J. Fischer, N.A. Lynch et M.S. Paterson, *Impossibility of Distributed Consensus with One Faulty Process*, dans Journal of the ACM, Volume 32, Numéro 2, pp. 374–382, Avril 1985.
- Exclusion mutuelle :
  - O.S.F. Carvalho et G. Roucairol, *On mutual exclusion in computer networks*, Communications of the ACM, Volume 26, Numéro 2, pp. 145–147, Février 1983.
  - G. Ricart et A.K. Agrawala, *Authors' response to "On mutual exclusion in computer networks"*, Communications of the ACM, Volume 26, Numéro 2, pp. 147–148, Février 1983.
  - B.A. Sanders, *The Information Structure of Distributed Mutual Exclusion Algorithms*, ACM Transactions on Computer Systems, Volume 5, Numéro 3, pp. 284–299, Août 1987.
  - M. Raynal, *A simple taxonomy for distributed mutual exclusion algorithms*, ACM SIGOPS Operating Systems Review, Volume 25 Numéro 2, pp. 47–50, Avril 1991.
- Interbocage :
  - G. Tel, *Chapter 10 : Snapshots*, dans Introduction to Distributed Algorithms, Cambridge University Press, pp. 331–351, 1994.
  - J. Brzezinski, J.-M. Hélary and M. Raynal, *A General Definition of Deadlocks for Distributed Systems*, dans Proc. 1st IEEE International Conference on Algorithms and Architectures for Parallel Processing, pp. 324–327, April 1995.
  - S.J. Rosenkrantz, R.E. Stearns, and P.M. Lewis, *System Level Concurrency Control for Distributed Database Systems*, ACM Transactions on Database Systems, Vol. 3, N. 2, june 1978.
  - K.M. Chandy and L. Lamport, *Distributed Snapshots : Determining Global States of Distributed Systems*, ACM Transactions on Computer Systems, 3(1) :63–75, February 1985.
- Détection de terminaison :
  - G. Tel, *Chapter 8 : Termination Detection*, dans Introduction to Distributed Algorithms, Cambridge University Press, pp. 264–302, 1994.

Index