# CHAPITRE I

# GÉNÉRALITÉS

*version du 14 octobre 2002*

## 1 Objectifs et organisation du cours

L'une des tendances majeures des systèmes informatiques est la répartition des traitements entre des "entités" coopératives. Celles-ci peuvent être soit des processeurs d'une machine multi-processeurs, soit des stations de travail d'un réseau local, soit des serveurs d'application connectés par l'Internet. Les avantages de la répartition sont nombreux : augmentation progressive de la puissance de calcul (par ajout de nouvelles entités), tolérance aux pannes, adaptation à l'utilisateur (choix de postes clients hétérogènes), découpage logique des applications (comme dans les modèles client-serveur), etc. [Tan94].

Cependant la conception d'applications réparties se heurte à des difficultés algorithmiques spécifiques à l'activité concurrente des entités et à l'absence de mémoire partagée entre celles-ci (excepté dans le cas particulier des machines multi-processeur). Autrement dit, le concepteur doit d'une part résoudre des problèmes liés à son application (par exemple messagerie, forum, etc.) mais aussi des problèmes liés à son modèle d'application comme la cohérence des données, la détection de la terminaison de l'application, l'exclusion mutuelle entre sections de code critiques, etc.

Une manière naturelle d'aborder la conception (qui se pratique en réseau depuis longtemps) consiste à construire l'application en couches, chaque couche s'appuyant sur l'interface de la couche inférieure et fournissant une interface de service à la couche supérieure. Dans notre cas, l'application sera divisée entre une couche applicative dont les traitements sont spécifiques aux fonctionnalités recherchées et une couche service qui résout des problèmes génériques et récurrents dans le domaine de la répartition.

L'objectif de ce cours est donc de décrire les principaux services nécessaires à la conception d'applications réparties. Sept domaines seront abordés dans la suite : la communication, le temps, la concurrence, la cohérence, la mémoire virtuelle, l'élection et l'auto-stabilisation. Chaque type de service fera l'objet d'un chapitre différent. Dans la suite de ce premier chapitre, on introduira le modèle de répartition sur lequel seront bâtis les services et on discutera des méthodes d'évaluation et de vérification des algorithmes répartis.

Ce document est destiné à des étudiants de quatrième année ou de cinquième année. Il est le résultat de plusieurs années d'enseignement en IUP GMI (mathématiques et informatique), en IUP MIAGE (informatique de gestion) et en DESS SITN (systèmes d'information et technologies nouvelles). L'intégralité du document correspond à 45h. d'enseignement. Dans le déroulement du cours, les étudiants sont confrontés à la réalisation d'un service. Ils découvrent les difficultés inhérentes à la nature du service. Puis ils élaborent (aidés par l'enseignant) le principe de la solution et enfin ils développent l'algorithme associé. Par la suite, ils procèdent à l'analyse de la solution (preuve, complexité, applicabilité,...). Il est recommandé de travailler sans ce support afin de stimuler la réflexion des étudiants.

Nous conseillons aux étudiants de consulter les trois ouvrages de M. Raynal [Ray91,Ray92a,Ray92b] qui couvrent un large spectre d'algorithmes pour compléter ce cours et d'étudier le livre de G. Tel [Tel00] - en anglais - qui se situe clairement au niveau d'un troisième cycle (D.E.A. ou doctorat) pour approfondir ce sujet.

## 2 Modèle de répartition

L'environnement réparti que nous considérons est composé :

- d'entités actives que nous appellerons indifféremment *sites* ou *stations* disposant d'une mémoire dédiée non accessible aux autres entités et d'un (ou plusieurs) processeur(s) qui partage son temps entre l'exécution de plusieurs processus. Chaque station dispose d'une identité unique et numérique. Cette adresse pourrait être l'adresse MAC de sa carte réseau ou son adresse IP.
- de *lignes de communication* bipoints (i.e. reliant deux stations) et bidirectionnelles (i.e. la transmission d'information est possible dans les deux directions). Chacune des directions est appelée un *canal*. Ces lignes sont le seul moyen d'échange d'information entre les stations. *Le graphe de communication* non orienté qui s'en déduit est obtenu en considérant les stations comme des noeuds et les liaisons comme des ar êtes.

Hypothèse n°1 Dans les chapitres 3,4,5,7 (sauf mention du contraire), nous supposons que ce graphe est une clique : toute paire de noeuds est reliée par une arête. Ceci correspond à la situation relativement courante des réseaux locaux. Le cas des réseaux étendus est abordé au chapitre 2. Enfin dans le cadre du développement de services Web, ce graphe correspond à une interconnexion logique entre serveurs et il peut être alors quelconque (cf. chapitres 6 et 8).

Nous ne nous intéressons à chaque station qu'en tant que composante d'une application répartie. Aussi nous découpons la station en trois couches. La couche application réalise les traitements spécifiques et fait appel à la couche service pour certaines fonctionnalités génériques. La réalisation de cette deuxième couche est l'objet de ce cours et nous désignons par *algorithme réparti* l'ensemble des codes associés. Pour l'échange des messages, la couche service se repose sur l'interface de la couche réseau dont la qualité de service dépend de l'environnement (figure 1.1). Nous détaillons ci-après les trois couches.

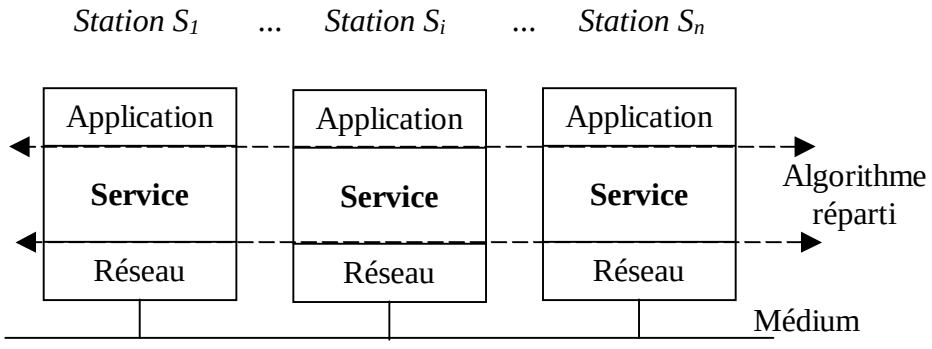



Figure 1.1 : L'environnement réparti

### *2.1 Le réseau et la couche réseau des stations*

Un canal de communication reliant un site $S_i$ à un autre site $S_j$ a les propriétés suivantes :

- les messages ne sont pas altérés,
- les messages ne sont pas perdus,
- le canal est fifo. Autrement dit l'ordre (temporel) de réception des messages de $S_j$ venant de $S_i$ est identique à l'ordre d'émission des messages de $S_i$ à destination de $S_j$.

Bien qu'un message soit certain de parvenir à son destinaire, deux situations sont à considérer. Soit le délai de transit d'un message est quelconque (par exemple Ethernet avec un médium partagé) soit il est borné par un délai maximum connu du concepteur de l'application (par exemple Token Ring). Dans le premier cas, on parle d'un *réseau asynchrone* alors que dans le second cas, on parle d'un *réseau synchrone*.

Hypothèse n°2 Sauf mention du contraire, nous supposons que le réseau est asynchrone. Ce choix se justifie d'une part parce qu'une application conçue pour un environnement asynchrone fonctionne aussi en environnement synchrone et d'autre part parce qu'exploiter la borne nécessite un choix souvent délicat de valeurs temporelles (comme le délai des chiens de garde ou "time-out"). Cependant nous discuterons des avantages d'un réseau synchrone dans le chapitre consacré au temps.

La couche réseau fournit un mécanisme de transfert de message pour la couche service. Deux options sont possibles pour ce mécanisme : des primitives synchrones (bloquant éventuellement le processus qui les appelle) ou asynchrones (non bloquantes). Une émission est synchrone si le processus émetteur est bloqué jusqu'à la réception d'un acquittement de son message (acquittement signifiant que l'application destinatrice l'a pris en compte). Une réception est synchrone si le processus destinataire appelle explicitement une primitive pour recevoir un message et se bloque éventuellement en attente de ce message. Chaque alternative a ses avantages : une communication synchrone conduit à des applications plus simples à vérifier et donc à concevoir, une communication asynchrone conduit à des applications plus flexibles où le concepteur peut choisir lui-même les points d'attente. Nous choisirons une communication asynchrone parce qu'il est relativement aisé d'émuler des primitives synchrones par des primitives asynchrones et que dans certains cas, l'asynchronisme de communication est plus adapté au problème considéré.

Plus précisément, les primitives d'émission retenues seront :

- `envoyer_à(destinaire,message)` qui permet d'envoyer un message au destinaire indiqué et de poursuivre son exécution. Le destinataire est désigné par son identité et le message peut être structuré selon les besoins de la couche service.
- `diffuser(message)` qui permet d'envoyer un message à tous les autres sites participant à l'application. L'utilisation de cette primitive n'est possible que si le graphe de communication est une clique.

Ces primitives seront appelées dans l'algorithme réparti par la couche service. La primitive de réception soulève un point important dans la définition d'une interface entre deux couches que nous détaillons maintenant à travers le trajet d'un message. Pour l'émission du message, la couche service appelle `envoyer_à`, puis le message circule sur le canal et parvient à la couche réseau (figure 1.2). Le traitement du message par la couche service dépend bien entendu du service mais puisque la réception est asynchrone, aucune primitive n'est appelée par la couche service. Autrement dit, la couche réseau doit appeler une primitive "écrite" par le service qui fait partie de l'algorithme réparti. Cette primitive est une primitive de l'interface que l'on appelle *montante* - écrite par la couche supérieure et appelée par la couche inférieure - par opposition aux primitives *descendantes* comme `envoyer_à`.

La primitive de réception sera :

- `sur_réception_de(émetteur,message)` qui spécifie le traitement à effectuer sur réception d'un message. Cette primitive est déclenchée par interruption du traitement courant par la couche réseau. L'émetteur est désigné par son identité et le message peut être structuré selon les besoins de la couche service. Pour simplifier l'écriture des algorithmes, cette primitive peut être dupliquée. La copie adéquate est appelée soit en fonction de l'identité de l'émetteur, soit en fonction du type de message (dans le cas d'un message structuré). Les paramètres jouent alors le rôle de filtre.

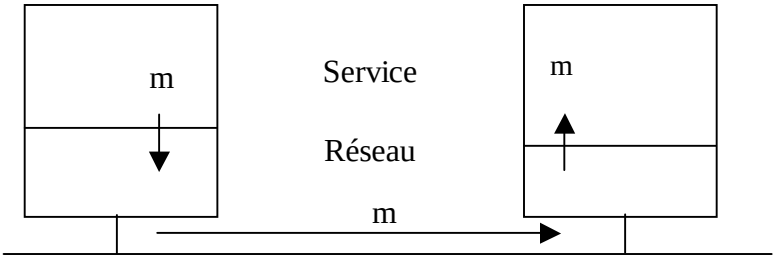



Figure 1.2 : Le trajet d'un message à travers les couches

Attention, il ne faut pas confondre le synchronisme de la couche réseau et celui du réseau. En effet, toutes les combinaisons sont possibles.

### *2.2 La couche service*

La couche service sera composée :

- Des variables qui lui sont propres, inaccessibles à la couche application (sauf peut-être en lecture). Une variable `var` d'une station `p` sera généralement désignée dans l'algorithme par $\texttt{var}_p$ pour insister sur le caractère local de la mémoire.
- Les primitives descendantes de l'interface application-service. Ces primitives sont en général la transcription de la fonctionnalité recherchée.
- Des primitives internes utilisées pour factoriser des traitements communs à plusieurs primitives.
- Occasionnellement, un processus interne au service lorsqu'une tâche devra être exécutée de manière concurrente au fonctionnement de l'application (par souci d'efficacité par exemple).
- Les primitives montantes `sur_réception_de` de l'interface service-réseau.

Le langage de programmation utilisé sera un pseudo-C. N'importe quel autre langage ferait aussi l'affaire. Certaines extensions méritent cependant d'être détaillées.

- Un processus d'application ou de service peut être suspendu (attente passive : autrement dit allocation du processeur à un autre processus) jusqu'à ce qu'une condition soit remplie (elle peut l'être immédiatement). La primitive utilisée sera `Attendre(Expression booléenne)`.
- Il peut aussi être suspendu jusqu'à l'expiration d'un time-out. Dans ce cas, la primitive utilisée sera `Attendre()`sans paramètre. La primitive qui arme le temporisateur sera `Armer(délai)`.
- Nous étendons les expressions booléennes avec les quantificateurs ∀ et ∃. Ceci sera principalement utilisé lorsque l'indice du quantificateur portera sur un ensemble de sites ou de messages.

### *2.3 La couche application*

Hypothèse n°3 Sauf mention du contraire, nous supposons que l'application est exécutée sur un site par un unique processus. Ceci simplifie la gestion du service. Si nécessaire, on peut généraliser la plupart des algorithmes au cas où l'application est exécutée par plusieurs processus (dont le nombre est connu) en dupliquant la couche service.

### *2.4 Déroulement de l'application*

Afin de fixer les idées, nous décrivons l'interaction entre les différentes couches lors du déroulement de l'application. Nous supposons que le système d'exploitation est capable de mettre en oeuvre les règles énoncées ci-dessous (ce qui est le cas de tous les systèmes réalistes).

- **Règle n°1** Si le processus applicatif exécute du code de la couche application, toute réception d'un message donne lieu à un déroutement et à l'exécution de la primitive `sur_réception_de` correspondante. Le traitement applicatif se poursuit ensuite.
- **Règle n°2** Si le processus applicatif (ou un processus de service) exécute du code de la couche service, l'exécution de la primitive `sur_réception_de` suite à une réception de message est retardée jusqu'à un blocage du processus (par la primitive `Attendre`) ou jusqu'au retour en couche application.
- **Règle n°3** Le code des primitives `sur_réception_de` ne comporte pas d'appel à la primitive `Attendre`. En effet ce code est exécuté par le système en interruption et ne correspond à aucun processus particulier.
- **Règle n°4** Si le système exécute une primitive `sur_réception_de` suite à une réception de message, tout exécution de `sur_réception_de` suite à une autre réception de message est retardée jusqu'à la fin de l'exécution de la première primitive.

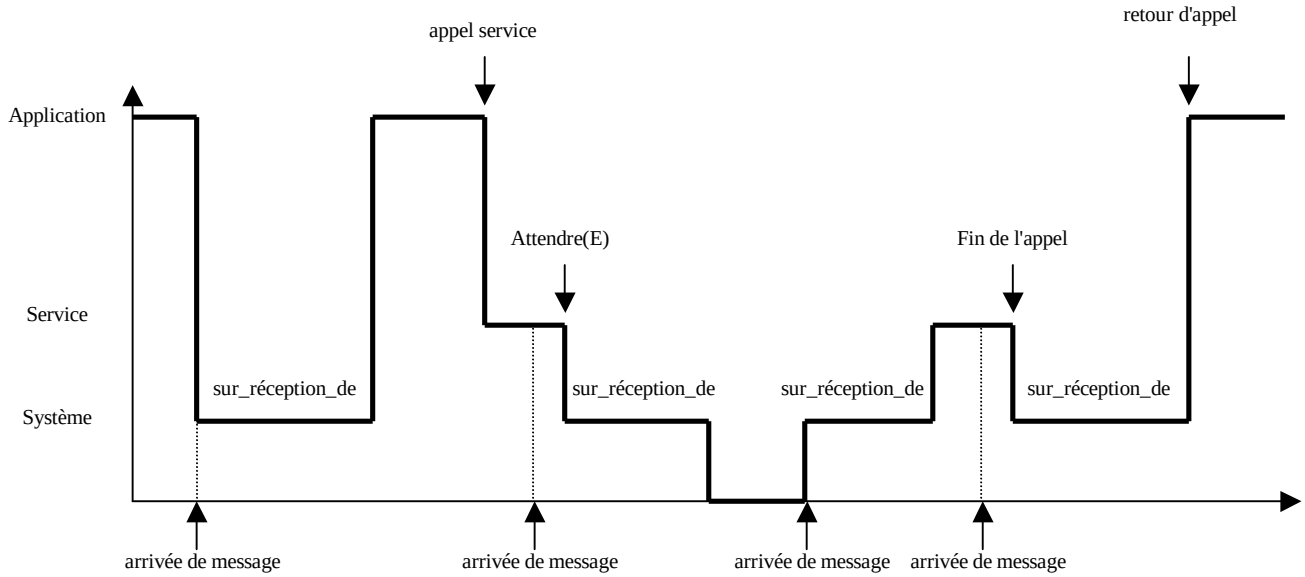



Figure 1.3 : Un déroulement d'une application sur un site

La figure 1.3 illustre le déroulement d'une application sur un site. On notera que la prise en compte des messages reçus dépend du niveau d'exécution courant. De plus la réception d'un message permet de rendre vrai la condition d'un `Attendre` qui bloque le processus dans la couche service.

# 3 Scénario, évaluation et vérification d'algorithmes répartis

## *3.1 Scénario d'exécution*

Afin d'illustrer les algorithmes, nous utiliserons un schéma de représentation connu sous le nom de *chronogramme* (figure 1.3). Dans ce schéma, l'activité de chaque site est représentée sur un axe temporel (ici horizontal) orienté de gauche à droite. Les transferts de message sont dénotés par des segments orientés dont l'origine et l'extrémité sont respectivement associées à l'envoi et à la réception de message. Nous noterons de plus le début et la fin d'une primitive sur un site par des crochets ouvrant et fermant. Une variante possible consiste à utiliser les axes verticaux où le temps croît du haut vers le bas.

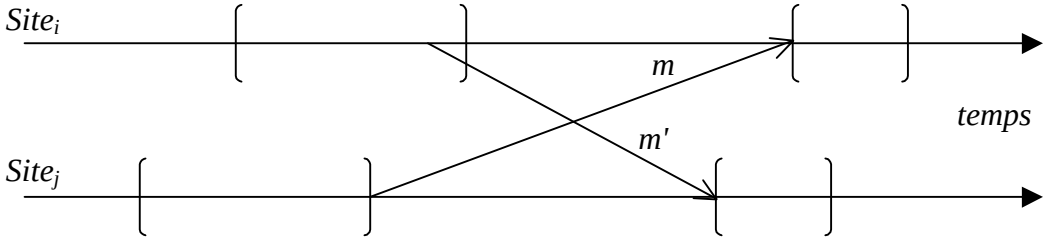



Figure 1.3 : Un exemple de chronogramme

## *3.2 Evaluation*

Les mesures usuelles de complexité en algorithmique standard sont la taille de l'espace occupé par l'algorithme et le temps d'exécution (ou de manière équivalente le nombre d'instructions exécutées) de l'algorithme.

En algorithmique répartie, sauf cas particulier (comme dans la gestion des tampons), l'espace n'est pas un facteur déterminant. De même, on considère que le temps d'exécution des primitives est négligeable devant le temps de transit des messages. Ceci nous conduit à deux mesures de complexité pertinentes pour un algorithme réparti.

- Le trafic sur le réseau décompté par le nombre maximum de messages échangés en fonction du nombre de sites de l'application (le plus souvent noté *n*). Une mesure plus exacte tiendrait compte de la taille des messages. Nous ne ferons pas cette distinction laissant le soin à l'étudiant de vérifier que dans nos comparaisons entre algorithmes, les tailles des messages sont similaires.
- Le temps d'exécution maximum de l'algorithme en fonction du nombre de sites de l'application. Comme nos algorithmes s'exécutent sur un réseau asynchrone, ce temps peut sembler difficile à évaluer. L'astuce habituelle consiste à considérer que le temps de transit du message le plus lent est de 1 unité de temps et donc que le temps de transit d'un message quelconque est compris entre 0 et 1. Le temps d'exécution du code est égal à 0. Contrairement à ce qu'on aurait tendance à imaginer, les exécutions les plus lentes ne correspondent pas nécessairement au cas où tous les temps de transit seraient égaux à 1. Nous n'évoquerons que brièvement cette mesure de complexité.

Nous ne nous intéressons qu'aux ordres de grandeur. Aussi lorsque nous écrivons $g(n)=\Theta(f(n))$, ceci signifie que lorsque $n$ tend vers l'infini le rapport de $g$ sur $f$ reste compris dans un intervalle fini de valeurs strictement positives.

### *3.3 Vérification*

La vérification des propriétés d'un algorithme réparti est un sujet difficile pour un étudiant de second cycle. Nous décrivons rapidement les problèmes à résoudre.

- Un algorithme réparti donne lieu à plusieurs exécutions possibles ne serait-ce qu'en raison du délai aléatoire de transit d'un message. On peut donc considérer comme représentation formelle du "comportement", l'ensemble des exécutions possibles. Cette solution simple n'est pas entièrement satisfaisante car elle masque les branchements entre les différentes exécutions possibles dans un état donné. De manière alternative, on peut considérer un arbre d'exécution dont les noeuds sont les états accessibles et les arcs sont les événements possibles dans cet état.
- Les propriétés à vérifier sont généralement de deux natures : des propriétés de sûreté qui s'expriment de la façon suivante "Dans tout état de l'algorithme, une assertation est vérifiée" et des propriétés de vivacité qui s'expriment de façon plus complexe. En voici deux exemples "De tout état, on peut (doit) atteindre un état qui vérifie une assertion" ou encore "Une assertion est vérifiée sur une infinité d'états de toute exécution".
- La vérification peut s'effectuer de manière manuelle en utilisant des techniques élémentaires comme l'induction qui consiste à vérifier que la propriété est vérifiée à l'état initial puis que, si elle est vérifiée en un état, elle est vérifiée dans tout état qui le suit. De manière plus élaborée, le vérificateur peut se faire aider un logiciel d'aide à la preuve (le plus souvent basé sur la logique). Enfin, dans le cas de systèmes à nombre d'états finis, la validation peut être entièrement automatisée par les techniques dites de "model-checking".

Nous établirons quelques preuves pour illustrer les méthodes de vérification mais aussi pour montrer qu'en s'appuyant sur des preuves d'un algorithme "abstrait", on peut construire des algorithmes "concrets".

## 4 Références

[Ray91] M. Raynal "La communication et le temps dans les réseaux et les systèmes répartis" Collection Direction des Etudes et des Recherches d'EDF n°75. Hermès. 1991 ISSN 0399-4198

[Ray92a] M. Raynal "Synchronisation et état global dans les systèmes répartis" Collection Direction des Etudes et des Recherches d'EDF n°79. Hermès. 1992 ISSN 0399-4198

[Ray92b] M. Raynal "Gestion de données réparties : problèmes et protocoles" Collection Direction des Etudes et des Recherches d'EDF n°82. Hermès. 1992 ISSN 0399-4198

[Tan94] A. Tanenbaum "Systèmes d'exploitation. Systèmes centralisés. Systèmes distribués" Troisième Edition Dunod – Prentice Hall. ISBN 2-10-004554-7. 1994

[Tel00] G. Tel "Introduction to Distributed Algorithms" Second Edition Cambridge University Press. ISBN 0-521-79483-8. 2000

# CHAPITRE II

# LA COMMUNICATION

*version du 21 novembre 2002*

## 1 Introduction

Dans ce chapitre, nous nous focaliserons sur la communication aussi bien à travers le mode de communication entre les entités qu'à travers le comportement du réseau dans la fonction de transit.

Tout d'abord, nous reviendrons sur la réception de messages. Un des avantages de la réception asynchrone est qu'un message est immédiatement traité (moyennant l'éventuel retard dû à l'arrivée d'un message lors de l'exécution d'une primitive de service). Dans le cas d'une réception synchrone, si l'application n'est pas immédiatement intéressée par l'arrivée d'un message celui-ci doit être stocké pour un usage ultérieur. Quelque soit la taille prévue du tampon des messages, un débordement est toujours possible si un mécanisme adéquat n'est pas prévu. Ce mécanisme s'appelle *le contrôle de flux* et nous l'étudierons à la fois dans un environnement local [Boc79] et dans un environnement étendu où les entités sont les noeuds de routage et où le graphe de communication n'est plus une cliq ue.

Un des objectifs de l'algorithmique répartie est la recherche de la symétrie car celle-ci est souvent synonyme de simplicité et d'efficacité de conception. Un des moyens d'y parvenir est de rendre les primitives de réception et d'émission quasiment identiques tout au moins en ce qui concerne la synchronisation. Ceci nous conduit au concept de *rendez-vous* que nous développerons dans la troisième section.

Enfin, nous montrerons que le fait que les canaux d'un réseau soient fifo n'empêche pas des comportements pathologiques dans le transit des messages. Aussi nous développerons le concept de réseau fifo qui garantit l'absence de tels comportements et nous montrerons comment émuler un réseau fifo au dessus d'un réseau asynchrone quelconque. Au coeur de cette problématique, se trouve *l' ordre causal*entre événements d'une exécution sur lequel nous reviendrons à plusieurs reprises.

# 2. Contrôle de flux

## *2.1 Contrôle point à point : le modèle producteur / consommateur*

2.1.1 Description du problème

Considérons deux sites, *P* le producteur et *C* le consommateur, tels que le producteur envoie des messages au consommateur via un canal (unidirectionnel). Lorsque l'application du producteur veut envoyer un message au consommateur elle appelle la primitive `produire(m)` où m est un paramètre par valeur de message. Lorsque l'application du consommateur veut recevoir un message du producteur, elle appelle la primitive `consommer(m)` où m est un paramètre par référence de message.

La vitesse d'exécution de *P* n'étant assujettie à aucune contrainte, il est possible que le rythme de production et d'émission de messages de *P* soit supérieur au rythme avec lequel *C* les consomme. Dans une telle situation, quelque soit le nombre d'emplacements prévus pour stocker les messages reçus et non encore consommés, ceux-ci peuvent se trouver occupés alors qu'un nouveau message arrive. Nécessairement soit le nouveau message est rejeté, soit l'un des messages stockés est écrasé par le nouvel arrivant. Nous devons donc mettre en place un contrôle de flux pour prévenir ce problème.

2.1.2 Description de la solution

Afin de résoudre ce problème dû à l'asynchronisme des sites *P* et *C*, une solution consiste à asservir la production et l'émission des messages par le site *P* à des informations de consommation fournies par *C*. Le producteur disposera ainsi d'autorisations de produire (initialement égales à la taille du tampon du consommateur). A chaque envoi, celui-ci consomme une autorisation. De son côté, le consommateur envoie une autorisation à la fin de chaque appel à `consommer` signifiant qu'une place s'est libérée dans le tampon. Bien entendu, le consommateur ne peut consommer que si son tampon n'est pas vide.

2.1.3 Algorithme

Nous débutons notre spécification par les deux variables de contrôle de chacun des sites.
- `Nbmess`, la variable du consommateur indiquant le nombre de messages dans le tampon
- `Nbcell`, la variable du producteur indiquant le nombre d'autorisations

D'autre part la gestion du tampon chez le consommateur amène à définir les variables suivantes :
- `T`, le tampon de taille `N` contenant les messages à consommer
- `in`, l'indice d'insertion dans le tampon
- `out`, l'indice d'extraction du tampon

Le tampon est géré de manière circulaire : lorsqu'un indice est en fin de tableau, il est remis à zéro.

***Algorithme du producteur***

```
Var  Nbcell : entier initialisé à N;
```

```
produire(m)
Début
     Attendre(Nbcell>0);
     envoyer_à(C,m);
     Nbcell = Nbcell - 1;
Fin

sur_réception_de(C,Ack)
Début
     Nbcell = Nbcell + 1;
Fin
```

***Algorithme du consommateur***

```
Var  Nbmess : entier initialisé à 0;
     in,out : 0..N-1 initialisé à 0;

consommer(m)
Début
     Attendre(Nbmess > 0);
     m = T[out];
     out = (out + 1) % N;
     Nbmess = Nbmess - 1;
     envoyer_à(P,Ack);
Fin

sur_réception_de(P,m)
Début
     T[in] = m;
     in = (in + 1) % N;
     Nbmess = Nbmess + 1;
Fin
```

Remarque : Lorsque la valeur de l'indice `in` est égale à celle de `out`, ceci signifie soit que le tampon est plein soit qu'il est vide. Aussi il est nécessaire d'introduire la variable `Nbmess` pour tester l'existence d'un message dans le tampon.

2.1.3 Un exemple de déroulement

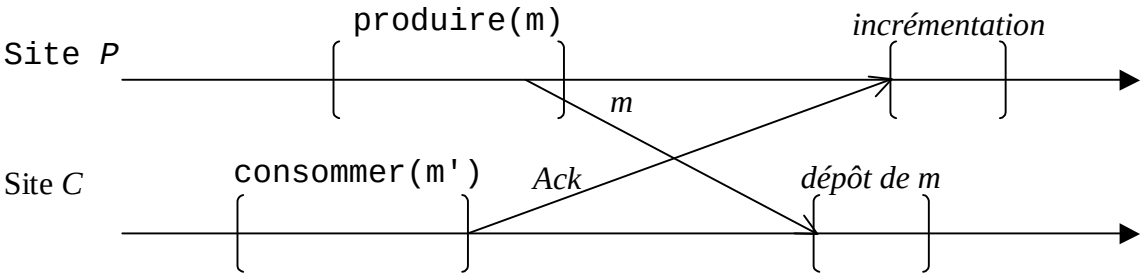



Figure 2.1 : Un extrait d'une exécution du producteur / consommateur

2.1.4 Vérification de l'algorithme

Nous nous limiterons ici à la preuve du non débordement du tampon. Nous allons démontrer que pour tout état accessible, l'égalité suivante est vérifiée : `Nbmess+Nbcell+Nbt=N` où `Nbt` désigne le nombre de messages en transit (application ou acquittement). Ceci garantit qu'un message d'application arrivant ne trouve jamais le tampon plein.

Nous le démontrons par induction. L'égalité est vérifiée initialement. Supposons la vraie dans un état donné et examinons l'effet de chacune des primitives :

- après un appel à `produire`, `Nbcell` est décrémenté et `Nbt` est incrémenté
- après un appel à `consommer`, `Nbmess` est décrémenté et `Nbt` est incrémenté
- après une réception d'un message, `Nbmess` est incrémenté et `Nbt` est décrémenté
- après une réception d'un acquittement, `Nbcell` est incrémenté et `Nbt` est décrémenté

Dans tous les cas, l'égalité reste vérifiée.

2.1.4 Une amélioration possible

L'envoi de messages d'acquittement peut se révéler coûteux puisque le contenu informatif de ces messages est nul : seule leur arrivée provoque une incrémentation. Une amélioration élémentaire consiste à remarquer que sur une liaison bidirectionnelle, chaque site (selon le canal) joue le rôle du producteur et du consommateur. Il suffit alors de retarder (selon un délai à configurer) l'envoi des acquittements. Si, avant l'expiration du délai, un message d'application doit être envoyé, on lui adjoint le nombre d'acquittements retardés et ceci à moindre coût. Dans le cas contraire, on envoie un message d'aquittements multiples, ce message étant envoyé sur un canal peu occupé (donc sans gêne pour l'application). Dans tous les cas, on réarme un nouveau "time-out" lors du prochain acquittement à envoyer.

## *2.2 Généralisation aux producteurs multiples*

On considère une généralisation du modèle producteur-consommateur dans lequel on a `p` sites de production $P_1,\ldots,P_p$ et toujours un seul site de consommation $C$. Comme dans le modèle précédent, le consommateur dispose d'un tampon de stockage de `N` cellules (figure 2.2).

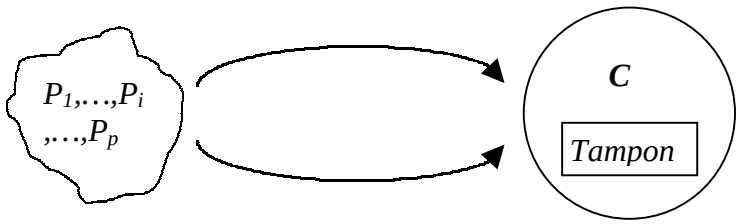



Figure 2.2 : `p` producteurs et un consommateur

Les interfaces qui définissent le service sont inchangées : `produire(m)` et `consommer(m)`. Une application évidente de ce schéma est la gestion des requêtes dans un modèle client-serveur : les producteurs sont les clients et le consommateur est le serveur. On ne s'intéresse pas ici au traitement des requêtes.

Le principal problème soulevé par ce modèle est le partage du tampon entre les différents producteurs.

2.2.1 Une première solution par partage statique

La solution la plus simple consiste à partager a priori (d'où le caractère statique de la solution) les N emplacements entre les p producteurs. Le site $P_i$ se voit attribuer $N_i$ places avec $\sum N_i = N$ (ce qui implique $N \geq p$). Dès lors il n' y a plus compétition entre les producteurs puisqu'ils ne partagent plus de ressources. D' un certaine manière, cette solution consiste à appliquer la solution précédente pour p canaux de communication moyennant un multiplexage des messages lors de la consommation. Chaque site producteur ne communique qu'avec le consommateur : au niveau du contrôle, le réseau *logique* est une étoile dont le site *C* est le centre.

Malheureusement cette solution conduit à une sous-utilisation des ressources. Considérons la situation où seuls q ($<< p$) producteurs désirent produire. Chacun de ces q sites va remplir sa partie du tampon et se trouvera ensuite momentanément bloqué jusqu'à ce que *C* consomme des messages alors qu'il y a de nombreuses cellules disponibles dans le tampon (celles attribuées aux autres producteurs). Il est bien connu que **ce type d'activité sporadique est caractéristique du comportement d'une application répartie**. Il s'agit donc là d'un inconvénient majeur lié à toute solution statique.

2.2.2 Distribution d' autorisations sur un anneau logique

La solution que nous allons exposer consiste à faire circuler des autorisations (correspondant à des cellules disponibles) sur un anneau logique constitué de la façon suivante : $C, P_1, P_2, ..., P_p, C$. Les producteurs sont placés en ligne, et cette ligne est bouclée par le site de consommation. A la structure de communication qu'est l'anneau est associé le moyen standard de l' exploiter le jeton. Il s'agit d'un message de contrôle spécial qui parcourt l'anneau de manière unidirectionnelle; il visite donc tous les sites et ceci de façon répétitive. Dans notre cas particulier, on associe au jeton une valeur `val` indiquant le nombre de cellules utilisables par les producteurs qui se serviront au passage du jeton.

La valeur du jeton décroît au cours des visites chez les producteurs et croît lors de la visite chez le consommateur du nombre de cellules consommées depuis le dernier passage du jeton. Cependant, il reste un problème à résoudre : de quel nombre d' autorisations a besoin un producteur lorsque passe le jeton ? Si on en reste à une généralisation immédiate de la solution précédente, l' application qui appelle `produire` se trouve bloquée jusqu' au passage du jeton. Une seule autorisation est alors nécessaire. Cette solution est très inefficace car le débit maximum de productions est d' un message par passage de jeton !

Il faut donc désynchroniser la production de messages du passage du jeton. Pour cela, on ajoute deux "ingrédients" chez chaque producteur : un tampon local et un processus de service appelé *le facteur*. L' application se contente de déposer les messages dans le tampon local et ne se bloque que lorsque le tampon est plein. Le facteur envoie les messages du tampon local à destination du consommateur. Pour ce faire, il se sert de deux variables indiquant le nombre de messages de son tampon et le nombre d' autorisations dont il dispose. Tant qu' il a une autorisation, il envoie un message. Lorsque le jeton arrive, le producteur cherche à obtenir

autant d'autorisations que de messages à envoyer pour lesquels il ne dispose pas d'autorisation. Nous détaillons maintenant l'algorithme.

***Algorithme du producteur $P_i$***

Chaque producteur $P_i$ maintient les variables locales suivantes :

- $T_i[0..N_i -1]$ : tableau contenant les messages produits par le producteur $P_i$.
- $in_i$ : indice d'insertion du tampon $T_i$, initialisé à 0.
- $out_i$ : indice d'extraction du tampon $T_i$, initialisé à 0.
- $nbmess_i$ : nombre de messages stockés dans $T_i$, initialisé à 0.
- $nbaut_i$ : nombre d'autorisations d'envoi de messages initialisé à 0.
- $Succ_i$ : identificateur du site successeur du site i. Si $i < m$ alors $Succ_i$ prend la valeur $P_{i+1}$ sinon la valeur de cette variable sera l'identificateur du site consommateur.
- $temp_i$ : variable de calcul temporaire. Elle prend la valeur minimale entre le nombre de cellules que le site désire réserver et le nombre de cellules libres associé au jeton.

Remarque : d'après la gestion de l'algorithme on aura à tout instant $nbmess_i \geq nbaut_i$.

```
produire (m)
Début
     Attendre(nbmess_i<N_i);
     T_i[in_i] = m;
     in_i = (in_i + 1) % N_i;
     nbmess_i++;
Fin
```

Lorsqu'un site $P_i$ reçoit le jeton de son prédécesseur sur l'anneau, il calcule $temp_i$, le minimum entre le nombre d'autorisations qu'il veut obtenir et la valeur associée au jeton. A partir de $temp_i$, il met à jour la variable val associée au jeton. Ensuite, il expédie ce dernier à son successeur sur l'anneau.

```
sur_réception_de(j,(jeton,val))
Début
     temp_i = Min((nbmess_i-nbaut_i),val);
     nbaut_i += temp_i;
     val -= temp_i;
     envoyer_à(succ_i,(jeton,val));
Fin
```

Comme pour la plupart des processus de service, le code de $Facteur_i$ consiste en une boucle infinie où chaque tour consiste en une attente (d'autorisations) suivi d'un traitement (l'envoi d'un message).

```
Facteur_i
Début
     Tant que(vrai)
          Attendre(nbaut_i>0);
          envoyer_à(consommateur,(app,T_i[out_i]));
          out_i = (out_i + 1) % N_i;
          nbaut_i--;
          nbmess_i--;
     Fin tant que
Fin
```

***Algorithme du consommateur***

Le site consommateur possède les variables suivantes :

- `T[0..N -1]` : un tableau contenant les messages des sites producteurs (le tampon du consommateur).
- `in` : l'indice d' insertion du tampon `T`, initialisé à `0`.
- `out` : l'indice d' extraction du tampon `T`, initialisé à `0`.
- `nbmess` : nombre de messages stockés dans `T` et non encore consommés initialisé à `0`.
- `nbcell` : nombre de cellules libérées entre deux passages du jeton initialisé à `0`.

```
sur_réception_de(j,(app,m))
Début
     T[in] = m;
     in = (in + 1) % N;
     nbmess++;
Fin

consommer(m)
Début
     Attendre(nbmess>0);
     m = T[out];
     out = (out+1) % N;
     nbmess--;
     nbcell++;
Fin

Sur_réception_de(j,(jeton,val))
Début
     val += nbcell;
     nbcell = 0;
     envoyer_à(P1,(jeton,val));
Fin
```

2.2.3 Un scénario d' exécution

Soit un système formé de trois producteurs $P_1$, $P_2$, $P_3$ et d'un consommateur $C$. Chacun des sites $P_1$ et $P_3$ dispose un tampon de taille 4. Ceux de $P_2$ et $C$ sont respectivement de taille 3 et 5. Nous examinons le comportement de ce système pendant deux phases. Chaque phase correspond à un tour du jeton.

Phase 1

Le consommateur envoie le jeton - avec la valeur `5` - vers le premier producteur. Ce producteur a produit le message $m_1$ et sa variable $\texttt{nbmess}_1$ vaut `1`. A la réception du jeton, le site $P_1$ obtient une autorisation d' envoi et $\texttt{nbaut}_1$ passe à `1` alors que la variable `val` passe à `4`.

Entre temps, le site $P_2$ produit `3` messages $m_2$, $m_3$ et $m_4$ qu'il stocke dans son tampon local $\texttt{T}_2$. et la variable $\texttt{nbmess}_2$ passe à `3`. Le processus applicatif de $P_2$ appelle encore une fois la

fonction `produire` et se bloque car son tampon est plein. A la réception du jeton, le site $P_2$ obtient trois autorisations d'envoi et $nbaut_2$ passe à `3` alors que la variable `val` passe à `1`.

Avant la réception du jeton, $P_3$ a déjà produit deux messages et sa variable $nbmess_3$ vaut `2`. Quand le jeton arrive, $nbaut_3$ passe à `1` ( `=min(1,2)` ) et `val` passe à `0`. A son tour $P_3$ expédie le jeton au site C, son successeur dans l'anneau (figure 2.3).

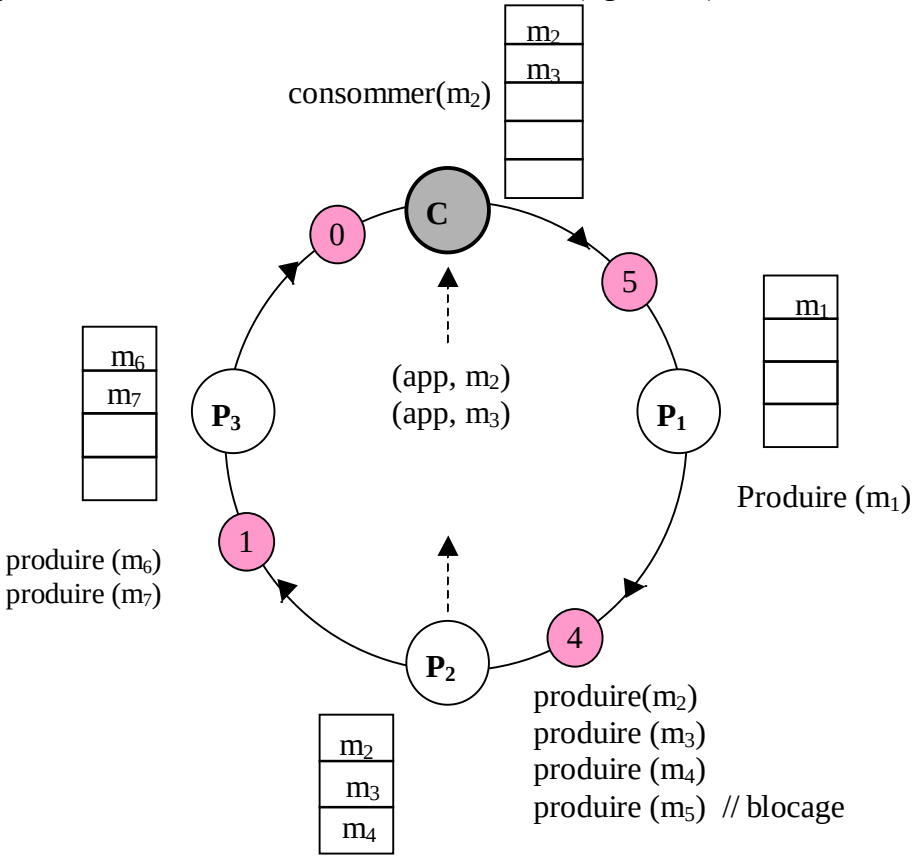



Figure 2.3 : Le premier tour du jeton sur l'anneau

Phase 2

Pendant ce temps, *C* a consommé un message incrémentant `nbcell`. A la réception du jeton avec la valeur `0`, il incrémente cette valeur avec `nbcell` et le jeton entame un nouveau tour avec la valeur `1`. Le site $P_1$ a déjà produit $m_8$, et donc a `2` messages dans son tampon et $nbaut_1$ positionné à `1`. A la réception du jeton, *$P_1$* met à jour la variable `val` qui passe à `0`. Ensuite, il expédie le jeton à son successeur dans l'anneau (figure 2.4).

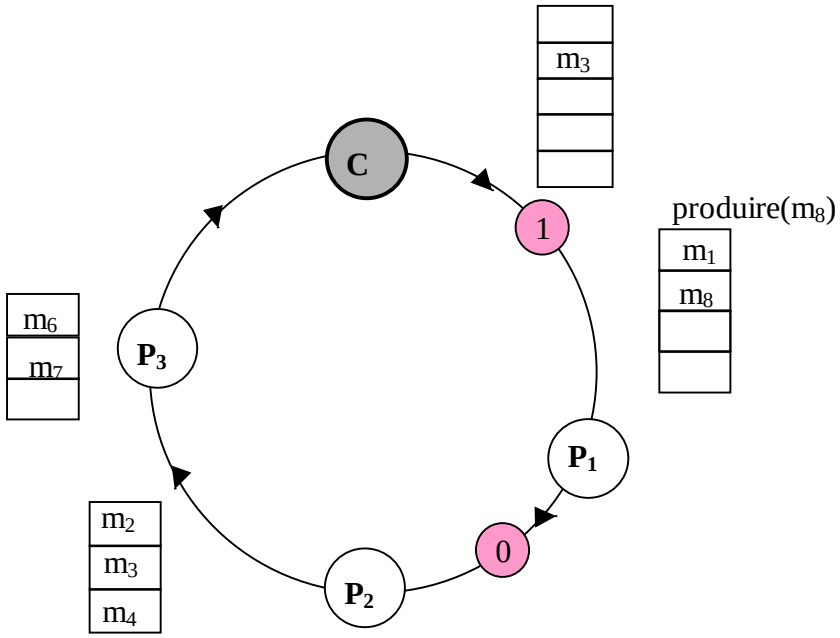



Figure 2.4 : Le deuxième tour du jeton sur l'anneau

Bien que chaque producteur voit passer le jeton une infinité de fois, il se peut que le jeton passe systématiquement devant lui à partir d'un certain tour avec la valeur nulle. Dans ce cas, le producteur restera en attente infinie d'autorisations. Seul $P_1$ est assuré de voir passer le jeton avec une valeur non nulle une infinité de fois. On pourrait bien sûr changer l'ordre sur l'anneau à chaque nouveau tour mais une solution plus radicale nous permettra à la fois de garantir l'équité mais aussi d'accroître l'efficacité.

2.2.4 Une solution équitable

Dans cette solution,

1. l'anneau est réduit aux sites producteurs qu'on numérote à présent de `0` à `p-1`.
2. chaque producteur voit passer le jeton avec une valeur supérieure à un seuil,
3. lorsqu'un producteur s'aperçoit qu'après sa mise à jour, le jeton contient une valeur inférieure au seuil, il le renvoie au consommateur,
4. le consommateur conserve le jeton jusqu'à ce que sa valeur soit supérieure à un deuxième seuil (supérieur ou égal au premier) et le renvoie au producteur suivant sur l'anneau.

Ainsi l'équité est garantie puisque que le jeton est toujours reçu avec une valeur supérieure au premier seuil. En règlant les deux seuils, on peut s'ajuster au comportement de l'application. Par exemple, le seuil des producteurs peut correspondre à une production moyenne entre deux passages de jeton tandis que le seuil du consommateur peut prendre en compte le nombre de producteurs pour éviter un retour trop rapide du jeton.

***Algorithme du consommateur***

On ajoute aux variables du consommateur, les variables suivantes :

- `Seuil` : constante entière. Elle correspond au seuil au delà duquel le consommateur envoie le jeton vers un producteur.
- `Présent` : booléen. Elle prend la valeur `VRAI` si le consommateur possède le jeton, `FAUX` dans le cas contraire. Elle est initialisée à `FAUX`.

- `Prochain` : identificateur du prochain producteur auquel le consommateur va expédier le jeton.

A chaque fois que le site *C* consomme un message, il incrémente `nbcell`. Au cas où il possède le jeton, *C* doit tester si cette nouvelle valeur est supérieure à la valeur de sa constante `Seuil`. Dans ce cas, il expédie le jeton au `Prochain` dans l'anneau.

```
consommer(m)
Début
     Attendre(nbmess>0);
     m = T[out];
     out = (out+1) % N;
     nbmess--;
     nbcell++;
     Si (Présent et nbcell > Seuil) Alors
          envoyer_à(Prochain,(jeton,nbcell));
          Présent = Faux;
          nbcell = 0;
     Fsi
Fin

sur_réception_de(j,(app,m))
Début
     T[in] = m;
     in = (in+1) % N;
     nbmess++;
Fin
```

A la réception du jeton, le site *C* détermine l'identité du prochain site producteur auquel il va expédier le jeton. Ensuite, il incrémente le nombre de cellules libres (`nbcell`) par le nombre de cellules non réservées par les producteurs durant le dernier passage du jeton. Si cette nouvelle valeur est supérieure à `Seuil`, alors *C* envoie le jeton à son prochain et réinitialise `nbcell`. Dans le cas contraire, il conserve le jeton.

```
Sur_reception_de(j,(jeton,val))
Début
     Prochain = (j+1)%p;
     nbcell += val;
     Si (nbcell > Seuil) Alors
          envoyer_à(Prochain,(jeton,nbcell));
          nbcell = 0;
     Sinon
          Present = Vrai;
     Fsi
Fin
```

***Algorithme du producteur $P_i$***

$P_i$ possède une nouvelle constante entière:

- $\texttt{seuil}_i$ initialisée à la même valeur pour tous les producteurs ($\texttt{seuil}_i \leq \texttt{Seuil}$)

Seule la réception de jeton est différente de la solution précédente.

```
sur_réception_de(j,(jeton,val))
Début
     tempi = Min(nbmessi-nbauti,val);
     val -= tempi;
     nbauti += tempi;
     Si (val > seuili) Alors
          envoyer_à((i+1)%p,(jeton,val)) ;
     Sinon
          envoyer_à(C,(jeton,val));
     Fsi
Fin
```

### *2.3 Contrôle de flux dans les réseaux étendus*

Dans les réseaux étendus, le transfert d'un paquet entre deux stations passe par des noeuds intermédiaires spécialisés : les routeurs. La principale tâche des routeurs est de déterminer le prochain noeud à qui envoyer un paquet en fonction de la destination finale de celui-ci. Le routage peut être calculé de façon centralisée ou distribuée. De plus, il peut être statique ou adaptatif. Dans le premier cas, le routage est établi une fois pour toutes alors que dans le second, de nouvelles routes peuvent être calculées en fonction du trafic ou de l'opérationnalité des lignes de communication ou des routeurs voisins.

**Hypothèse générale** Dans la suite, nous supposons qu'un routeur (ou une station pour un paquet entrant) conserve un paquet jusqu'à ce qu'il soit accepté par le prochain routeur. Si le paquet est arrivé à destination d'une station, le dernier routeur ne le conserve qu'un temps fini en l'absence de réponse de la station.

Un des problèmes liés à la circulation des paquets à travers le routeur est connu sous le nom de *congestion*. Supposons qu'un paquet soit reçu sur un routeur dont le tampon est plein. ce routeur rejettera le paquet. Dans ce cas, le routeur qui a émis le paquet devra le conserver dans son tampon, avec à son tour un risque de saturation de son tampon. Lorsqu'en une partie du réseau, ce phénomène se produit, on parle de congestion. Par analogie avec le trafic routier, on pourra imaginer que les tampons sont les routes et qu'un envoi de paquet correspond à un changement de route (les liaisons étant alors les carrefours). En présence de congestion, le réseau fonctionne au ralenti. *L'interblocage* est un cas extrême de congestion qui se produit lorsqu'un sous-ensemble de routeurs se retrouve avec ses tampons pleins et que le prochain noeud de n'importe quel paquet d'un ces tampons appartient à ce sous-ensemble. Dans l'exemple de la figure 2.5, le routeur X tente d'envoyer ces paquets aux routeurs Y et Z, qui eux aussi ont les mêmes intentions.

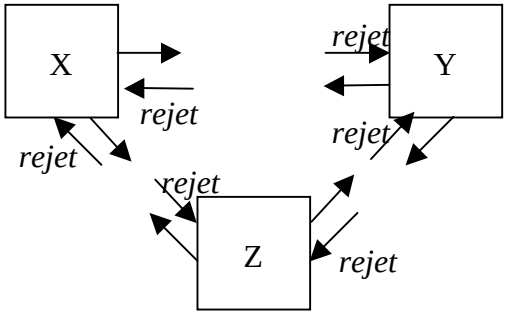



Figure 2.5 : Une situation d'interblocage

Deux types de stratégie sont possibles pour traiter la congestion et l'interblocage. Les stratégies dites *optimistes* consistent à adjoindre aux réseaux un mécanisme de *détection* qui reconnait de telles situations. Lorsque la congestion est détectée, un mécanisme de *guérison* se met en place. Par exemple, la détection pourrait consister à calculer le taux de saturation moyen du tampon, sur un intervalle donné et de tester qu'il est au-dessus d'un certain seuil. La guérison consisterait à supprimer des paquets et à envoyer vers les sources des flux des demandes de ralentissement ("feedback"). Les stratégies optimistes sont adaptées à des interconnexions de réseaux gérées par un ensemble d'opérateurs où la possibilité d'un contrôle global est illusoire.

Les stratégies dites *pessimistes* mettent en oeuvre un mécanisme de *prévention* de l'interblocage qui réduit par voie de conséquence les risques de congestion. Ces solutions nécessitent d'adopter la même politique sur tous les routeurs et donc l'existence d'un unique opérateur.

Nous présentons ci-dessous deux techniques de prévention des interblocages. Ce choix ne préjuge pas d'une appréciation sur la supériorité d'un des types de stratégie sur l'autre. Il se trouve simplement que les techniques optimistes sont des heuristiques et ne présentent donc pas un intérêt algorithmique majeur.

### 2.3.1 Structuration des tampons

Les techniques de structuration des tampons suppose que l'on exploite des informations sur le routage connues a priori (voir [MS80a] pour une approche générale). De manière évidente, plus un algorithme sera adaptatif moins l'on pourra s'appuyer sur le routage. La première information que nous allons exploiter est le nombre maximum `N` de routeurs traversés par un paquet. Dans le cas d'un routage statique optimal, `N` est le diamètre du graphe; dans le cas d'un routage statique quelconque, `N` est borné par le nombre de routeurs. Si le routage est adaptatif, l'établissement d'une borne s'obtient par analyse de l'algorithme utilisé. Certains algorithmes ne bornent pas le nombre de routeurs traversés et sont inadéquats pour supporter cette technique.

Le tampon d'un routeur est partionné en `N` sous-tampons indicés de 1 à `N`. Les règles ci-dessous définissent des contraintes de placement lors de la réception d'un paquet en fonction de sa provenance. Ceci implique que parmi les informations de contrôle d'un paquet se trouve l'indice de son paquet d'origine.

**R1** Un paquet entrant (c'est à dire provenant d'une station connectée au réseau) est inséré dans le sous-tampon d'indice `1`.

**R2** Un paquet provenant d'un sous-tampon indice `i` est placé dans le sous-tampon d'indice `i+1`.

D'après l'hypothèse faite sur le routage, le sous-tampon d'indice `i+1` existe toujours (même s'il est éventuellement plein).

**Proposition** L'application des règles précédentes prévient l'interblocage.
Preuve
Nous raisonnons par l'absurde. Supposons qu'il existe un ensemble de sous-tampons se bloquant mutuellement. Autrement dit, ces sous-tampons sont pleins et tout paquet d'un de ces

sous-tampons doit être expédié vers un autre sous-tampon de cet ensemble. Choisissons l'un de ces tampons `T` d'indice `i`. En vertu de la règle 2, `T` est bloqué par des sous-tampons d'indice `i+1`. En appliquant le même raisonnement à ces sous-tampons, on parvient, par itération, à la conclusion qu'au moins un sous-tampon `T'` d'indice `N` est en interblocage. Or par hypothèse sur le routage, ces paquets sont des paquets sortants (à destination des stations). D'après l'hypothèse générale, ces paquets ne sont pas conservés indéfiniment et `T'` ne peut donc être en interblocage. D'où la contradiction. ◊◊◊

*Une amélioration*

Cette solution est relativement inefficace car dans la mesure où `N` est le nombre maximum de routeurs traversés, peu de paquets traverseront les sous-tampons d'indice élevé entrainant une sous-utilisation de ces ressources. On pourrait bien sûr choisir de sous-dimensionner un tampon d'indice élevé par rapport à un tampon de plus faible indice. Le problème de la sous-utilisation persisterait alors mais à un degré moindre.

Pour résoudre plus radicalement le problème, nous supposons que le routage nous permet de connaitre une borne sur le nombre de routeurs traversés par paire (source,destination). Ainsi quand un paquet `p` pénètre dans le réseau, on adjoint à ses informations de contrôle, cette borne $N_p$. Très souvent, la borne $N_p$ sera largement inférieure à `N`. A chaque routeur traversé, la borne est décrémentée. Ainsi elle indique à tout moment, une borne sur le nombre de routeurs qui restent à traverser (en incluant le routeur qui reçoit le paquet). Nous allons relâcher les contraintes des règles **R1** et **R2**, qui se réécrivent comme suit :

**R1** Un paquet entrant `p` qui doit traverser au plus $N_p$ routeurs est inséré dans un sous-tampon dont l'indice appartient à l'intervalle $[1..N-N_p+1]$.

**R2** Un message provenant d'un sous-tampon d'indice `i` et ayant encore à traverser au plus $N_p'$ routeurs est placé dans un sous-tampon dont l'indice appartient à $[i+1..N-N_p'+1]$.

La règle 2 est toujours applicable car sur le routeur précédent l'indice `i` du sous-tampon vérifie $i \leq N-(N_p'+1)+1$ (en vertu de la règle 1 ou 2) ce qui est équivalent à $i+1 \leq N-N_p'+1$.

**Proposition** L'application des règles précédentes prévient l'interblocage.
Preuve
La preuve est similaire à la précédente en remarquant que la contradiction provient du fait que les paquets d'un sous-tampon sont bloqués en attente de place dans un sous-tampon d'indice supérieur. ◊◊◊

Plusieurs choix de sous-tampons sont possibles à la réception d'un nouveau paquet et il est judicieux de sélectionner le sous-tampon non plein d'indice le plus faible. En effet, cela laisse un intervalle plus grand pour le choix lors du prochain noeud à traverser. Dans [TU81], ce type de technique est employé sans décomposition en sous -tampons.

2.3.2 Estampillage des paquets

Nous souhaitons élaborer une technique de prévention ne reposant pas sur une connaissance du routage. Aussi nous ferons cette fois-ci l'hypothèse que chaque routeur dispose d'une

horloge qui progresse avec le temps. Nous n'exigeons pas que les horloges soient synchronisées (voir le chapitre sur le temps). Cependant, plus les horloges seront synchronisées, plus le traitement des paquets en cas de congestion sera équitable. Le principe de cette solution consiste à privilégier les paquets les plus âgés sachant qu'un paquet demeurant dans le réseau finira par être âgé et bénéficiera de cette priorité. Nous devons tout d'abord définir un âge de telle sorte que deux paquets n'aient jamais le même âge.

**Définition** Un âge est un couple (heure,identité de site). Soient deux âges $(h_1,i_1)$ et $(h_2,i_2)$ alors $(h_1,i_1)<(h_2,i_2)$ si et seulement si :
1. $h_1<h_2$ ou
2. $h_1=h_2$ et $i_1<i_2$

Lorsqu'un paquet pénètre sur le réseau, le premier routeur lui ajoute une information de contrôle appelée *estampille* (qui sera son âge) constituée de son heure d'arrivée et de l'identité de ce routeur. Deux messages ne peuvent jamais avoir le même âge car si les heures sont les mêmes alors les identités sont différentes.

Nous supposons de plus que chaque routeur dispose d'une cellule (tampon réduit à un élément) par ligne entrante. Cette cellule sera appelée *cellule d' échange* Nous définissons maintenant le traitement d'un paquet p venant du routeur X et arrivant sur le routeur Y.

**R1** Si le tampon de Y n'est pas plein, alors Y stocke le paquet p.

**R2** Si le tampon de Y est plein et qu'il a un paquet p' à expédier à X, alors Y stocke p dans la cellule d'échange associée à X si celle-ci est vide et envoie p' vers X "pour échange" avec p. Lorsque p' arrive, X le place dans la cellule occupée par p. A la réception de l'acquittement de p', Y place à son tour p dans la cellule occupée par p' et libère la cellule d'échange. Si la cellule d'échange est pleine, il rejette le paquet en indiquant "cellule d'échange pleine".

**R3** Si le tampon de Y est plein, qu'il n'a pas de paquets à expédier à X mais que le plus jeune des paquets de son tampon p' est plus jeune que p alors il procède comme indiqué la règle 2. Ceci revient à détourner p' de sa route initiale.

**R4** Si aucune des règles précédentes n'est applicable alors Y rejette p pour cause de "tampon plein".

Nous spécifions aussi la règle du choix des paquets à envoyer par un routeur sur une ligne :
1. si celui-ci a des paquets pour échange, il les envoie sur la ligne en fonction de l'ordre d'arrivée dans le tampon ;
2. en dehors de ces paquets, si le routeur a des paquets rejetés pour cause de "cellule d'échange pleine", il renvoie uniquement le paquet dont le rejet est le plus ancien jusqu'à ce qu'il soit rejeté pour "tampon plein" ou accepté (en l'intercalant régulièrement si nécessaire entre des paquets pour échange).

**Proposition** L'application des règles précédentes prévient l'interblocage.

Preuve

Premièrement une cellule d'échange ne reste jamais indéfiniment pleine puisque le paquet pour échange correspondant sera envoyé (règle des envois), reçu, accepté et acquitté.

Notons ensuite qu'un paquet rejeté pour cause de "cellule d'échange pleine" finira par être accepté ou rejeté pour "tampon plein". Si tel n'est pas le cas, alors soit ce paquet est réémis indéfiniment sur la ligne, soit un autre paquet rejeté antérieurement pour la même raison est réémis indéfiniment. Ceci signifie qu'un tel paquet trouvera indéfiniment la cellule pleine. En raison du premier point, cela signifie qu'elle se remplit indéfiniment. Or la cellule d'échange n'est pas utilisée par les paquets "pour échange" et ce sont les seuls autres paquets émis par le routeur. D'où la contradition.

Pour démontrer l'absence d'interblocage, nous raisonnons par l'absurde. Supposons qu'il existe un scénario d'envois de paquets sur le réseau tel qu'au moins un paquet reste indéfiniment dans le réseau. Puisque les horloges croissent, parmi ces paquets il en existe un plus âgé noté p (pas nécessairement le premier entré sur le réseau). L'ensemble des paquets plus âgés que p est fini toujours en raison de la croissance des horloges. Aucun de ces paquets ne reste indéfiniment dans le réseau d'après la définition de p. Donc il existe un instant à partir duquel p est (et restera) le plus âgé des paquets du réseau. Mais dans ce cas, ni la règle R3 où il jouerait le rôle de p', ni la règle R4 ne lui sont plus applicables. De plus, il ne peut être rejeté indéfiniment pour cause de "cellule d'échange pleine" (voir le point précédent) donc il sera accepté au bout d'un temps fini par tout routeur. Par conséquent, il sortira du réseau au bout d'un temps fini. D'où la contradiction.◊◊◊

# 3 Communication synchrone : le rendez-vous

## *3.1 - Le schéma du rendez-vous*

Le rendez-vous est un schéma de communication entre processus supporté par certains langages (ADA, CSP, Occam, etc.). Dans sa forme la plus simple, ce schéma met en jeu deux sites. La communication se déroule en deux phases. Durant la phase de *synchronisation*, le premier processus qui appelle la primitive de rendez-vous est bloqué jusqu'à l'appel correspondant par le deuxième processus. S'engage alors la phase d'*échange de données* dont les modalités sont propres au langage et qui ne présente pas d'intérêt algorithmique. Aussi dans la suite, nous nous limiterons à l'étude de la phase de synchronisation.

Dans une forme plus élaborée, un site peut souhaiter un rendez-vous avec un site choisi parmi un ensemble fixé lors de l'appel à la primitive. Ceci peut être le cas quand :

- un service est assuré par un ensemble de serveurs redondants. Dans ce cas, un client souhaite un rendez-vous avec l'un quelconque des serveurs pour soumettre sa requête.
- un service est assuré par un serveur sécurisé qui n'accepte des requêtes que d'un ensemble de clients identifiés. Dans ce cas, le serveur souhaite un rendez-vous avec l'un quelconque de ces clients pour traiter sa requête.

C'est cette forme que nous allons étudier [Bag89]. Enonçons tout d'abord les propriétés attendues de notre algorithme :

- A aucun instant, le service ne peut engager l'application dans deux rendez-vous simultanément. Ceci est un exemple de propriété de sûreté.
- Si à un instant donné, un rendez-vous est possible en raison des différents appels en cours alors un rendez-vous doit avoir lieu au bout d'un temps fini. Ceci est un exemple de propriété de vivacité. Notons que rien n'est précisé sur l'identité des participants au rendez-vous.

## *3.2 Principes de la solution*

Afin de dégager les principes d'une solution au problème du rendez-vous, nous allons d'abord exhiber deux problèmes que tout algorithme doit traiter. Nous illustrons ces problèmes à l'aide d'exemples.

**Exemple 1** Soit une application répartie sur trois sites $S_1$, $S_2$ et $S_3$. L'application sur chacun des sites souhaite à peu près au même moment obtenir un rendez-vous avec l'un quelconque des deux autres sites. Sans préjuger d'une solution, on peut penser que pour obtenir un rendez-vous le service de chaque site interroge le service d'un autre site pour savoir si le rendez-vous est aussi souhaité.

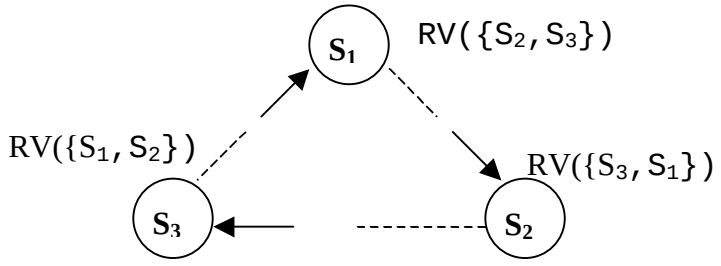



Figure 2.6 : Des demandes de rendez-vous symétriques

Ainsi sur la figure 2.6, le service de $S_1$, suite à l'appel par l'application de `RV({S2,S3})` envoie une requête de rendez-vous au premier site de sa liste $S_2$. De manière similaire, $S_2$ a envoyé sa requête à $S_3$, celui ayant fait de même avec $S_1$. Examinons les différents choix qui s'offrent à $S_2$ lors de la réception de la requête de $S_1$.

**Comportement 1** Comme $S_2$ "est provisoirement engagé" par sa requête à $S_3$, il rejette la requête du site $S_1$. Par symétrie, chacun des autres sites $S_1$ et $S_3$ rejettent les requêtes reçues. La situation peut se reproduire avec le second élément de la liste. Ainsi bien que des messages soient continuellement échangés, l'algorithme ne progresse pas vers un rendez-vous. Ce type de blocage est appelé "livelock" au sens où bien que les services soient actifs, cette activité est inutile.

**Comportement 2** $S_2$ accepte la requête de $S_1$ et annule celle faite à $S_3$. Par symétrie, $S_1$ et $S_3$ annulent les requêtes envoyées. La situation est similaire à la précédente avec un surcoût en nombre de messages échangés. Il s'agit ici aussi d'un "livelock".

**Comportement 3** De manière opportuniste, $S_2$ attend une réponse de $S_3$. pour rendre sa réponse à $S_1$ : si $S_3$ rejette sa requête il accepte celle de $S_1$ sinon il la rejette. Dans tous les cas de figure, il semble assuré d'un rendez-vous. Malheureusement par symétrie, les autres sites font de même et tous les sites restent bloqués en attente d'une réponse. On affaire ici à un interblocage de communication. Cette fois-ci, aucune activité n'est plus présente. On appelle cette situation un "deadlock".

On se trouve confronté à l'un des paradoxes de l'algorithmique répartie. Pour simplifier la conception, on recherche la symétrie mais **les solutions complètement symétriques ne garantissent pas la progression de l'algorithme**. Il faut introduire à faible dose l'asymétrie. Ceci peut se faire avec un code identique en s'appuyant sur ce qui distingue chaque site, leur identité. Ici selon l'identité de l'émetteur d'une requête reçue alors que le site attend la réponse à sa propre requête, celui-ci choisira entre le comportement 1 et le comportement 3. Ainsi supposons que $S_i$ attende la réponse de $S_k$ et qu'il recoive d'une requête venant de $S_j$. Dans ce cas, si j > i alors $S_i$ retarde sa réponse à $S_j$ sinon il rejette cette requête.

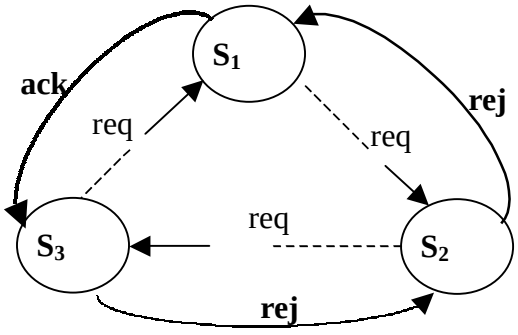



Figure 2.7 : Asymétrie de comportement

La figure 2.7 illustre l'application de cette règle à l'exemple précédent : $S_1$ retarde sa réponse, $S_3$ (respectivement $S_2$) rejette la demande de $S_2$ (respectivement $S_1$). $S_1$ peut répondre favorablement à $S_3$ et un rendez-vous aura lieu.

Dans le cas général, l'absence d'interblocage se vérifie par un raisonnement similaire à celui développé lors de la structuration des tampons en notant qu'un site ne peut être bloqué que par un site d'identité inférieure.

**Exemple 2** Soit une application répartie sur deux sites $S_1$ et $S_2$. Supposons que l'application du site $S_1$ désire un rendez-vous avec $S_2$. Comme précédemment, le service émet une requête de rendez-vous. A la réception, le service du site $S_2$ dont l'application n'est pas à cet instant interessée par un rendez-vous rejette la requête. Que doit faire le service du site $S_1$ à la réception de ce rejet? Il pourrait après un délai, réemettre sa requête. Ceci soulève le problème de la configuration du délai : trop court, il peut engendrer un trafic inutile sur le réseau; trop long, il peut retarder le fonctionnement de l'application.

On remarque que dans le contexte du rendez-vous, il est inutile de retransmettre une requête. Quand l'application du site $S_2$ souhaitera un rendez-vous, son service enverra une requête aussitôt acceptée. Autrement dit, puisqu'il faut être deux pour le rendez-vous, il suffit qu'à tout instant l'un des deux ait l'initiative du rendez-vous. La mise en oeuvre de cette prise d'initiative se fait par l'intermédaire d'un jeton par paire de sites {i,j}. Le site qui possède le jeton a le droit d'envoyer une requête et il perd ce droit avec l'envoi de telle sorte qu'il n'y aura jamais de réémission et ceci sans perdre la possibilité du rendez-vous. La répartition initiale des jetons sur les sites est arbitraire ; par commodité de programmation, on donnera un jeton associé à une paire de sites au site de plus grande identité.

### *3.3 Graphe d'état d'un service de rendez-vous*

Avant d'écrire l'algorithme, nous allons en donner une représentation abstraite à l'aide d'un graphe d'état du service d'un site. Les états sont représentés par les noeuds du graphe et les arcs correspondent aux transitions d'états. Chaque transition respecte la syntaxe :

$$\text{garde} \rightarrow \text{actions}$$

où la garde est une expression booléenne traduisant les conditions d'occurence de la transition et les actions indiquent les modifications des variables locales du service. La garde ou les actions peuvent être omises.

De plus, afin de rendre plus compacte la représentation, nous notons `j?m` la réception d'un message `m` venant du site `j` par le service et `j!m` l'envoi par le service au site `j` du message `m`.

Lorsque l'application du site `i` n'a pas appelé la primitive de rendez-vous, le service est au `repos`. A l'appel de la primitive de rendez-vous `RV`, la variable $E_i$ est initialisée avec le sous-ensemble des sites, possibles correspondants du rendez-vous. Le site passe à l'état `encours`. Dans cet état, le site recherche un $candidat_i$ c'est à dire un site de $E_i$ pour lequel il possède le jeton associé à la paire $\{i, candidat_i\}$. La présence des jetons est mémorisée dans le tableau de booléens $jeton_i$. Après l'envoi d'une requête à $candidat_i$, le service passe alors à l'état `attente-sans-retarder`. A la réception d'un acquittement, il passe à l'état `succès` puis au retour de la primitive (ayant renvoyé à l'application le site retenu) de nouveau au `repos`. En cas de rejet, il retourne à `encours` pour rechercher un autre $candidat_i$. Alors que le site est dans l'état `attente-sans-retarder`, celui-ci peut recevoir une requête qui lui conviendrait; il applique alors la procédure décrite dans l'exemple 1 et passe à l'état `attente-en-retardant` où le site retardé est indiqué par la variable $retardé_i$. Dans cet état, quelque soit la réponse du

$\texttt{candidat}_i$, le site `i` passera à l'état `succès` avec éventuellement un changement d'identité de $\texttt{candidat}_i$. Il devra de toutes façons envoyer une réponse à $\texttt{retardé}_i$.

Dans un état où le site n'est pas en cours de service (`repos`) ou éventuellement bloqué (`encours`, `attente-sans-retarder`, `attente-en-retardant`), il faut prendre en compte les éventuelles réceptions de requêtes. Deux cas méritent une explication. Si l'on reçoit une requête dans l'état `attente-en-retardant` on la rejette car, de toutes façons, on est assuré d'un rendez-vous. Si l'on reçoit une requête d'un site de $E_i$ dans l'état `encours`, on l'accepte et on passe directement à l'état `succès`.

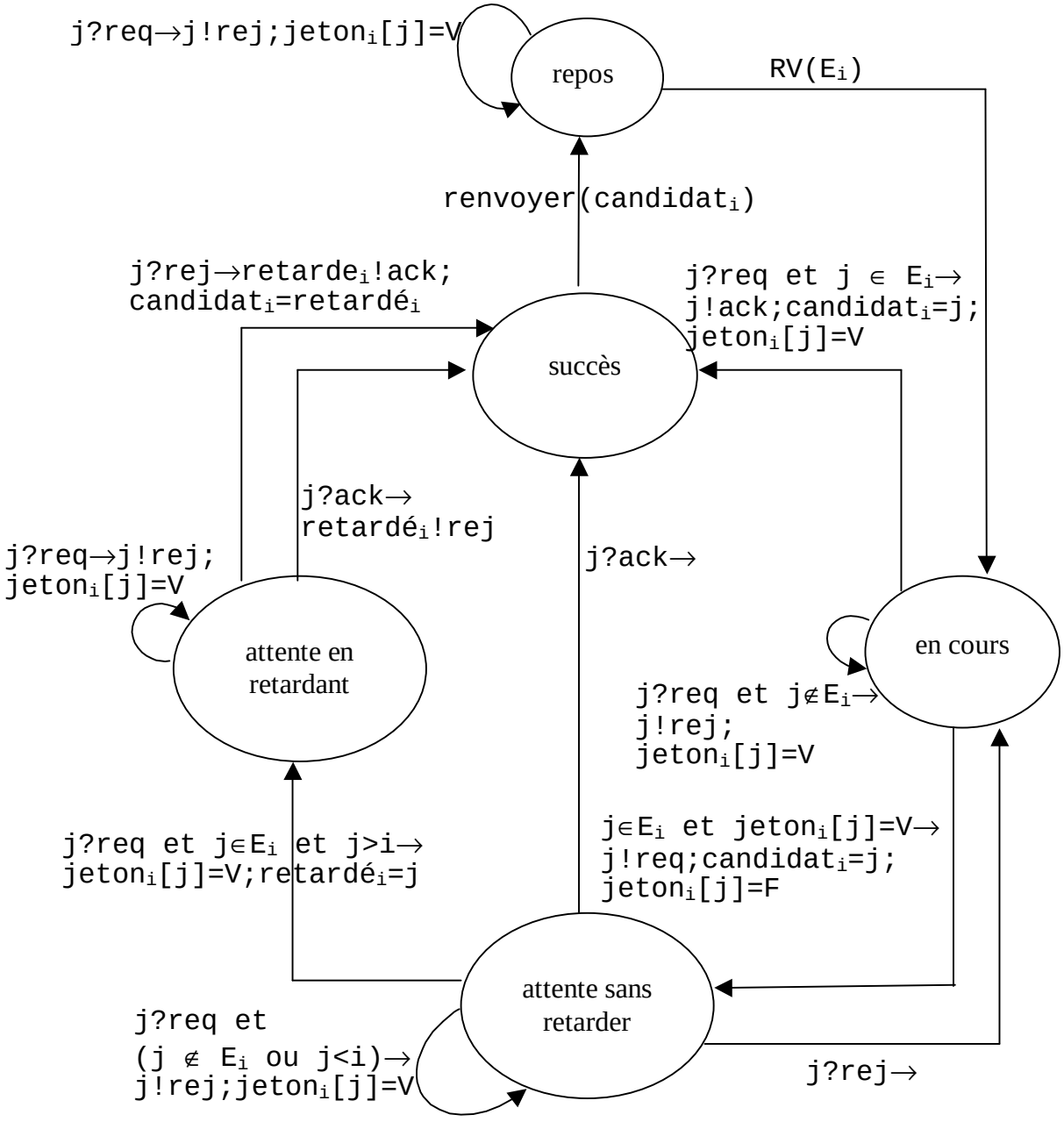



Figure 2.8 : Graphe d'états d'un site

### *3.3 Algorithme du rendez-vous*

Variables du site i

- $E_i$ : ensemble des identités des sites avec lesquels `i` désire un rendez-vous.
- $état_i$ : état du site à valeur dans {repos,encours,attente,succès} initialisé à repos
- $retardé_i$ : identité du site retardé par `i`. Par convention lorsque `i` ne retarde aucun site, sa valeur est `i` (sa valeur initiale).
- $jeton_i[1..N]$ : tableau de booléens indiquant la présence des jetons. Initialement, $jeton_i[j] = (i \geq j)$.
- $candidat_i$ : identité du site candidat courant.

L'interface de service se résume à la primitive $RV(E_i)$ qui doit renvoyer l'identité d'un site de $E_i$ avec lequel le rendez-vous est engagé. A chacun des trois types de messages est associé un traitement. Notons qu'une réception ne peut intervenir dans l'état `succès` car c'est un état dans lequel on ne se bloque pas (comme `repos`) et qui n'intervient que durant la primitive de service (contrairement à `repos`).

**RV($E_i$)**

```
Début
     état_i=encours;
     Répéter
(1)  Attendre(∃candidat_i ∈ E_i et jeton_i[candidat_i]==Vrai);
          Si état_i==encours Alors
               envoyer_à(candidat_i,req);
               jeton_i[candidat_i]=Faux;
               état_i=attente;
(2)            Attendre(état_i!=attente);
          Fsi
     Jusqu'à état_i==succès;
     état_i=repos;
     renvoyer(candidat_i);
Fin
```

(1) Si l'attente ne bloque pas le site `i` alors un candidat a été trouvé à qui envoyer une requête. Dans le cas contraire, seule la réception d'une requête lui permettra de sortir de l'attente avec l'état positionné à `succès`.
(2) L'attente est toujours bloquante et on en ressort soit dans l'état `succès`, soit dans l'état `encours` auquel cas `i` s'engage pour un tour de boucle supplémentaire.

**sur_réception_de(j,ack)**

```
Début
     état_i=succès;
     Si retardé_i!=i Alors
          envoyer_à(retardé_i,rej);
          retardé_i=i;
     Fsi
Fin
```

**sur_réception_de(j,rej)**

```
Début
     Si retardé_i=i Alors
          état_i=encours;
     Sinon
          état_i=succès;
          envoyer_à(retardé_i,ack);
          candidat_i=retardé_i;
          retardé_i=i;
     Fsi
Fin

sur_réception_de(j,req)
Début
     jeton_i[j]=Vrai;
     Si (état_i==repos) ou (j ∉ E_i) Alors
          envoyer_à(j,rej);
     Sinon si état_i==encours  Alors
          état_i=succès;
          envoyer_à(j,ack);
     Sinon si (retardé_i!=i) ou (j<i) Alors
          envoyer_à(j,rej);
     Sinon
          retardé_i=j;
     Fsi
Fin
```

# 4 Qualité de service : le réseau fifo

## *4.1 Un exemple introductif*

Supposons qu'une base de données soit dupliquée sur trois sites afin d'assurer une tolérance aux pannes et de diminuer les temps de réponse par répartition de la charge. Une lecture ne pose aucun problème de gestion tandis qu'une mise à jour de la base nécessite un mécanisme de synchronisation pour garantir la cohérence entre les différentes copies.

Décrivons tout d'abord un mécanisme très simple reposant sur la circulation d'un jeton entre les trois sites. Lorsqu'un site désire $S_i$ effectuer une mise à jour, il attend le passage du jeton. Puis $S_i$ modifie sa copie et diffuse un message de mise à jour MAJ contenant les modifications (figure 2.9). Chaque site, qui reçoit un message MAJ, enregistre les changements dans sa propre copie et renvoie un acquittement. Lorsque le site initiateur de la modification a reçu tous les acquittements, il libère le jeton.

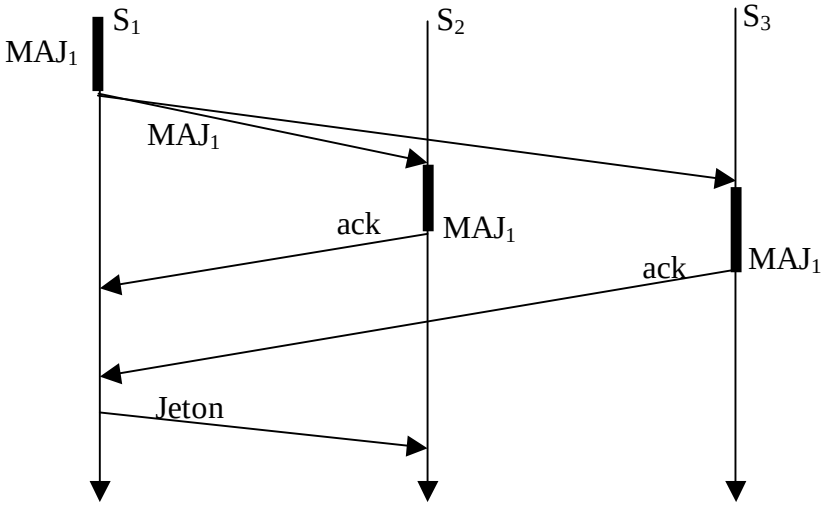



Figure 2.9 : Un mécanisme de mise à jour d'une base de données dupliquée

L'inconvénient de ce mécanisme est son manque de performances car l'initiateur doit attendre tous les acquittements et par conséquent le délai d'une mise à jour est au moins celui du serveur le plus lent. Un mécanisme alternatif consisterait à libérer le jeton sans attendre d'acquittement.

Examinons un scénario possible (figure 2.10). Le site $S_1$ effectue sa mise à jour, envoie le message de mise à jour aux sites $S_2$ et $S_3$ et expédie ensuite le jeton au site $S_2$. Quand ce dernier reçoit le message de mise à jour, il enregistre les changements. A la réception du jeton, $S_2$ effectue une autre mise à jour sur la copie locale qu'il maintient et diffuse aussi son message de mise à jour aux sites $S_1$ et $S_3$. Ce dernier site reçoit aussi le jeton. Supposons que la mise à jour de $S_2$ arrive à $S_3$ avant la première mise à jour. $S_3$ appliquera les changements de $S_2$ puis ceux de $S_1$, ceci conduisant à une incohérence de la base de données.

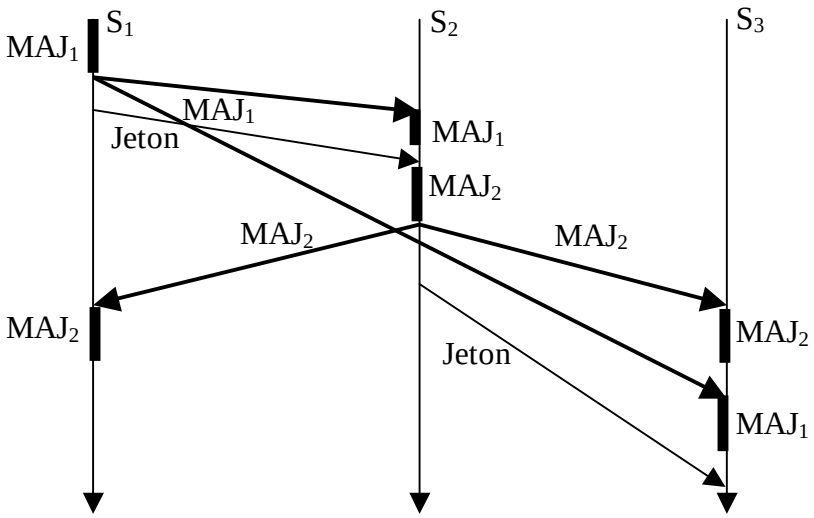



Figure 2.10 : Deux transactions conduisant à des copies incohérentes

Si on analyse cette situation, on s'aperçoit que trois messages sont à l'origine de ce problème : la mise à jour de S1 vers S2 qui précède l'envoi du jeton dont la réception précède l'envoi de la mise à jour de S1 vers S3 (figure 2.11). D'après ces relation de précédence, il semblerait raisonnable que le troisième message soit reçu après le premier. Or il n'en est rien.

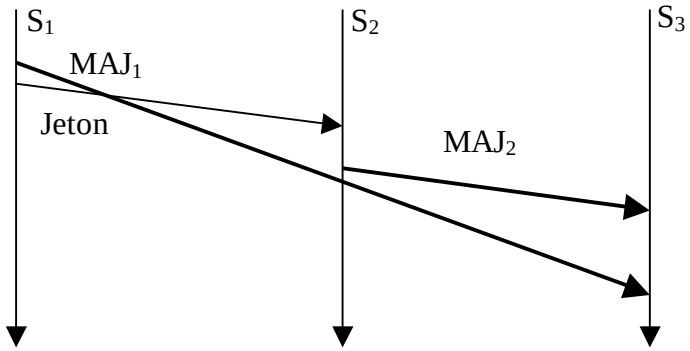



Figure 2.11 : Un comportement "pathologique" du réseau

### *4.2 L'ordre causal*

Nous allons maintenant formaliser les contraintes qui interdisent ces comportements pathologiques. Pour raisonner simplement sur les messages, nous définissons pour chaque message `m` quatre attributs :

- `m.emet` : identité du site émetteur,
- `m.dest` : identité du site destinataire,
- `m.hem` : heure "locale" de l'émission,
- `m.hdest` : heure "locale" de réception.

Ici aussi, aucune hypothèse n'est faite sur la synchronisation des horloges. D'ailleurs dans ce qui suit, seules des heures d'un même site sont comparées.

A titre d'exemple, montrons comment exprimer que les canaux d'un réseau sont fifo. Considérons deux message m et m' émis sur un même canal ; leur ordre de réception doit être le même que leur ordre d'émission. Ce qui s'exprime par :

$$\forall\ m,m'\ m.emet=m'.emet \text{ et } m.dest=m'.dest \text{ et } m.hem<m'.hem \Rightarrow m.hdest < m'.hdest$$

Nous voulons à présent introduire l'ordre causal entre deux messages [Lam78]. Cet ordre a pour signification intuitive qu'à l'émission du deuxième message, le site émetteur pourrait avoir connaissance du premier message et donc que ce premier message a pu avoir une influence sur le second message. Nous débutons par l'ordre causal immédiat qui caractérise de telles situations uniquement par l'historique d'exécution d'un site. Nous noterons "$<_i$" l'ordre causal immédiat.

**Définition** Soient m et m' deux messages; alors $m <_i m'$ si et seulement si :

1. $m.emet = m'.emet$ et $m.hem < m'.hem$

ou

2. $m.dest = m'.emet$ et $m.hdest < m'.hem$

Dans le cas 1, les deux messages sont émis sur le même site, le premier avant le second. Dans le cas 2, le premier message est reçu sur le site qui emettra ensuite le deuxième message. Du point de vue mathématique, $<_i$ n'est pas un ordre. En effet il ne vérifie pas la propriété de transitivité. Ceci nous conduit directement à l'ordre causal général.

**Définition** Soient m et m' deux messages; alors $m < m'$ si et seulement si :

- $\exists\ m_1,...,m_n$ tels que $m_1 = m$ et $m_n = m'$
- $\forall\ 0<k<n,\ m_k <_i m_{k+1}$

Autrement dit, l'ordre causal est la fermeture transitive de l'ordre causal immédiat et cette fois-ci on a bien affaire à un ordre au sens mathématique. Notons qu'il s'agit d'un ordre partiel car par exemple deux messages émis simultanément sur deux sites différents ne peuvent être en relation causale.

Nous sommes maintenant en mesure de caractériser un réseau fifo : dans un tel réseau, si un message en précède causalement un autre et qu'ils ont tous deux même destination, alors le premier message sera reçu avant le second.

**Définition** Un réseau est fifo si et seulement si :

$$\forall m,m'\ m < m' \text{ et } m.dest = m'.dest \Rightarrow m.hdest < m'.hdest$$

### *4.3 Emulation d'un réseau fifo*

Comme l'a illustré l'exemple introductif, un réseau asynchrone n'est pas nécessairement fifo (bien que tous ses canaux soient fifo). Pour simplifier la construction d'applications, le service que nous allons définir émule un réseau fifo au dessus d'un réseau asynchrone.

4.3.1 Une solution simple mais irréaliste

Une première solution consiste à envoyer avec un message, la liste des messages qui le précèdent causalement. L'ordre de la liste doit respecter l'ordre causal. Le service de chaque site maintient une liste des messages dont il a connaissance qu'il soit ou non émetteur ou récepteur de ces messages. Initialement cette liste est vide. Lorsque l'application désire envoyer un message, le service concatène ce nouveau message à la liste et envoie au service du destinataire la liste toute entière. A la réception d'une liste, le service concatène à sa liste tous les messages de la liste reçue absents de sa liste dans l'ordre de leur liste. Pour chaque nouveau message de sa liste, si celui-ci est à destination de son application, il le délivre à celle-ci.

Examinons l'application de cette solution au comportement pathologique précédent (figure 2.12). L'application du site $S_1$ désire envoyer un message $m_1$ au site $S_3$. Le service met à jour sa liste avec ce message et envoie la liste réduite à ce message. Puis, l'application du site $S_1$ désire envoyer un message $m_2$ au site $S_2$. Le service complète sa liste qui devient $(m_1, m_2)$ et l'envoie. Lorsque le service de $S_2$ reçoit cette liste, il complète sa liste (initialement vide) et examine les nouveaux messages : il n'est pas destinataire de $m_1$ mais est destinaire de $m_2$. Il le délivre donc à son application. Lorsque l'application de $S_2$ désire envoyer $m_3$ au site $S_3$ , son service complète sa liste qui devient $(m_1, m_2, m_3)$ et l'envoie au site $S_3$. A la réception de cette liste, le service de $S_3$ complète sa liste (initialement vide) et examine les nouveaux messages : il est destinataire de $m_1$ et le délivre, il n'est pas destinaire de $m_2$, il est destinataire de $m_3$ et le délivre. Enfin lors de la réception de la liste associée au message $m_1$, la liste ne contient aucun nouveau message.

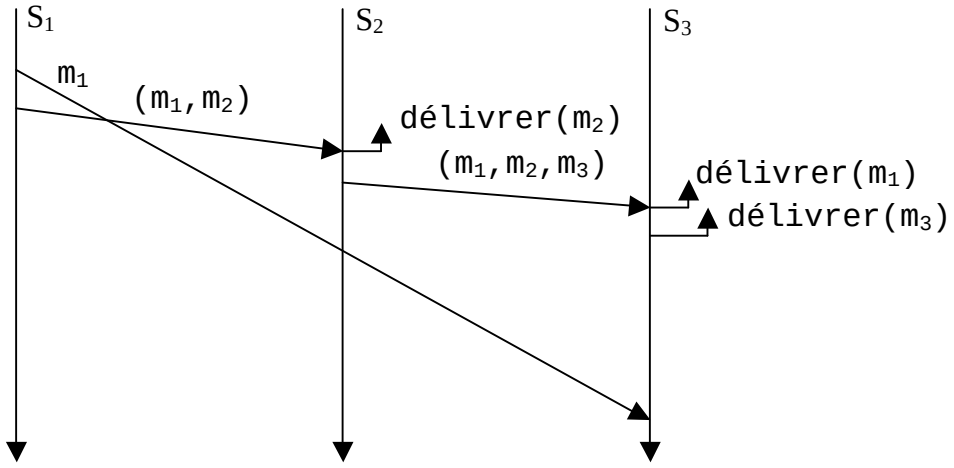



Figure 2.12 : Elimination du comportement pathologique

Le comportement du réseau (vue de l'application) est bien fifo car les messages qui précèdent causalement un message, le précèdent aussi dans toute liste où il apparaît. Par conséquent si l'un d'entre eux a même destination que ce message, il sera délivré avant lui. Cependant au fur et à mesure de l'exécution de l'application, la taille des listes de messages croît de manière considérable. Il nous faut donc adapter cette solution pour générer moins de trafic.

4.3.2 Une deuxième solution plus efficace

La première idée est de ne transporter qu'une seule fois chaque message et de remplacer la liste des messages qui le précèdent causalement par l'indication de leur existence. Ainsi quand un message arrive, si le service a l'information qu'un autre message à destination du même site le précède causalement et n'est pas encore reçu, alors celui-ci retarde la délivrance du message à l'application. Il reste à déterminer quelle abstraction de la liste choisir. La représentation retenue doit permettre de déterminer s'il faut retarder la délivrance d'un message. Compter le nombre de messages de la liste à destination d'un site n'est pas suffisant car l'égalité de deux compteurs ne signifierait pas nécessairement qu'il s'agit des mêmes ensembles de messages. Par contre, si on compte plus finement le nombre de messages de la liste émis par un site à destination d'un autre, alors l'abstraction conserve suffisamment d'informations.

Ainsi chaque site `i` maintiendra un tableau de compteurs à deux entrées appelé $\texttt{connaissance}_i$ tel que $\texttt{connaissance}_i\texttt{[j,k]}$ soit le nombre de messages émis par `j` à destination de `k` qui précèderaient causalement un message qui serait émis à cet instant par `i`. Chaque message envoyé est accompagné de la valeur du tableau à l'instant d'émission. A la réception d'un message `m`, le service du site `i` détermine s'il a reçu tous les messages qui précèdent causalement `m` en comparant élément par élément la colonne `i` de son tableau avec celle du tableau de `m`. Si c'est le cas, il le délivre et met à jour son tableau en prenant le maximum, élément par élément, des deux tableaux (ce qui est équivalent à une fusion de liste) et en prenant compte le message délivré.

**Interface du service**

Le service doit émuler un réseau. Il y aura donc une primitive descendante et une primitive montante :

- `émettre_vers(j,m)` : primitive du service, appelée par l'application pour envoyer un message sur le réseau émulé.
- `délivrer_de(j,m)` : primitive de l'application pour traiter les réceptions de message sur le réseau émulé, appelée par le service lorsqu'un message doit être délivré à l'application. Comme nous l'avons précisé au premier chapitre, ce type de primitive est utilisé par l'algorithme mais n'en fait pas partie.

**Variables du service**

- $\texttt{connaissance}_i\texttt{[1..N,1..N]}$ : tableau d'entiers à deux dimensions. Initialement, tout les éléments de ce tableau sont nuls. `N` est le nombre de sites.
- $\texttt{tampon}_i$ : tampon des message reçus non encore délivrés initialement vide. Chaque message est stocké avec l'identité de son émetteur et le tableau envoyé avec lui. Le tampon dispose de trois primitives de haut niveau :
  1. `insérer(élément)` qui ajoute un nouvel élément au tampon
  2. `tester(T,cond)` où `T` est un paramètre formel de tableau et `cond`, une condition portant sur `T` et sur $\texttt{connaissance}_i$ qui renvoie `Vrai` s'il existe un élément dont le tableau jouant le rôle de `T` vérifie la condition.
  3. `extraire(élément)` qui extrait l'élément sélectionné par la primitive précédente

Nous faisons le choix de faire transiter tous les messages par le tampon pour diminuer le temps d'exécution des réceptions de message et de confier la délivrance à un processus de service, appelé ici encore $\texttt{facteur}_i$.

**Algorithme**

L'émission d'un message applicatif est précédée de l'encapsulation du tableau $\texttt{connaissance}_i$ . Après l'envoi, celui-ci est mis à jour.

```
émettre_vers(j,m)
Début
     envoyer_à(j,(connaissance_i,m));
     connaissance_i[i,j]++;
Fin
```

La réception se limite à insérer le message dans le tampon du service.

```
sur_réception_de(j,(c,m))
Début
     tampon_i.insérer(<j,c,m>);
Fin
```

Le facteur, à l'instar des processus de service standard, exécute une boucle infinie où à chaque tour de boucle, il attend de trouver dans son tampon un message qui peut être délivré à l'application, le délivre et met à jour le tableau $\texttt{connaissance}_i$ en n'oubliant pas de compter le message qui vient d'être délivré.

```
Facteur_i
Début
     Tant que(Vrai)
          Attendre(tampon_i.tester(T,
                    ∀k,connaissance_i[k,i]≥T[k,i]);
          Tampon_i.extraire(<j,c,m>);
          connaissance_i=Max(connaissance_i,c);
          // maximum élément par élément
          connaissance_i[j,i]++;
          délivrer_de(j,m);
     Fin tant que
Fin
```

Remarquons que l'algorithme fonctionne correctement même si on ne fait plus l'hypothèse que les canaux du réseau asynchrone sont fifo.

Il reste cependant un problème à résoudre : comment dimensionner les compteurs sachant qu'ils ne cessent de croître? Une solution consiste à calculer, à partir d'une borne supérieure sur la durée d'exécution de l'application et d'une borne inférieure sur l'intervalle entre deux envois de messages, une borne supérieure sur le nombre de messages qui peuvent transiter dans un canal. Ce problème sera à nouveau abordé dans le cadre des horloges logiques au cours du chapitre sur le temps.

# 5 Exercices

***Sujet 1***
L'algorithme de rendez-vous réparti vu en cours garantit que si au moins un rendez-vous peut avoir lieu entre les différents demandeurs, alors l'un de ces rendez-vous aura lieu. Il ne garantit cependant aucune propriété d'équité dans le choix du rendez-vous.

**Question 1** Exhibez un scénario avec trois sites, où le site `3` attend indéfiniment un rendez-vous avec l'un quelconque des sites `1` et `2` alors que :

- une infinité de rendez-vous ont lieu entre les sites `1` et `2`,
- le site `3` apparaît une infinité de fois (mais pas toutes les fois) comme partenaire possible du site `1` et du site `2` dans leurs demandes successives de rendez-vous.

Pour remédier à ce problème, chaque site associe *un poids variable* aux autres sites initialisé avec l'identité du site concerné.
Lorsqu'un site `i` désire un rendez-vous, il s'adresse au partenaire potentiel `j` de moindre poids pour lequel il possède le jeton `(i,j)`.
Lorsqu'un site `i` obtient un rendez-vous avec le site `j` il réévalue le poids du site `j` comme étant le maximum des poids des autres sites `+1`.

**Question 2** Décrivez les variables de cette version de l'algorithme et leur initialisation.

**Question 3** Écrivez la nouvelle version de l'algorithme.

**Question 4** Reprenez le scénario de la question 1 et montrez qu'avec ce nouvel algorithme le site `3` finit par avoir un rendez-vous.

**Question 5** Quel inconvénient (déjà vu en cours) soulève cette adaptation ? Comment y remédier dans ce cas particulier ?

## 6 Références

[Bag89] R.L. Bagrodia "Synchronisation of asynchronous processes in CSP" ACM Toplas, vol 11,4 (Oct. 1989), pp. 585-597

[Boc79] G.V. Bochmann "Architecture of distributed computer" Springer-Verlag, LNCS 77 (1979)

[Lam78] L. Lamport "Time, clocks, and the ordering of events in a distributed system" Communications of ACM 21 (1978), 558-564

[MS80a] P.P. Merlin P.J. Schweitzer "Deadlock avoidance in store-and forward networks I : Store and Forward deadlock" IEEE Transactions on Communcations COM-28 (1980), 345-360

[TU81] S. Toueg J.D. Ullman "Deadlock-free packet switching networks" SIAM Journal of Computing 10,3 (1981), 594-611

# CHAPITRE III

# LE TEMPS

*version du 25 novembre 2003*

## 1 Introduction

Le temps est un concept fondamental des applications et des systèmes informatiques. De plus, dans un contexte réparti, la multiplicité des sites crée autant de référentiels possibles. Enfin, le temps revet des aspects complémentaires dont chacun d'entre eux nécessite un traitement particulier. Aussi dans cette courte introduction, nous décrivons brièvement les différentes notions associées au temps et leurs applications.

### *1.1 Le temps interne*

Lorsqu'un ordinateur exécute les instructions d'un ou plusieurs programmes, il s'appuie sur des durées fournies par son horloge sans qu'il y ait besoin que ce référentiel ait une cohérence avec un autre référentiel. Citons quelques exemples d'utilisation :
Le partage du temps entre les différents processus de la machine se fait par des techniques de quanta mesurés par cette horloge.

- L'interruption horloge, qui se déclenche avec une périodicité définie à partir de cette horloge.
- Le délai de suspension d'un processus avant sa réactivation (bien que le programmeur suppose qu'une seconde comptée par la machine n'est pas trop éloignée d'une seconde réelle).
- La refacturation du temps d'exécution (ici aussi la remarque précédente s'applique).

Cette notion du temps est généralement décrite dans les cours de système d'exploitation et ne fera pas ici l'objet d'une étude spécifique.

### *1.2 Le temps de l'environnement (temps réel)*

Dès que la machine doit interagir avec l'environnement dans lequel elle est plongée, on parle de *temps réel* pour désigner le rythme d'évolution de l'environnement. En règle générale, cette interaction se traduit par :

1. la prise en compte de signaux provenant de l'environnement,
2. l'émission de commandes permettant d'agir sur l'environnement.

Par exemple, un ordinateur de bord d'un avion de chasse pourrait à l'aide de capteurs détecter l'approche d'un missile, analyser la situation et ordonner à l'avion des actions telles que le changement de trajectoire ou l'envoi d'un missile anti-missile. Le point important est ici le fonctionnement de l'ordinateur en une suite de *cycles* "calcul-commandes-signaux". De plus, ces cycles sont *synchrones* au sens où le calcul doit être réalisé entre deux prises en compte consécutives de signaux. Un tel fonctionnement est schématisé sur la figure 3.1.

L'objet de la deuxième section sera l'étude du caractère synchrone d'un environnement réparti au sens où chaque station fonctionne en cycles et les cycles sont synchonisés entre les

différentes stations. Dans ce contexte, l'envoi de message joue le rôle de commande et la réception d'un message celui d'une prise en compte d'un signal.

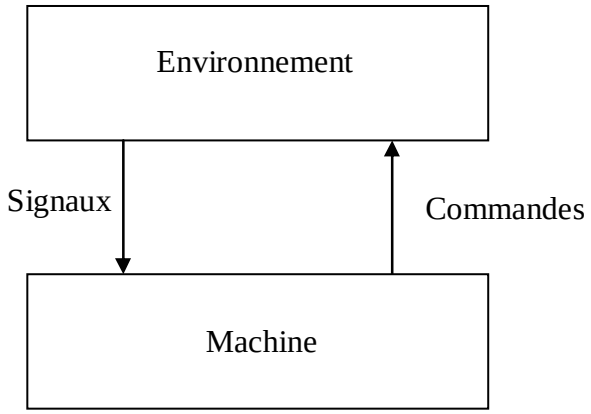



Figure 3.1 : Une application temps réel

### *1.3 Le temps universel*

Les deux notions précédentes du temps reposent essentiellement sur la durée. Le temps sert aussi à dater les événements (comparaison, tri, etc.). Ainsi un utilisateur a la possibilité de trier les fichiers d'un répertoire selon la date de dernière modification. La pratique usuelle est de se référer au temps "universel" (délivré par des organismes officiels [LPTF]). Dans un environnement réparti, chaque processeur est doté de sa propre horloge physique qui fournit une approximation du temps universel. Afin de coordonner une application répartie qui fait usage de ce temps, il est nécessaire de synchroniser ces horloges de façon à ce que l'écart entre deux horloges soit borné à tout instant par une valeur connue (petite de préférence). Cette borne dépendra bien sûr des incertitudes sur les délais de transfert des messages et des dérives des horloges par rapport au temps universel.

La troisième section est consacrée à la synchronisation des horloges physiques et à une application (la mise en place d'un rendez-vous temps réel).

### *1.4 Le temps logique*

Dans un algorithme séquentiel, si l'exécution d'une instruction précède temporellement l'exécution d'une autre instruction, le résultat de la première peut influencer le résultat de la seconde. Dans un algorithme réparti, la précédence temporelle n'est pas significative. En effet en raison des vitesses relatives des processeurs et des délais de transfert des messages, plusieurs exécutions d'un même algorithme sont possibles et des instructions peuvent se trouver inversées dans des exécutions différentes. Cependant quelque soit l'exécution une émission d'un message précédera toujours sa réception et deux instructions d'une même station seront toujours exécutées dans le même ordre. Autrement dit, seul l'ordre causal (vu au chapitre sur la communication) est significatif. De nombreux mécanismes basés sur cet ordre sont possibles. Nous nous limiterons ici à présenter l'un d'entre eux - les horloges logiques - que nous appliquerons lors du chapitre sur la concurrence.

# 2 Environnement synchrone

Nous allons d'abord définir un environnement synchrone pour une application répartie. Les caractéristiques d'un tel environnement reposent sur des contraintes en termes de comportement de réseau, de rapidité d'exécution et de structuration d'application.

**Hypothèse 1** L'application de chaque site travaille sous forme de *cycles numérotés*. Chaque cycle est initié par le battement d'une pulsation.

**Hypothèse 2** La primitive exécutée lors du battement de la pulsation est notée **`sur_pulsation(numéro)`**. Son unique paramètre correspond au numéro de la pulsation courante. L'exécution de ce code est non bloquant (pas d'appel à **`Attendre`**) et doit se terminer avant le battement de la pulsation suivante.

**Hypothèse 3** Durant l'exécution de **`sur_pulsation`**, un site émet au plus un seul message vers chaque autre site. Puisque le code n'est pas bloquant, il ne s'agit pas réellement d'une restriction. Le programmeur peut modifier son code pour concaténer les différents messages et les envoyer à la fin de la primitive.

**Hypothèse 4** Un message émis lors de l'appel à **`sur_pulsation(p)`** est reçu et traité par le site récepteur après l'exécution de **`sur_pulsation(p)`** et avant le battement de la pulsation `p+1`.

**Hypothèse 5** Il n'y a pas d'émission de message dans les primitives **`sur_réception_de`**.

Pour résumer, l'application travaille de manière synchrone. Au début d'un cycle, tous les sites exécutent la primitive **`sur_pulsation`** conduisant à des émissions de message. Ces messages sont ensuite reçus et traités par les différents sites. Puis une nouvelle pulsation est battue. Notons que du moment que l'application se comporte globalement comme indiqué, il est inutile que la même pulsation soit simultanément battue sur deux sites différents.

## *2.1 Comparaison entre environnement asynchrone et synchrone*

### 2.1.1 Construction d'un arbre de plus courts chemins

Afin d'illustrer la facilité de programmation que fournit un environnement synchrone ainsi que les gains en complexité, nous étudions la construction d'un arbre de plus courts chemins sur un graphe de communication. **Dans ce qui suit, le graphe de communication est quelconque**.

Cette construction est initiée par un site qui deviendra la racine de l'arbre. De plus, le chemin qui conduit de l'initiateur à un site quelconque doit être l'un des plus courts chemins possibles parmi ceux qui les relient dans le graphe de communication. Généralement la construction de l'arbre n'est que la première phase d'une transaction répartie dont la deuxième phase consiste à distribuer la transaction sur les sites à travers la structure de contrôle qu'est l'arbre. **Une contrainte forte est la nécessité de la fin de la construction avant de débuter la transaction**.

2.1.2 Algorithme en environnement asynchrone

Le principe de l'algorithme consiste, lorsqu'un site est rattaché à l'arbre (i.e. que l'on possède un père), à proposer à ses autres voisins de devenir ses fils. Puisqu'on recherche un arbre de plus courts chemins on adjoint à la proposition, la nouvelle distance à la racine qu'obtiendra le noeud sollicité.

Dans un environnement asynchrone, la première proposition de rattachement n'est pas nécessairement la meilleure. Aussi à chaque rattachement de meilleure qualité, on réitère sa proposition aux voisins.

On peut démontrer par récurrence sur la distance minimale à la racine que chaque noeud recevra une proposition correspondant à cette distance et donc que lorsque l'arbre se stabilisera, celui-ci sera un arbre de plus courts chemins.

Variables du site `i`

- $\texttt{voisins}_i$ : sous-ensemble des sites voisins de `i`. Cette constante définit le graphe de communication.
- $\texttt{père}_i$ : identité du site "père" dans l'arbre. Pour des facilités de programmation, elle est initialisée pour l'initiateur à sa propre identité. Elle n'est pas initialisée pour les autres sites.
- $\texttt{distance}_i$ : variable contenant la longueur du chemin allant du site jusqu'à la racine. Elle est initialisée à 0 pour l'initiateur et à l'infini ($+\infty$) ou à une constante supérieure ou égale au nombre de sites pour les autres sites.

Algorithme du site `i`

L'initiateur lance l'exécution par appel à la primitive **`Construire`**. Cette primitive propose à tous ses voisins (à l'exception de son père) de se rattacher à l'arbre par un message de construction `cons`. La distance à la racine qui en découlerait est fournie.

```
construire()
Début
     Pour tout j ∈ voisinsi\{pèrei}
          envoyer_à(j,(cons,distancei+1));
     Fin pour
Fin
```

A la réception d'un message de construction, le site examine si la distance proposée est meilleure que la distance courante. Si c'est le cas, il accepte la proposition et appelle à son tour `construire` pour proposer à ses voisins de devenir leur père car il s'est rapproché de la racine.

```
sur_réception_de(j,(cons,distance))
Début
     Si distancei > distance Alors
          pèrei = j;
          distancei = distance;
          construire() ;
     Fsi
Fin
```

Un des inconvénients de cet algorithme est qu'aucun des sites ne sait quand la construction est terminée. Or cette connaissance est indispensable pour passer à la phase transactionnelle. Il s'agit là du problème de la *terminaison*. Nous pourrions transformer de manière ad hoc l'algorithme mais nous élaborerons une solution générique dans le chapitre sur la cohérence.

Complexité de l'algorithme

Nous nous restreignons à mesurer la complexité calculée en nombre de messages échangés dans le pire des cas. `n` représente le nombre de sites.

Nous commençons par borner supérieurement le nombre de messages échangés :

- Le site initiateur appelle `construire` exactement une fois d'où un envoi d'au plus `n-1` messages.
- A chaque appel à `construire`, un site envoie au plus `n-2` messages (puisque son père est exclu de l'envoi).
- A chaque fois qu'un site appelle `construire` sa distance décroît. La valeur maximale (différente de l'infini) que celle-ci peut prendre est `n-1`. Donc un site peut changer au plus `n-1` fois de valeurs. Par conséquent, il appellera au plus `n-1` fois `construire`.

Posons `nbmess` le nombre de messages échangés. D'après l'évaluation précédente :

$$\texttt{nbmess} \leq \texttt{(n-1)+(n-1).(n-1).(n-2)} = \Theta(n^3)$$

Il nous reste à vérifier qu'il existe une exécution dont le nombre de messages échangés est de l'ordre de $n^3$. Notre scénario est basé sur un graphe de communication totalement maillé (autrement dit une clique). Dans une clique, le résultat de l'algorithme est nécessairement un arbre de hauteur 1 (figure 3.2).

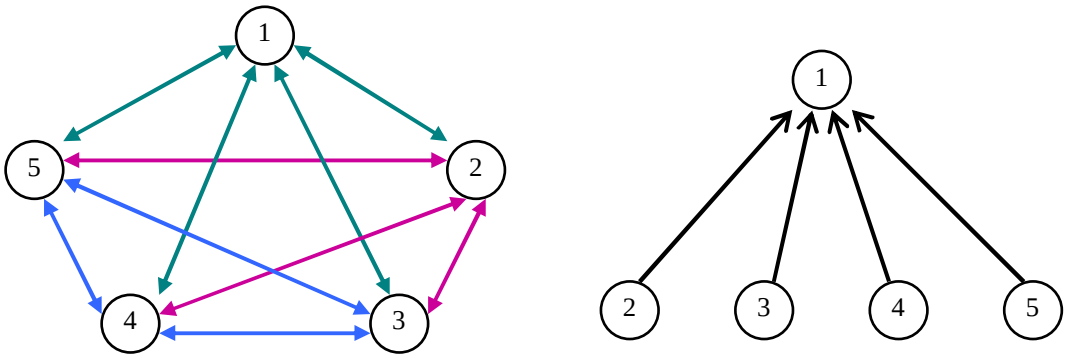



Figure 3.2 : Arbre de plus courts chemins d'une clique

Le site `1` est l'initiateur et envoie `n-1` messages de construction à ses voisins.

- Nous supposons qu'à l'exception du message au site `2`, tous les autres messages sont retardés. A la réception, le site `2` prend comme père le site `1` et envoie `n-2` messages à ses voisins. On suppose de même que tous les messages sont retardés excepté celui envoyé au site `3`. En itérant ce procédé, on construit un arbre (non stabilisé) qui n'est autre qu'un chemin qui parcourt les sites de `1` à `n`.
- Intéressons-nous au site `i`. Il est actuellement à une distance `i-1` de la racine. Supposons qu'il recoive successivement les messages de construction de `i-2,i-3,...,1`. Sa

distance diminuera par pas de 1 jusqu'à la valeur finale 1. Supposons un tel scénario pour chacun des sites i. L'arbre sera stabilisé.

Posons nbmess le nombre de messages échangés de ce scénario. L'initiateur envoie exactement n-1 messages. Un site i différent de l'initiateur appelle exactement i-1 fois construire() provoquant l'envoi de (i-1).(n-2) messages. D'où :
nbmess = (n-1)+2.(n-2)+3.(n-2)+...+(n-2).(n-2)+(n-1).(n-2)
=(n-1)+(n-2).(1+2+3+...+(n-1))=(n-1)+(n-2).$^1/_2$.n.(n-1)
= $\Theta(n^3)$
Ce qui achève le calcul de complexité.

2.1.3 Algorithme en environnement synchrone

Dans ce cas précis, il suffit de reprendre l'algorithme précédent en remarquant que, puisque l'environnement est synchrone, la première proposition de rattachement est nécessairement la meilleure. La seule difficulté consiste à détecter au début d'une pulsation qu'un site vient d'être rattaché à l'arbre. On remarque le site racine est rattaché à la pulsation 0, les sites à distance 1 le sont à la pulsation 1, ..., les sites à distance d le sont à la pulsation d. Il suffira alors de comparer la distance du site à la valeur de la pulsation pour savoir quand proposer à ses voisins le rattachement à l'arbre.

Variables du site i

- $voisins_i$ : sous-ensemble des sites voisins de i. Cette constante définit le graphe de communication.
- $père_i$ : identité du site "père" dans l'arbre. Pour des facilités de programmation, elle est initialisée pour l'initiateur à sa propre identité. Elle n'est pas initialisée pour les autres sites.
- $distance_i$ : variable contenant la longueur du chemin allant du site jusqu'à la racine. Elle est initialisée à 0 pour l'initiateur et à l'infini pour les autres sites.
- $pulsation_i$ : indice de la pulsation courante initialisée à 0.

Algorithme du site i

```
sur_ pulsation(pulsation_i)
Début
     Si pulsation_i == distance_i Alors
          Pour tout j ∈ voisins_i\{père_i}
               envoyer_à (j,(cons,distance_i+1));
          Finpour
     Fsi
Fin

sur_réception_de(j,(cons,distance))
Début
     Si distance_i == +∞ Alors
          père_i = j;
          distance_i = distance;
     Fsi
Fin
```

Complexité de l'algorithme

Le calcul du nombre de messages échangés est ici très simple. Un site n'envoie qu'une proposition de construction à tous ses voisins excepté son père. Donc le pire des cas est atteint sur une clique où l'initiateur envoie $n-1$ messages et chaque autre site envoie $n-2$ messages. Ce qui nous donne $(n-1)+(n-1).(n-2)=(n-1)^2$ messages. Nous obtenons donc un gain d'un ordre de grandeur par rapport à l'algorithme asynchrone précédent. Notons qu'il existe des algorithmes asynchrones plus sophistiqués qui pour une complexité de $\Theta(n^3)$ messages calculent un arbre pour chaque noeud [Tou80].

## *2.2 Emulation d'un environnement synchrone*

Dans le paragraphe précédent, nous avons mis en évidence l'intérêt du développement d'une application en environnement synchrone. Comme les environnements asynchrones sont encore les plus répandus, nous allons montrer dans ce paragraphe comment émuler un environnement synchrone au dessus d'un réseau synchrone.

### 2.2.1 Difficultés de mise en oeuvre

Le premier problème à résoudre pour le service est la détermination de l'instant où celui-ci pourra battre la pulsation suivante. Plus précisément avant de battre la pulsation, deux conditions doivent être remplies :

1. Le code associé à la pulsation doit avoir été exécuté. Dans ce cas, il suffit de fournir à l'application une primitive de l'interface qui lui permet d'indiquer que le traitement de la pulsation courante est terminé. On la notera **`fin_traitement()`**.
2. Tous les messages envoyés par les autres sites à destination de ce site doivent être reçus et traités. Comme le réseau est asynchrone, il est impossible à un site `i` lorsqu'il n'a pas reçu de message de `j` de déterminer (quelque soit le délai qu'il se donne) si `j` n'a pas envoyé de message ou si son message est encore en transit.

Pour résoudre ce problème, les services doivent se coordonner comme suit. Chaque service observe les messages émis par son application lors du traitement de la pulsation. A la fin du traitement, il détermine les sites qui ne recevront pas de message de son application et envoie à leur service des messages de contrôle. Ainsi chaque site est assuré de recevoir d'un autre site soit un message d'application, soit un message de service. La deuxième condition est alors remplie quand tous ces messages sont reçus. La figure 3.3 illustre ce mécanisme : les messages de service y sont représentés par $\Rightarrow$ et les messages d'application par $\rightarrow$.

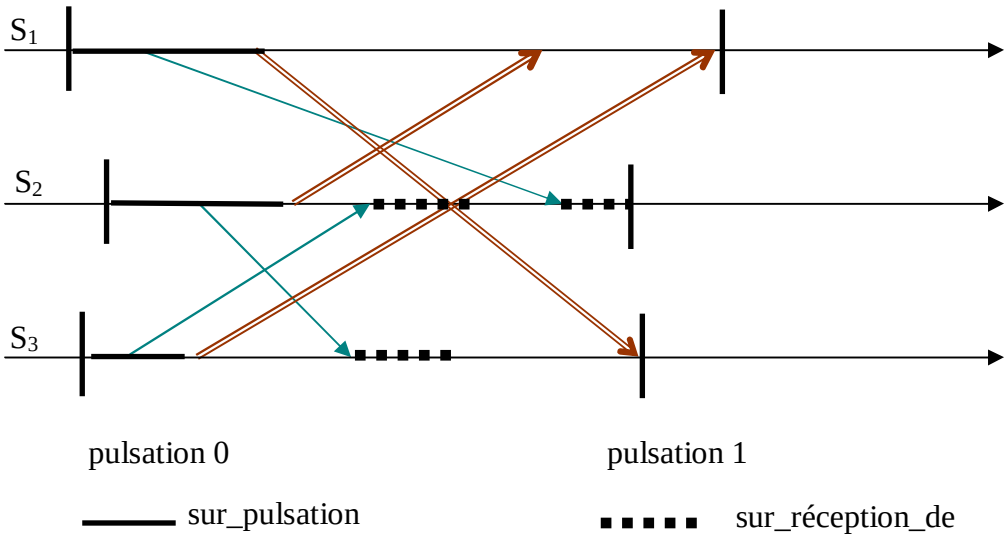



Figure 3.3 : Quand battre la pulsation?

Le deuxième problème à résoudre pour le service est de conserver des messages qui pourraient arriver trop tôt. La figure 3.4 illustre cette situation. Deux sites participent à l'application. Le site `1` a un traitement de la première pulsation particulièrement long durant lequel il envoie un message au site `2`. Celui-ci peut donc débuter le traitement de la deuxième pulsation. Au cours de chacune des deux premières pulsations, le site `2` envoie un message au site `1`. Aucun de ces messages ne peut être délivré immédiatement : le premier message doit attendre la fin du traitement de la pulsation courante tandis que le deuxième message doit attendre la fin du traitement de la prochaine pulsation.

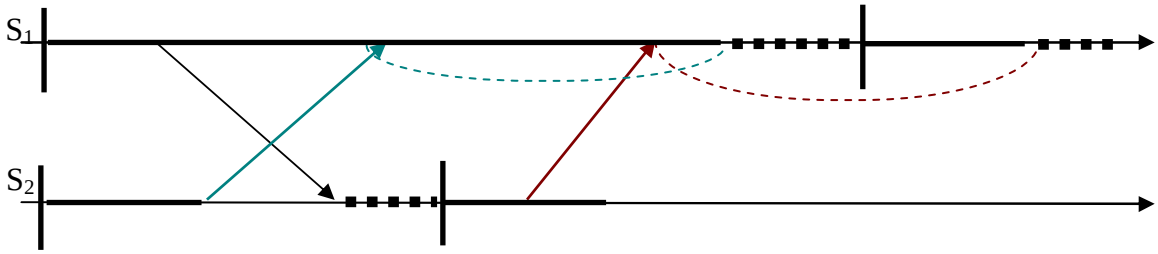



Figure 3.4 : Combien de messages conserver ?

Etant donnés deux sites `i` et `j`, quel est le nombre maximum de messages que doit conserver le service de `i` provenant de `j` ? Répondre à cette question nous permettra de dimensionner les tampons de messages. Nous allons démontrer que la situation décrite par l'exemple précédent est le pire des cas, autrement dit que ce nombre est `2`. Notons :

- $pulse_i(n)$ : l'instant de battement de la pulsation `n` sur le site `i`.
- $emet_{ij}(n)$ : l'instant d'émission du message de `i` vers `j` durant la pulsation `n`.
- $rec_{ij}(n)$ : l'instant de réception du message, émis de `i` vers `j` durant la pulsation `n`.

Nous établissons une série d'inégalités :

- $emet_{ji}(n+2) < rec_{ji}(n+2)$ car l'émission d'un message précède sa réception
- $pulse_j(n+2) < emet_{ji}(n+2)$ par définition des instants
- $rec_{ij}(n+1) < pulse_j(n+2)$ par la synchronisation entre les sites

- $emet_{ij}(n+1) < rec_{ij}(n+1)$ car l'émission d'un message précède sa réception
- $pulse_i(n+1) < emet_{ij}(n+1)$ par définition des instants

Par transitivité, on obtient : $pulse_i(n+1) < rec_{ji}(n+2)$

D'autre part, en raison des condtions sur le battement de la pulsation, $rec_{ji}(n-1) < pulse_i(n)$.

Autrement dit dans l'intervalle $[pulse_i(n), pulse_i(n+1)]$ seuls peuvent être reçus les messages émis au cours de la pulsation `n` et `n+1`. Ce raisonnement est illustré par la figure 3.5.

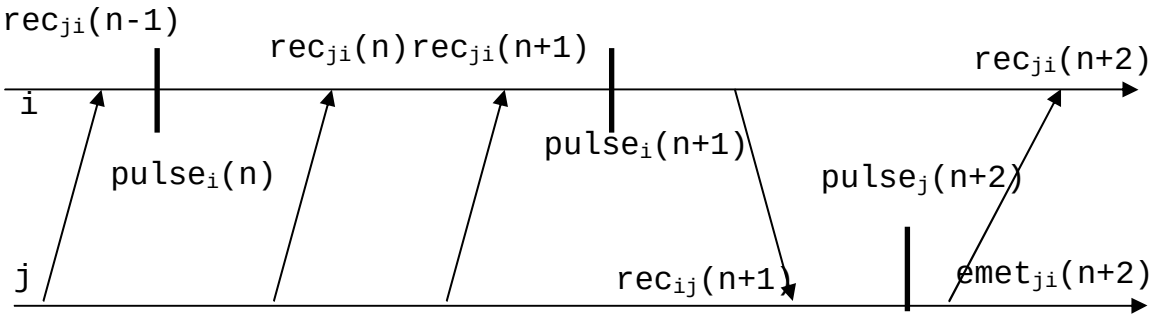



Figure 3.5 : Au plus deux messages à conserver par pulsation

2.2.2 Réalisation de l'émulation

Variables du site `i`

- $pulsation_i$ : indice de la pulsation courante. Elle est initialisée à 0.
- $traitement\_en\_cours_i$ : booléen qui indique si le site exécute le code associé à la pulsation, initialisé à `Vrai`.
- $nbmess\_reçus_i$ : nombre de messages reçus par `i` pour la pulsation courante, initialisé à 0.
- $nbmess\_avance_i$ : nombre de messages reçus par `i` pour la pulsation suivante, initialisé à 0.
- $à\_envoyer_i[1..N]$ : tableau de booléens initialisés à `Vrai`. $à\_envoyer_i[j]$ est `Vrai` si le service doit envoyer un message de service à j à la fin du traitement de la pulsation. `N` est le nombre de sites de l'application.
- $courant_i[1..N]$ : tampon de stockage des messages reçus pour la pulsation courante en attente d'être délivrés à l'application. $courant_i[j]$ contient éventuellement le message provenant de `j`. Toute cellule de ce tableau est composée des deux champs `<présent,contenu>`. `présent` est un booléen initialisé à `Faux` indiquant si un message est contenu dans la cellule. `contenu` représente les données du message.
- $avance_i[1..N]$ : tampon de stockage des messages reçus en avance. La structure et l'initialisation de ce tableau sont identiques à celles du tableau précédent.

Interface application/service

Cette interface est schématisée en figure 3.6

Les primitives de services appelées par l'application sont :

- `émettre_vers(j,m)` envoie un message vers `j` dans l'environnement émulé

- `fin_traitement()` indique au service que le traitement `sur_pulsation` est terminé

Les primitives d'application appelées par le service sont :
- `délivrer_de(j,m)` délivre un message provenant de j à l'application
- `battre(pulsation`$_i$`)` relance le processus applicatif pour traitement de la pulsation courante

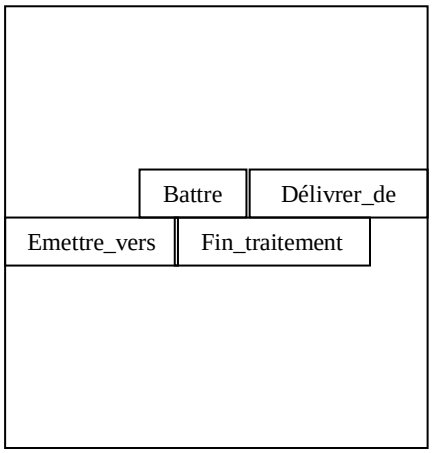



Figure 3.6 : Interface application / service

Algorithme du site `i`

Lorsqu'un le service envoie un message d'application à `j`, il n'a plus à envoyer un message de service vers ce site.

```
émettre_vers(j,m)
Début
     envoyer_à(j,(app,pulsation_i,m));
     à_envoyer_i[j]=faux ;
Fin
```

Nous introduisons une procédure interne `test_and_set_pulsation` appelée par les procédures `fin_traitement` et `sur_réception_de` pour vérifier les conditions de battement d'une pulsation et réaliser les traitements qui lui sont associés. En effet, le passage d'un cycle au suivant peut intervenir dans deux cas différents :
- L'application a fini son traitement et le service reçoit le dernier message de la pulsation courante.
- Le service a reçu tous les messages de la pulsation courante et l'application finit son traitement.

Cette procédure a trois tâches :
- Réinitialiser et mettre à jour les variables pour le début de cycle.
- Transférer les messages du tampon `avance`$_i$ au tampon `courant`$_i$.
- Battre la pulsation.

```
test_and_set_pulsation_i()
Début
     Si(traitement_en_cours_i==Faux)ET(nbmess_reçus_i==n-1) Alors
          pulsation_i++;
          traitement_en_cours_i = Vrai;
          nbmess_reçus_i  = nbmess_avance_i;
          nbmess_avance_i  = 0;
          Pour j de 1 à n Faire
               Si avance_i[j].present = Vrai Alors
                    courant_i[j] = avance_i|j];
                    avance_i[j].present = Faux ;
               Fsi
          Fin pour
          battre(pulsation_i);
     Fin si
Fin
```

Lorsqu'un message d'application `m` est reçu par le service, trois cas sont possibles :

- `m` arrive en avance (étiqueté par la pulsation suivante) et il est alors stocké dans le tampon $\texttt{avance}_i$.
- `m` arrive dans la bonne pulsation et `i` n'a pas fini son traitement, il est alors stocké dans le tampon $\texttt{courant}_i$.
- `m` arrive dans la bonne pulsation et `i` a fini son traitement, il est alors délivré à l'application.

Dans le cas d'un message de service, le premier cas conduit à incrémenter $\texttt{nbmess\_avance}_i$ tandis que la distinction entre le second et le troisième cas n'a pas lieu d'être puisque le seul but d'un message de service est d'incrémenter $\texttt{nbmess\_reçus}_i$. Comme la réception d'un message de la pulsation courante peut entrainer la fin de la pulsation, cette primitive appelle `test_and_set_pulsation`.

```
sur_réception_de(j,(tp,pulse,cont))
Début
     Si pulse > pulsation_i Alors
          nbmess_avance_i++;
          Si tp=app Alors
               avance_i[j].present = Vrai;
               avance_i[j].contenu = cont;
          Finsi
     Sinon
          nbmess_reçus_i++;
          Si tp=app Alors
               Si traitement_en_cours_i = faux Alors
                    delivrer_de(j,cont);
               Sinon
                    courant_i[j].present = Vrai;
                    courant_i[j].contenu = cont;
               Fsi
          Fsi
          test_and_set_pulsation();
     Fsi
Fin
```

A l'appel de la primitive `fin_traitement`, le service délivre les messages stockés dans le tampon $courant_i$. Il envoie aussi les messages de service aux sites pour lesquels l'application n'a pas envoyé de message. Comme il peut s'agir de la fin de la pulsation, il appelle `test_and_set_pulsation`.

```
fin_traitement()
Début
     Pour j de 1 à n faire
          Si (à_envoyer_i[j] == Vrai)  ET  (i != j) Alors
               envoyer_à(j,(serv,pulsation_i,∅));
          Sinon
               à_envoyer_i[j] = Vrai;
          Fsi
          Si courant_i[j].present == Vrai Alors
               délivrer_de(j,courant_i[j].contenu);
               courant_i[j].present = Faux;
          Fsi
     Fin pour
     traitement_en_cours_i = Faux;
     test_and_set_pulsation();
Fin
```

D'autres émulations de réseau synchrone de plus grande efficacité sont développées dans [Awe85]

# 3 Le temps physique

## *3.1 Synchronisation d'horloges*

### 3.1.1 Hypothèses de fonctionnement

Plaçons nous dans le cadre d'un système réparti où chaque site $i$ dispose d'une horloge physique $h_i$. Cette horloge est un dispositif permettant à chaque instant du temps $t$ de fournir une date $h_i(t)$ de préférence la plus proche possible de $t$. Comme nous l'avons déjà indiqué dans l'introduction, nous supposons l'existence d'un référentiel universel du temps qui n'est pas accessible aux sites.

La valeur de l'horloge (si elle n'est réajustée) croît linéairement en fonction du temps et nous notons $C_i$ son coefficient de croissance. Autrement dit entre deux instants $t_1$ et $t_2$, on a :

$$h_i(t_2)-h_i(t_1) = C_i.(t_2-t_1)$$

Le fournisseur des horloges garantit que la dérive des horloges vis à vis du temps universel reste borné en valeur relative par $\rho$. Autrement dit :

$$1-\rho \leq C_i \leq 1+\rho$$

La valeur de $\rho$ est donnée par le constructeur dans la spécification du matériel. Généralement $\rho$ est de l'ordre de $10^{-5}$ ou de $10^{-6}$. Contrairement aux hypothèses du premier chapitre, nous supposons que le délai de transit d'un message est borné inférieurement par $\delta_{min}$ et supérieurement par $\delta_{max}$. En effet si une borne inférieure existe toujours (0!), l'existence d'une borne supérieure implique le réseau n'est plus asynchrone. Dans la pratique, les délais de transit des paquets IP sur un réseau local (en incluant la traversée des couches réseaux) sont de l'ordre de $10^{-4}$ s. On pourra le vérifier avec la commande ping.

### 3.1.2 Objectif de la synchonisation

Remarquons que même si les horloges sont synchronisées "initialement" (i.e. $h_i(0)=0$) sans ajustement ultérieur leur dérive est non bornée :

$$\lim_{t\to\infty} | h_i(t)-h_j(t)| = \lim_{t\to\infty} |C_i-C_j|.t = \infty$$

Notre objectif consiste à ajuster les horloges de façon à obtenir une borne $B$ calculable à partir des paramètres de l'environnement et de l'algorithme telle que :

$$\forall\ t,\ \forall\ i,j\ |\ h_i(t)-h_j(t)|\leq B$$

On trouvera dans [Ram90] un panorama des mécanismes de synchronisation d'horloges.

### 3.1.3 Principe de la synchonisation

Afin de réaliser cet ajustement, chaque site diffuse la valeur de son horloge toutes les $per$ unités de temps (où les unités de temps sont comptées avec l'horloge).

D'autre part, il faut choisir sur "quelle horloge se recaler". Plusieurs possibilités s'offrent à nous :

- se caler sur un site fixe mais cela pose le problème de la panne de ce site. Nous rejetons donc cette solution.

- se caler sur la plus lente (de manière implicite) mais dans ce cas, les horloges sont amenées à revenir en arrière ce qui est inacceptable (il suffit de penser à l'attribut date de dernière modification d'un fichier).
- se caler sur la plus rapide (de manière implicite) ce qui crée des discontinuités dans le temps (cf la figure 3.7). Ces discontinuités sont tout à fait tolérables ; il suffit d'imaginer que la machine était éteinte durant l'intervalle concerné.

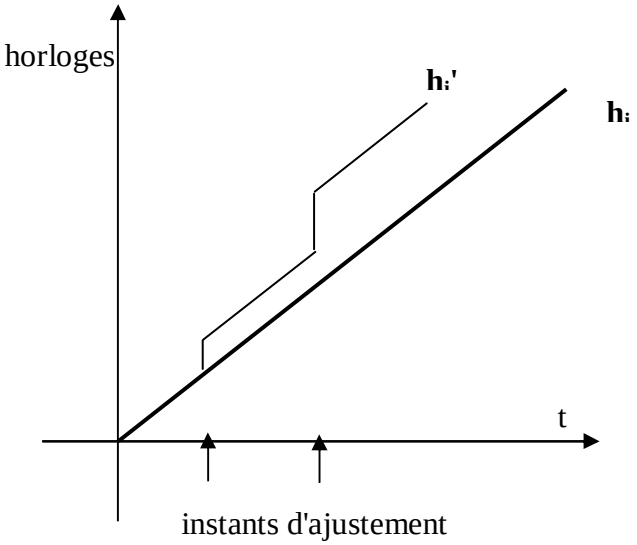



Figure 3.7 : Effet de l'ajustement ($h_i'$ horloge ajustée)

Il nous faut maintenant préciser le calcul de l'ajustement. Définissons les paramètres suivants (voir la figure 3.8) :

- $t_0$ instant d'envoi de la valeur $h_j(t_0)$ par le site $j$
- $t_1$ instant de réception de la valeur $h_j(t_0)$ par le site $i$
- $h_i^-(t_1)$ valeur de l'horloge du site $i$ juste avant l'ajustement
- $h_i^+(t_1)$ valeur de l'horloge du site $i$ juste après l'ajustement

Nous voulons nous recaler en nous rapprochant le plus possible de $\max(h_i^-(t_1), h_j(t_1))$ d'où $h_i^+(t_1) \geq h_i^-(t_1)$. D'autre part, $i$ ne connait pas la valeur de $h_j(t_1)$ mais le calcul d'un minorant est possible :

- $t_1 - t_0 \geq \delta_{min}$ (temps de transit d'un message)
- d'où $h_j(t_1) - h_j(t_0) \geq (1-\rho).\delta_{min}$ (qualité de l'horloge)
- autrement dit, $h_j(t_1) \geq h_j(t_0) + (1-\rho).\delta_{min}$

Ce qui nous donne comme règle d'ajustement :

$$h_i^+(t_1) = Max(h_i^-(t_1), h_j(t_0)+(1-\rho).\delta_{min})$$

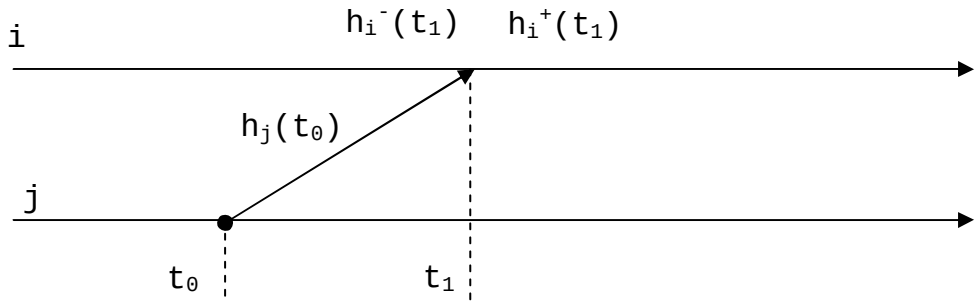



Figure 3.8 : Ajustement d'une horloge

3.1.4 Calcul de la borne

Soit dans un réseau, `i` la station la plus lente et `j` la station la plus rapide. Il est clair que si l'on éteint toutes les stations sauf `i` et `j`, alors la dérive des horloges sera plus grande car la diffusion des horloges par les stations intermédiaires ne peut que diminuer la dérive entre `i` et `j`. Autrement dit, la pire des situations est celle d'un réseau composé de deux stations. Nous nous limiterons donc à cette situation.

Afin de déterminer la borne $B$, nous décomposons l'écart d'horloge entre les stations `i` et `j` en deux quantités : d'une part l'écart l'obtenu après le dernier ajustement et d'autre part la dérive des horloges depuis cet ajustement (voir la figure 3.9). Nous recherchons donc une borne $B_1$ de la première quantité et une borne $B_2$ de la deuxième quantité ($B=B_1+B_2$).

Si nous reprenons les notations associées à l'ajustement, nous remarquons que :

- $t_1 - t_0 \leq \delta_{max}$ (temps de transit d'un message)
- d'où $h_j(t_1) - h_j(t_0) \leq (1+\rho).\delta_{max}$ (qualité de l'horloge)
- d'où, $h_j(t_1) \leq h_j(t_0) + (1+\rho).\delta_{max}$
- comme d'autre part, $h_i^+(t_1) \geq h_j(t_0) + (1-\rho).\delta_{min}$

On obtient $B_1 = (1+\rho).\delta_{max} - (1-\rho).\delta_{min} \geq h_j(t_1) - h_i^+(t_1)$

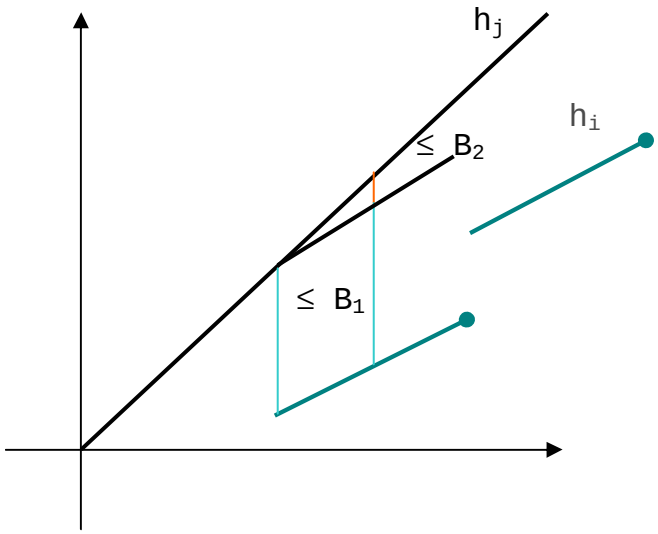



Figure 3.9 : Ecart entre deux horloges

Nous entreprenons maintenant le calcul de $B_2$. En suivant les notations de la figure 3.10, nous avons :

$t_3-t_1=(t_3-t_2)+(t_2-t_0)+(t_1-t_0)$

Le site j diffuse la valeur de son horloge puis il attend per unités de temps avant d'envoyer le suivant. Cependant le temps écoulé entre les deux évènements est réellement :

$t_2-t_0=per/C_j \leq per/(1-\rho)$

Par conséquent :

$t_3-t_1=(t_3-t_2)+(t_2-t_0)+(t_1-t_0)\leq\delta_{max}+per/(1-\rho)-\delta_{min}$

Notons d la dérive des horloges entre les deux mise à jour aux instants $t_1$ et $t_3$.

$d=(C_j-C_i).(t_3-t_1)\leq 2.\rho.[\delta_{max}-\delta_{min}+per/(1-\rho)]=B_2$

D'où :

$B=B_1+B_2=(1+\rho).\delta_{max}-(1-\rho).\delta_{min} + 2.\rho.[\delta_{max}-\delta_{min}+per/(1-\rho)]$

En négligeant ρ devant 1 ($\rho << 1$) on obtient :

$B \approx \delta_{max} - \delta_{min} + 2.\rho .(\delta_{max} - \delta_{min} + per) = (1 + 2\ \rho)(\delta_{max} - \delta_{min}) + 2.\rho .per$

$B \approx \delta_{max} - \delta_{min} + 2.\rho\ per$

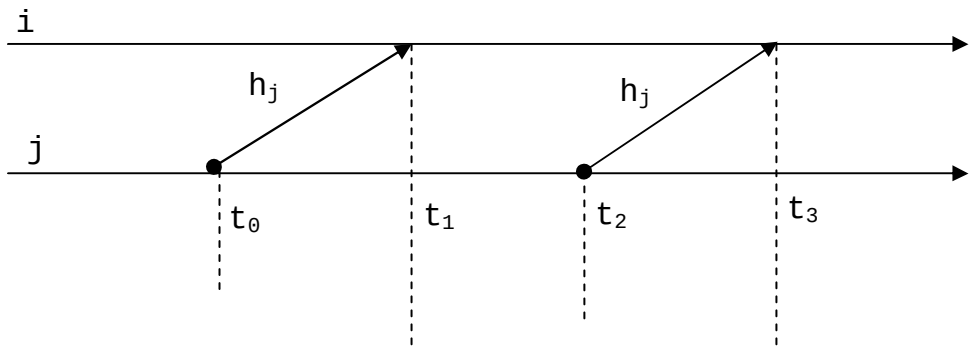



Figure 3.10 : Deux diffusions d'horloge

3.1.5 Choix de la périodicité de diffusion

Il reste à choisir la périodicité de la diffusion. Une périodicité plus courte augmente la synchronisation au prix d'une plus grande charge sur le réseau. Pour fixer les idées, considérons les valeurs suivantes : $\rho = 10^{-6}$s, $\delta_{max} - \delta_{min} = 10^{-4}$s. Calculons B pour différentes valeurs de per.

- Si per = 100s alors $B = 10^{-4} + 2.10^{-6}. 10^2 = 0,0003$s
- Si per = 10s alors $B = 10^{-4} + 2.10^{-5}. 10^1 = 0,00012$s
- Si per = 1s alors $B = 10^{-4} + 2.10^{-5} = 0,000102$s

Il s'avère que de passer de 10s à 1s multiplie par 10 le trafic pour un gain négligeable; une valeur très satisfaisante sera comprise entre 10s et 100s. A titre d'exemple, la valeur par défaut de timed (le daemon UNIX de synchronisation d'horloges) est de 30s.

### *3.2 Un exemple d'application : le rendez-vous temps réel*

Nous désirons modifier notre primitive de rendez-vous de telle sorte qu'elle incorpore des contraintes temps-réel. En effet, la primitive du chapitre précédent implique une attente inconditionnelle incompatible avec un environnement temps réel. Aussi nous voulons autoriser l'application à fournir une date d'échéance au delà de laquelle le rendez-vous est considéré comme ayant échoué et l'application reprend son exécution.

La nouvelle primitive $RV(E_i, dech_i)$ incorpore donc une date additionnelle relative à l'horloge locale. Nous supposons que les horloges sont synchronisées par le mécanisme de la section précédente.

Nous n'allons pas redéfinir entièrement l'algorithme mais nous focaliser uniquement sur le respect des contraintes temporelles entre deux sites 1 et 2 qui désirent un rendez-vous commun. Rappelons qu'il s'agit là d'un problème de prise de décision commune : le rendez-vous doit être accepté par les deux sites ou par aucun. De plus, les deux contraintes temporelles doivent être respectées.

Appelons $dappel_1$ et $dappel_2$ les instants mesurés par leur horloge respective, auxquels les sites 1 et 2 invoquent la primitive RV. Notons $dech_1$ et $dech_2$ les dates d'échéance des demandes de rendez-vous. Enfin notons $dchoix_1$ et $dchoix_2$ les instants mesurés par leur horloge respective, auxquels les sites 1 et 2 décident ou non d'accepter le rendez-vous. La condition exacte pour que le rendez-vous ait lieu est :

$$dchoix_1 \leq dech_1 \text{ et } dchoix_2 \leq dech_2$$

Cependant $dchoix_1$ est inaccessible au site 2 et vice versa. On est donc amené à remplacer pour chaque site la condition précédente par des conditions plus restrictives :

- **condition du site 1** $dchoix_1 \leq dech_1$ et $dappel_1+\delta_{max}+B \leq dech_2$
  - car soit l'appel au RV sur le site 2 est effectué avant la réception de la demande de rendez-vous du site 1 et $dappel_1+\delta_{max}+B$ est un majorant de $dchoix_2$
  - soit l'appel au RV sur le site 2 est effectué après la réception de la demande de rendez-vous du site 1 mais dans ce cas la décision sur le site 2 est prise au moment de l'appel et nécessairement $dchoix_2 \leq dech_2$
- **condition du site 2** $dappel_2+\delta_{max}+B \leq dech_1$ et $dchoix_2 \leq dech_2$ pour les mêmes raisons

Cependant ces tests ne sont pas satisfaisants car il peuvent induire une décision différente sur les deux sites. Aussi lors d'un appel à RV, le service de chaque site i envoie à un site candidat $dech_i$ mais également $dappel_i$. Nous obtenons alors une condition encore plus restrictive mais qui peut être testée simultanément sur les deux sites.

$$dappel_2+\delta_{max}+B \leq dech_1 \text{ et } dappel_1+\delta_{max}+B \leq dech_2$$

# 4 Les horloges logiques

## *4.1 Principe des horloges logiques*

Considérons le cas d'un serveur non réentrant (traitement d'une requête à la fois) qui met en attente les requêtes arrivées en cours de traitement. Une politique possible de choix de la prochaine requête à traiter pourrait être de les traiter dans l'ordre de réception. Il est clair qu'une telle politique favorise les sites les plus "proches" du serveur. A l'inverse, on pourrait tenter les traiter selon l'ordre d'émission. L'inconvénient est cette fois-ci l'écart éventuel entre les horloges physiques. Une troisième politique consisterait à traiter les requêtes de la façon suivante :

- si la requête $r_1$ précède causalement $r_2$, alors traiter d'abord $r_1$,
- sinon si la requête $r_2$ précède causalement $r_1$, alors traiter d'abord $r_2$,
- sinon (l'ordre causal est un ordre partiel) les traiter dans un ordre arbitraire.

Sur la figure 3.11, la requête $r_2$ arrive avant $r_1$ mais elle la suit causalement. Nous pourrions évidemment transformer le réseau en un réseau FIFO de telle sorte l'ordre de réception soit une extension de l'ordre causal mais cette solution présenterait deux inconvénients. Elle pourrait rendre le serveur inactif alors qu'une requête serait immédiatement disponible et d'autre part sa gestion impliquerait un tableau bidimensionnel indexé par les sites.

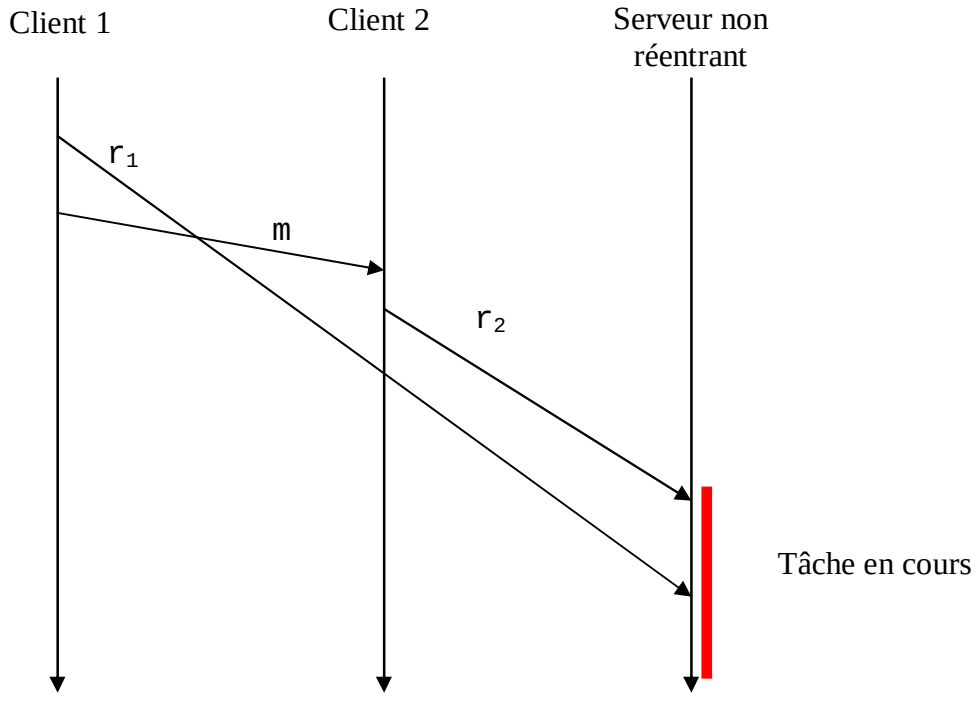



Figure 3.11 Un serveur non réentrant

Nous souhaitons donc définir un mécanisme qui permette d'adopter une telle politique sans contraindre le réseau à être FIFO. Le mécanisme que nous allons étudier est celui des horloges logiques [Lam78].
Plus précisément chaque site gère une horloge logique, c'est à dire un compteur croissant initialisé à `0`. Chaque message est estampillé avec la valeur courante de l'horloge logique et l'on désire que (dénotant $h_m$ l'estampille de `m`) :

$$m <_c m' \Rightarrow h_m < h_{m'}$$

La réciproque n'est pas nécessairement vérifiée puisque l'ordre causal est partiel et l'ordre sur les entiers est total. Au vu de la transitivité de l'ordre, il suffit de garantir la propriété précédente pour l'ordre immédiat. Ceci nous conduit directement aux deux règles de mise à jour de l'horloge logique illustrées par la figure 3.12

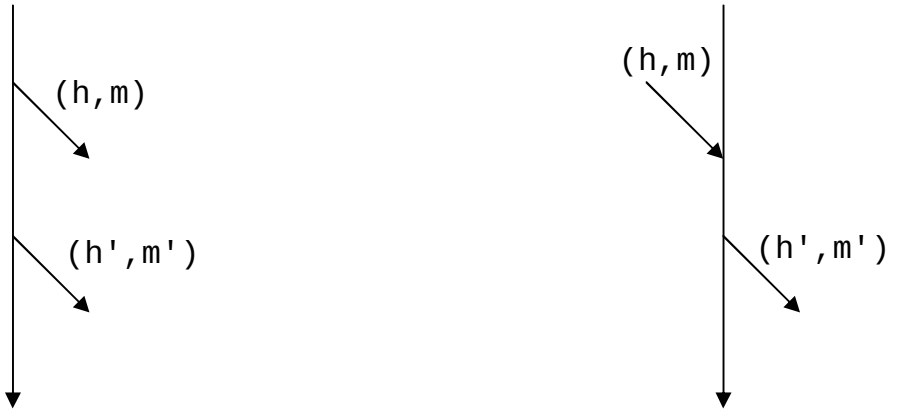



Figure 3.12 Les deux cas de l'ordre immédiat

**Règle d'émission** : Deux messages émis successivement par un même site se succèdent causalement. En conséquence, après chaque émission de messages un site `i` incrémente son horloge logique : $h_i$++
**Règle de réception** : Un message reçu sur un site précède causalement tous les messages qui seront émis par ce site. Aussi à la réception d'un message $(m,h_m)$, un site `i` effectue l'opération suivante : $h_i=\max(h_i,h_m+1)$

Pour appliquer le principe des horloges logiques à notre serveur non réentrant, il faut disposer d'un ordre total entre les messages. Aussi nous comparons les paires (horloge,identité du "propriétaire" de l'horloge) :

$$(h,i)<(h',i') \Leftrightarrow (h<h') \text{ ou } (h=h' \text{ et } i<i')$$

### *4.2 Comment borner des horloges logiques ?*

Il nous reste un dernier problème à résoudre. Les horloges sont des compteurs croissants, aussi un risque de débordement est-il possible. Deux solutions sont possibles pour remédier à ce problème.

La solution la plus simple consiste à :

1. estimer la durée de vie de l'application $d$,
2. calculer la fréquence maximale d'envoi de message $f$,
3. effectuer une multiplication $f.d$ pour obtenir la valeur maximale de l'horloge

On pourra vérifier que des mots de 64 bits permettent de gérer des applications d'une durée de vie de 100 ans !

La deuxième solution consiste à :

1. borner l'écart entre deux horloges logiques par une valeur $B$. Il suffit par exemple de diffuser périodiquement la valeur de son horloge (qui s'incrémente par la même occasion) et de retarder les envois de messages si l'écart maximum est atteint.
2. remplacer l'horloge logique $h_i$ par le compteur $c_i = h_i \bmod P$ (où $P=2.B+1$)
3. appliquer le test de comparaison indiqué dans la proposition ci-dessous qui en garantit la correction.

**Proposition**

$h_i < h_j \Leftrightarrow c_i < c_j \leq c_i+B$ ou $c_j+B<c_i$

**Preuve**

Posons $h_i = c_i+k_i.P$ et $h_j = c_j+k_j.P$

$h_i < h_j$ ($\leq h_i+B$ en vertu du point 1) est équivalent à

- $k_i = k_j$ et $c_i < c_j \leq c_i+B$

ou

- $k_i+1=k_j$ et $c_j+P \leq c_i+B$ autrement dit $k_i+1=k_j$ et $c_j+P-B \leq c_i$ soit d'après la définition de $P$, $k_i+1=k_j$ et $c_j+B < c_i$

# 5 Exercices

***Sujet 1***

On rappelle qu'en environnement asynchrone, l'algorithme de construction d'un arbre de plus courts chemins, vu en cours, a une complexité de $\Theta(n^3)$ messages échangés alors que celui présenté pour un environnement synchrone a une complexité de $\Theta(n^2)$ messages. L'objet de de l'exercice est de développer un algorithme en environnement asynchrone de complexité $\Theta(n^2)$ messages.

**Question 1** Combien y a-t-il d'arcs dans un arbre (quelconque) de `n` nœuds ? Montrer que les distances à la racine de deux voisins du graphe de communication ne peuvent différer de plus d'une unité.

Cet algorithme fonctionne en rondes successives. Au début de la ronde `k`, l'arbre des plus courts chemins restreint aux nœuds à distance `k-1` est construit et durant la ronde, les nœuds à distance `k` sont rattachés. Si à la fin d'une ronde, aucun nouveau nœud n'a été rattaché, la construction est terminée.
Chaque site dispose des variables suivantes :

- $\texttt{voisins}_i$ : sous-ensemble des sites voisins de `i`. Cette constante définit le graphe de communication.
- $\texttt{père}_i$ : identité du site "père" dans l'arbre. Pour des facilités de programmation, elle est initialisée pour l'initiateur à sa propre identité. Elle n'est pas initialisée pour les autres sites.
- $\texttt{distance}_i$ : variable contenant la longueur du chemin allant du site jusqu'à la racine. Elle est initialisée à 0 pour l'initiateur et à l'infini pour les autres sites.
- $\texttt{fils}_i[\texttt{voisins}_i]$ : tableau indicé par les voisins du site qui indique quels sont les sites fils de `i`. Chaque cellule peut prendre l'une des valeurs suivantes (`inconnu,vrai,faux`) . Le tableau est initialisé à `inconnu`.
- $\texttt{nbreq}_i$ : nombre de messages `req` envoyés dont on attend une réponse.
- $\texttt{messpos}_i$ : booléen indiquant si une réponse "positive" a été reçue.
- $\texttt{temp}_i$ : variable temporaire.

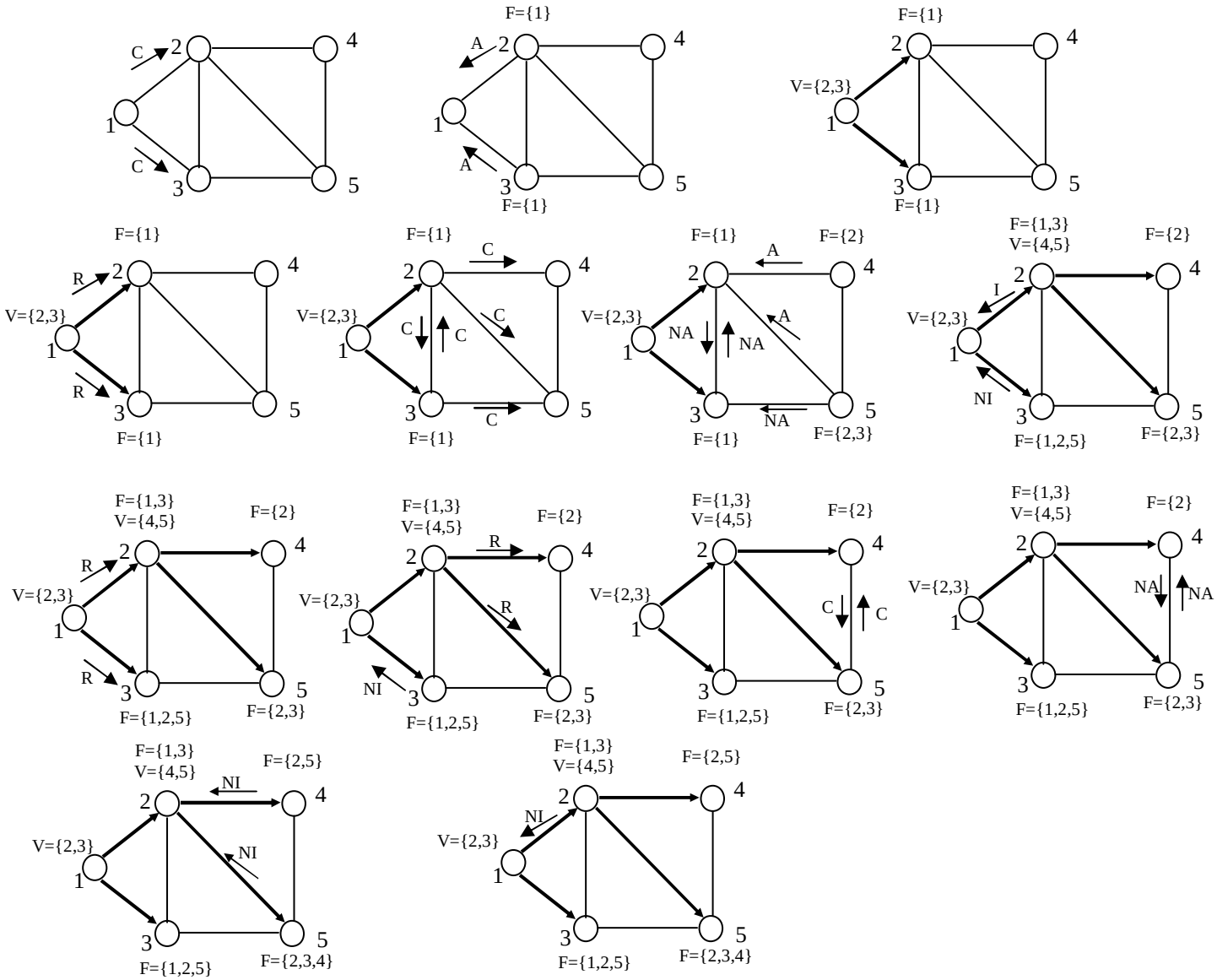
Durant la première ronde, l'initiateur envoie à tous ses voisins un `(cons,1)` qui provoque l'envoi d'un `(acq,1)` par ses voisins. A de la réception du dernier `(acq,1)`, l'initiateur démarre la deuxième ronde. Remarquons que ceci correspond au fait que le tableau $\texttt{fils}_i$ ne contient plus de valeurs `inconnu`.

Les messages qui circulent durant une ronde `k>1` sont les suivants :

- `(req,k)` envoyé à ses fils par un site à distance `<k-1` sur réception d'un `(req,k)` de son père. Pour la racine, ce message est envoyé lors de la réception du dernier message de la ronde précédente.
- `(cons,k)` envoyé aux voisins qui pourraient devenir ses fils par un site à distance `k-1` sur réception d'un `(req,k)` de son père.
- `(acq,k)` envoyé à un site qui devient son père par un site à distance `k` en réponse au premier `(cons,k)` reçu.
- `(nacq,k)` envoyé à un site par un site à distance `k` ou `k-1` en réponse aux autres `(cons,k)`reçus.

- `(info,k)` envoyé à son père par un site à distance `k-1` après la réception d'un message `req` et que son tableau `fils` ne contienne plus de valeurs `inconnu` s'il a acquis un fils ; envoyé à son père par un site à distance `<k-1` après réception d'un message `info` ou `ninfo` de tous ses fils si l'un de ces messages est un `info`.
- `(ninfo,k)` envoyé à son père par un site à distance `k-1` après la réception d'un message `req` et que son tableau fils ne contienne plus de valeurs `inconnu` s'il n'a pas acquis de fils ; envoyé à son père par un site à distance `<k-1` après réception d'un message `info` ou `ninfo` de tous ses fils si aucun de ces messages n'est un `info`.

Le schéma ci-dessous présente une exécution de l'algorithme.



La première rangée de ce schéma représente la première ronde, la deuxième représente la deuxième ronde et les deux dernières la troisième ronde. Le tableau $\texttt{fils}_i$ est représenté par l'identité de cellules qui sont à vrai ou à faux. L'arbre est représenté par les arcs en gras.

Remarquez que, suite à la requête du site 1, le site 3 n'ayant pas de fils peut répondre immédiatement.

**Question 2** Ecrire les primitives suivantes qui sont classées par ordre de difficulté croissante :

- `construire()`(primitive non bloquante appelée uniquement par l'initiateur)
- `sur_réception_de(j,(cons,k))`
- `sur_réception_de(j,(acq,k))`
- `sur_réception_de(j,(nacq,k))`
- `sur_réception_de(j,(info,k))`
- `sur_réception_de(j,(ninfo,k))`
- `sur_réception_de(j,(req,k))`

On n'oubliera pas de distinguer le cas de l'initiateur dans les réceptions de `acq`, `info` et `ninfo`.

**Question 3** Analyse de la complexité. `n` désigne dans la suite le nombre de sites.
4.1 Donnez une borne supérieure au nombre de rondes.
4.2 En vous servant de la question 2, montrez qu'il y moins de `n req`, moins de `n info` ou `ninfo` envoyés au cours d'une ronde.
4.3 Au cours de quelle ronde un site à distance `k` envoie-t-il des `cons` ? Combien en envoie-t-il au plus ?
4.4 Au cours de quelles rondes un site à distance `k` envoie-t-il des `acq` ou des `nacq` ? Combien en envoie-t-il au plus ?
4.5 Déduire de ce qui précède que le nombre de messages envoyés au cours de l'algorithme est borné par $\texttt{C}.n^2$ pour `C` une constante que vous préciserez.

## 6 Références

[Awe85] B. Awerbuch "Complexity of network synchronization" JACM 32 (1985), 804-823.

[Lam78] L. Lamport "Time, clocks, and the ordering of events in a distributed system" Communications of the ACM 21 (1978) pp 558-564.

[LPTF] "Laboratoire Primaire du Temps et des Fréquences" http://www.obspm.fr/lptf

[Ram90] P. Ramanathan, K.G. Shin, R.W. Butler "Fault-tolerant Clock synchronization in Distributed Systems" IEEE Transactions on Computers vol C-39 pp 514-524 avril 1990.

[Tou80] S. Toueg "An all-pairs shortest-path distributed algorithm" Technical Report RC 8327, IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, 1980.

# CHAPITRE IV

# LA CONCURRENCE

*version du 1er décembre 2003*

## 1 Problématique

La concurrence des processus lors de l'accès à des ressources partagées peut provoquer l'incohérence de celles-ci. Gérer la concurrence revient à contrôler que des accès concurrents s'exécutent de manière cohérente. La manière la plus simple de parvenir à cette cohérence consiste à garantir que pour chaque ressource au plus une section de code qui la manipule soit en cours d'exécution. Une telle section de code est appelée *section critique*. Bien que d'autres manières de gérer la concurrence soient possibles (e.g. la séquentialisation des transactions de bases de données), nous nous limiterons dans ce chapitre à étudier la mise en oeuvre de l'exclusion mutuelle entre sections critiques.

De manière plus précise, lorsqu'un processus applicatif désirera exécuter une section critique, son programme se présentera comme suit :

```
Prologue(); Section critique; Epilogue()
```

`Prologue` et `Epilogue` sont des primitives du service qui devront garantir les deux propriétés caractéristiques de l'exclusion mutuelle :

- A tout instant, au plus un processus est en cours d'exécution d'une section critique. Ce type de propriété est appelée propriété de *sûreté* et traduit le fait que "quelque chose de mal n'arrivera jamais".
- Tout processus demandant à exécuter une section critique (par appel à `Prologue`), pourra l'exécuter (par retour de `Prologue`) au bout d'un temps fini. Ce type de propriété est appelée propriété de *vivacité* et traduit le fait que "quelque chose de bien finira par arriver".

De nombreux algorithmes ont été proposés dans la littérature. Dans un souci pédagogique nous présenterons trois d'entre eux (parmi les plus anciens). Ces algorithmes sont tous basés sur les horloges logiques présentées au chapitre précédent. Chaque nouvel algorithme se construit par une modification conceptuelle du précédent et diminue la complexité (mesurée en nombre de messages échangés par exécution d'une section critique).

Avant d'étudier ces algorithmes, examinons pourquoi la technique de circulation d'un jeton sur un anneau pour assurer l'exclusion mutuelle n'est pas satisfaisante. Nous mesurons la complexité en nombre de messages échangés pour une section critique. Or il apparaît que même si les sites ne sont pas désireux d'exécuter une section critique, le jeton doit circuler. Ce qui signifie que le nombre de messages échangés peut s'accroître indéfiniment sans une seule entrée en section critique !

# 2 Algorithme de Lamport [Lam78]

Comme les deux algorithmes suivants, cet algorithme repose sur le mécanisme des horloges logiques. Aussi pour éviter l'écriture d'un code répétitif, nous supposerons que **la mise à jour de l'horloge logique est faite dans la couche réseau** à l'émission et à la réception. Ceci impose cependant à la couche service de fournir une valeur d'horloge dans tous les messages de l'algorithme (le lecteur pourra vérifier que cette contrainte est respectée).

## *2.1 Principe et réalisation de l'algorithme*

Le protocole échange trois types de message :

- des requêtes d'exécution de section critique diffusées à tous les autres sites par le site demandeur,
- des acquittements de requête pour indiquer que le site a "enregistré" la requête,
- des libérations diffusées à tous les autres sites par un site qui a terminé l'exécution d'une section critique.

Sans entrer maintenant dans les détails de l'algorithme et en posant `n` le nombre de sites, on voit immédiatement qu'une section critique s'accompagne de `3.(n-1)` messages : `(n-1)` requêtes, `(n-1)` acquittements et `(n-1)`libérations.

Le point clef de l'algorithme est la condition d'entrée en section critique. Pour ce faire, chaque site maintient un tableau de messages indicé par les identités des sites. Chaque cellule de ce tableau contient le type du message et son heure. Lorsqu'un site désire exécuter sa section critique, **il met à jour sa propre cellule avec sa requête**. La condition d'entrée en section critique est alors **d'avoir dans sa cellule le message le plus âgé du tableau**. Rappelons qu'au sens des horloges logiques, la "date de naissance" d'un message est constitué du doublet (horloge, identité du site) ce qui évite le cas d'égalité des âges de deux messages.

Toutes les cellules sont initialisées avec un message de libération daté de l'heure `–1` de telle sorte qu'une requête initiale doit donner lieu à des acquittements pour exécuter la section critique. Il reste à préciser la règle de mise à jour des autres cellules du tableau. On pourrait penser que tout message reçu de j provoque la mise à jour de la cellule j du tableau mais l'exemple introductif décrit ci-dessous (figure 4.1) montre que cette règle doit être modulée.

Dans ce scénario, les sites `1` et `2` désirent entrer en section critique. Ils diffusent donc leur requête et mettent respectivement à jour la cellule de leur tableau. Lorsque le site `2` reçoit l'acquittement du site `3` puis la requête du site `1`, il met à jour « leur » cellule de son tableau. Il ne peut entrer en section critique car la requête du site `1` est plus âgée - `(0,1)<(0,2)` - . Lorsque le site `1` reçoit la requête du site `2`, il met à jour la cellule correspondante de son tableau. Il ne peut entrer en section critique car le message de libération du site `3` est plus âgé - `(-1,3)<(0,1)` - . Que doivent faire les sites `1` et `2` lorsqu'ils reçoivent l'acquittement de l'autre site ? S'ils mettent à jour leur tableau, le site `2` rentre immédiatement en section critique, tandis que le site `1` entrera en section critique à la réception de l'acquittement du site `3`. Il n'y aura donc pas exclusion mutuelle entre les sections critiques. Le problème vient du fait que la mise à jour du tableau par l'acquittement provoque l'oubli de la requête qui pourtant n'est pas encore traitée.

Nous sommes maintenant en mesure de définir la règle de mise à jour du tableau lors d'une réception. Tout message arrivant est enregistré dans le tableau **à l'exception d'un acquittement qui remplacerait une requête**.

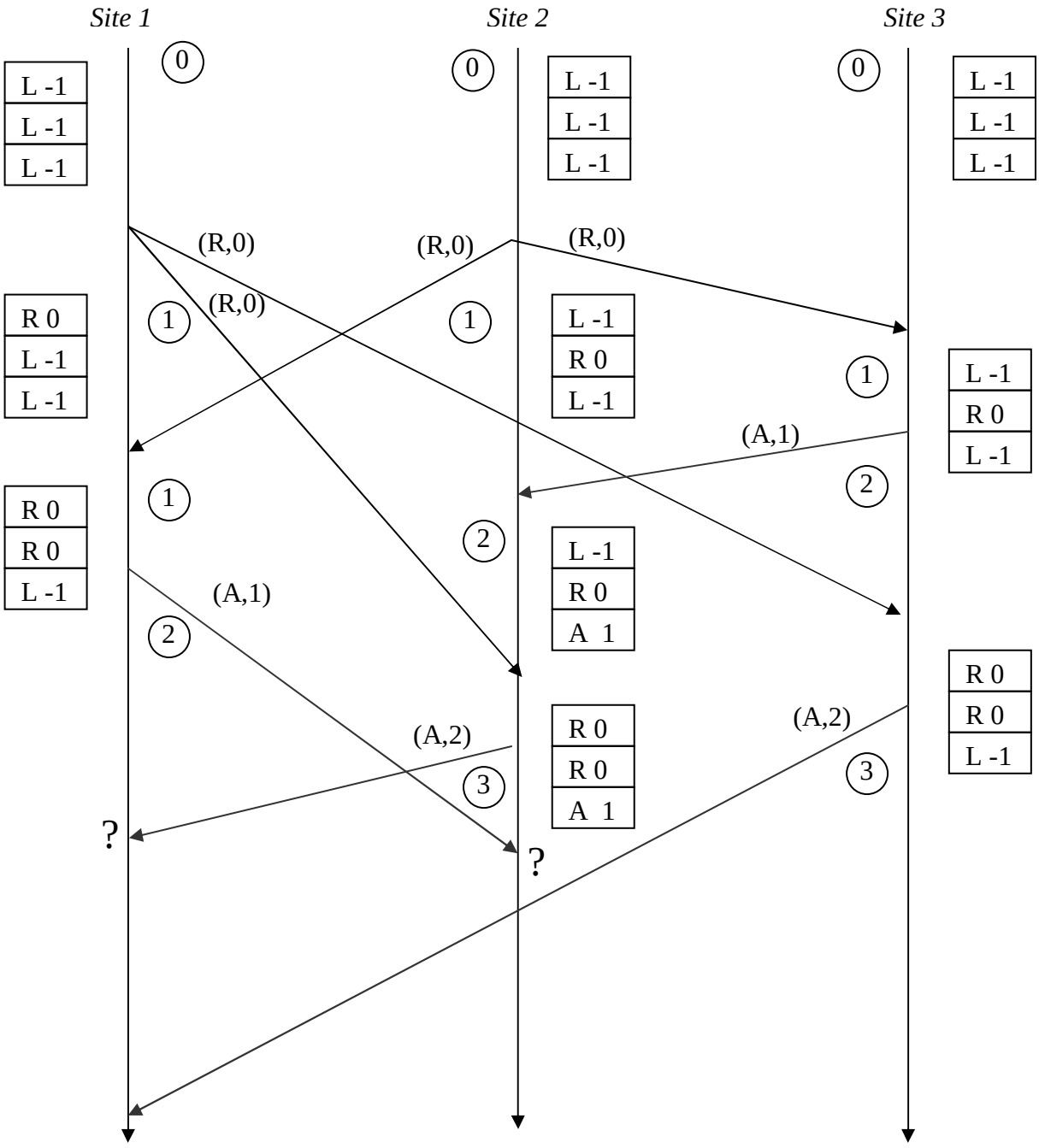



Figure 4.1 : Un scénario d'entrée en section critique (début)

La suite du scénario est présenté à la figure 4.2. Le site `1` entre en section critique à la réception de l'acquittement du site `3`. Lorsque l'application du site `1` termine l'exécution de sa section critique et appelle `Epilogue()`, la diffusion des messages de libération provoque à la réception par le site `2`, son entrée en section critique. On remarque que la diffusion des messages de libération par le site `2` est sans influence sur le comportement des autres sites. Nous reviendrons sur ce point au prochain algorithme.

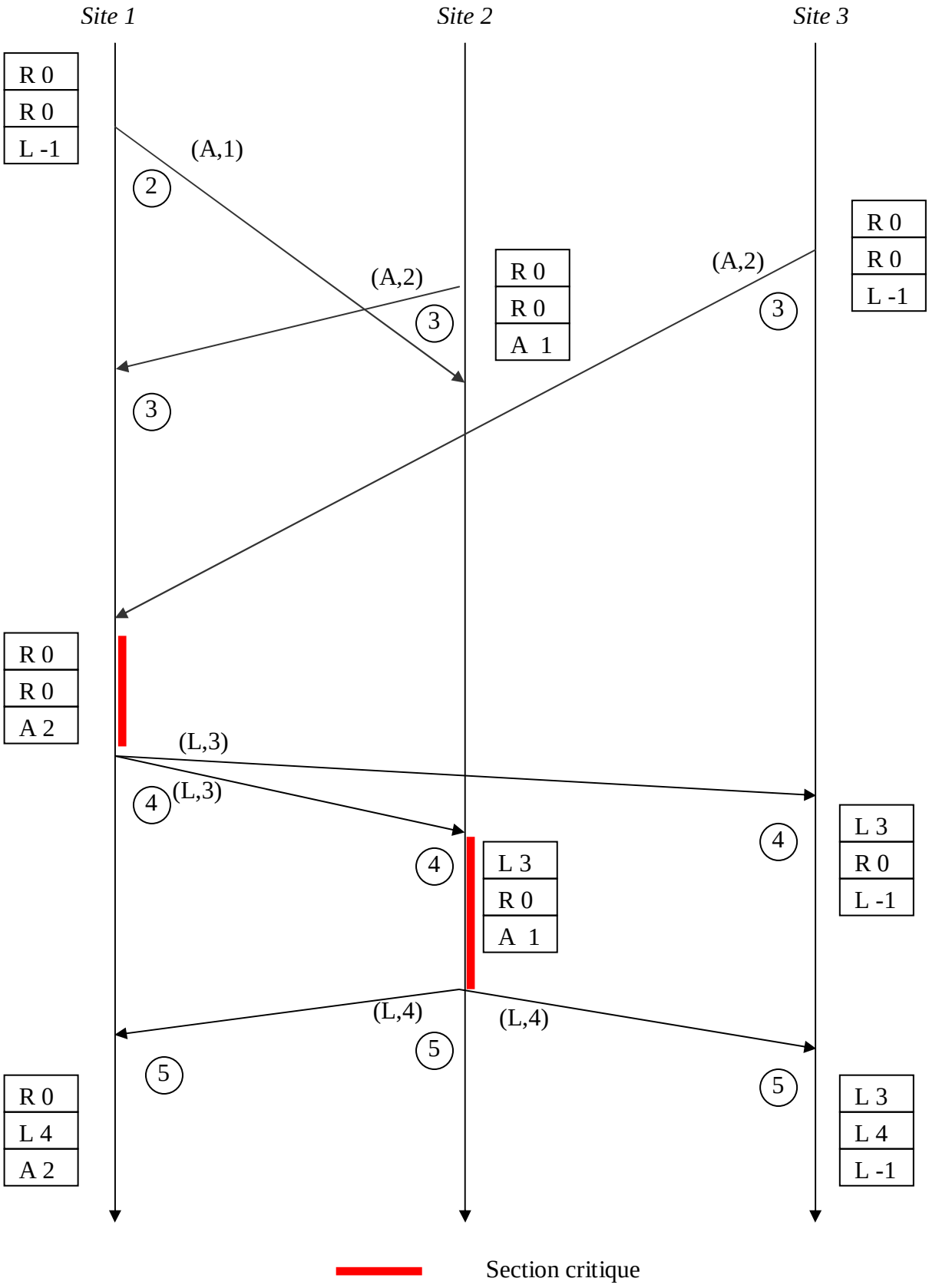



Figure 4.2 : Un scénario d'entrée en section critique (suite)

Variables du site `i`

- $h_i$ : variable contenant la valeur de l'horloge locale du site i. Elle est initialisée à `0`.
- $Mess_i[1..N]$ : tableau des messages. Chaque cellule de ce tableau est composée des deux champs `<type,heure>`. `type` prend ses valeurs parmi {`req`, `acq`, `lib`} et est initialisé à `lib`. `heure` est initialisé à `-1`.

Algorithme du site `i`

Avant d'entrer en section critique, un site enregistre sa requête courante, la diffuse et attend que la requête soit le message le plus âgé du tableau.

```
prologue()
Début
     Mess_i[i].heure = h_i;
     Mess_i[i].type = req;
     Diffuser(req,h_i); // h_i est incrémenté (par la couche réseau)
     Attendre(∀j≠i,(Mess_i[i].heure,i)<(Mess_i[j].heure,j);
Fin
```

Lorsqu'un site quitte la section critique, il diffuse un message de libération à tous les autres sites.

```
épilogue()
Début
     Diffuser(lib,h_i);
Fin
```

La réception d'une requête provoque l'envoi d'un accusé de réception. Le message est enregistré dans le tableau excepté dans le cas discuté plus haut.

```
sur_réception_de(j,(tp,h)) // h_i=max(h_i,h+1) (par la couche réseau)
Début
     Si tp == req Alors
          envoyer_à(j,(acq,h_i));
     Finsi
     Si (tp != acq) || (Mess_i[j].type != req) Alors
          Mess_i[j].heure = h;
          Mess_i[j].type = tp;
     Finsi
Fin
```

## *2.2 Preuve de l'algorithme*

A titre d'exemple, nous allons établir la propriété de vivacité et de sûreté de l'algorithme de Lamport. Nous invitons le lecteur à établir des preuves similaires pour les deux autres algorithmes.

### 2.2.1 Propriété de vivacité

Nous raisonnons par l'absurde : supposons qu'il existe une exécution (infinie) au cours de laquelle un ensemble de sites $E_0$ reste indéfiniment bloqué au cours d'un appel à `prologue()`. Les autres sites peuvent être partitionnés en deux catégories : $E_1$ l'ensemble des sites qui n'accèdent qu'un nombre fini de fois à une section critique et $E_2$ l'ensemble des sites qui accèdent un nombre infini de fois à une section critique.

Nous nous nous plaçons en un instant $t_0$ où :

1. Tous les messages des requêtes associées aux appels à `prologue()` non terminés des sites de $E_0$ ont été reçus par les autres sites et donc enregistrés dans leur tableau. Ces messages restent indéfiniment dans les tableaux.
2. Tous les acquittements associés à ces requêtes ont été reçus.
3. Tous les messages de libération associés aux derniers appels à `épilogue()`des sites de $E_1$ ont été reçus par les autres sites et donc enregistrés dans leur tableau.

Montrons d'abord que $E_2$ est vide. En effet, supposons qu'il existe un site $i \in E_2$, alors (par définition de $E_2$) l'application de `i` appelle `prologue()` après l'instant $t_0$. D'après le mécanisme des horloges logiques, cette requête a nécessairement une heure plus grande que celle d'une requête non traitée d'un quelconque site de $E_0$. D'après le point 1, i reste bloqué indéfiniment. Ce qui est contradictoire avec la définition de $E_2$.

Soit maintenant $i \in E_0$ et $j \in E_1$. Examinons le contenu de la cellule $Mess_i[j]$ après l'instant $t_0$. L'acquittement par `j` de la requête non traitée de `i` a été reçu.

- S'il a été enregistré, il ne peut bloquer le site `i` puisque son heure est plus grande que celle de la requête. Les messages, qui éventuellement lui succèdent dans cette cellule, ayant des heures plus grandes ne peuvent non plus bloquer le site `i`.
- S'il n'a pas été enregistré, cela signifie qu'une requête était présente dans $Mess_i[j]$. Dans ce cas, (par définition de $E_1$) le site `i` recevra au pire à l'instant $t_0$ le message de libération correspondant qui ne pourra bloquer le site `i` puisque son heure est plus grande que celle de l'acquittement donc de la requête non traitée de `i`. Les messages, qui éventuellement lui succèdent dans cette cellule, ayant des heures plus grandes ne peuvent non plus bloquer le site `i`.

Donc un site de $E_1$ ne peut bloquer un site de $E_0$ à partir de $t_0$.

Soit maintenant le site $i \in E_0$ dont la requête non traitée est la plus âgée. A l'instant $t_0$, il ne peut être bloqué par un message des autres sites de $E_0$ ni par les messages des sites de $E_1$. En conséquence son appel à `prologue()` devrait se terminer. D'où la contradiction.

2.3.2 Propriété de sûreté

Nous raisonnons par l'absurde : supposons que deux sites soient à un même instant en cours d'exécution d'une section critique.

Appelons $i_1$ le site dont la requête correspondante est la plus « jeune » et $i_2$ l'autre site. Le site $i_1$ diffuse la requête correspondante $r_1$ à l'heure logique $h_1$ et le site $i_2$ diffuse la sienne ($r_2$) à l'heure logique $h_2$. Appelons $m$ le message émis par le site $i_2$ à destination du site $i_1$, présent dans le tableau au moment où $i_1$ pénètre en section critique.

Remarquons tout d'abord que $r_2$ ne précède pas m car au moment où $i_1$ pénètre en section critique, $i_2$ n'a pas terminé la section critique correspondant à $r_2$. Il n'a donc pas envoyé de message de libération et les messages d'acquittement ne peuvent remplacer $r_2$ dans le tableau de $i_1$.

D'autre part $r_2$ et m ne sont pas confondus car la requête du site $i_2$ étant plus âgée, elle empêcherait $i_1$ de pénétrer en section critique.

Le message $m$ précède donc $r_2$ (cf. figure 4.3). Soit $h$ l'heure logique de $m$. Puisque $i_1$ pénètre en section critique $(h_1,i_1) < (h,i_2)$. Puisque $m$ précède $r_2$, $(h,i_2) < (h_2,i_2)$. Par transitivité, $(h_1,i_1) < (h_2,i_2)$. Ce qui est contradictoire avec nos hypothèses.

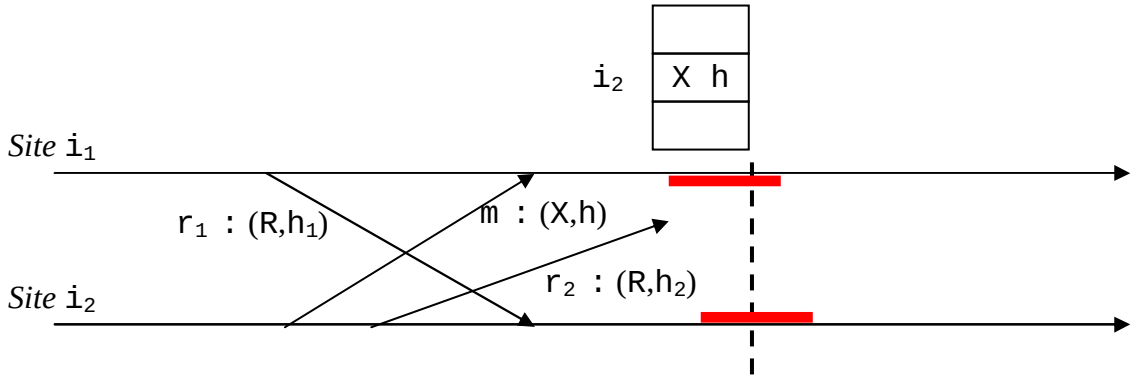



Figure 4.3 : Deux sites simultanément en section critique

# 3 Algorithme de Ricart et Agrawala [Ric81]

## *3.1 Principe de l'algorithme*

Dans l'algorithme de Lamport, un acquittement s'interprète comme le fait d'enregistrer la requête. Le principal apport de ce nouvel algorithme consiste à modifier la sémantique de l'acquittement.

A présent, un acquittement du site `j` à une requête du site `i` signifie que `j` autorise `i` à pénétrer en section critique. La condition d'entrée en section en section critique devient maintenant **la réception d'un acquittement venant de chacun des autres sites**. Comme chaque site n'envoie qu'un seul acquittement relatif à une requête, il suffit de compter le nombre d'acquittements reçus.

Il reste maintenant à définir la politique du site `j` lorsqu'il reçoit cette requête. Si son application n'est pas en attente ou en cours d'exécution d'une section critique, l'acquittement sera délivré immédiatement. Dans le cas contraire, `j` compare l'âge de sa requête avec celui de la requête de `i`. Si cette dernière est plus âgée, l'acquittement est délivré immédiatement. Dans le cas contraire, `j` diffère l'acquittement jusqu'à la fin de sa section critique. Les messages de libération de l'algorithme de Lamport s'interprètent alors comme des acquittements différés. Bien entendu, il s'agit pour un site de mémoriser, les sites qu'il a retardés pour réaliser cet envoi différé.

Il est immédiat que pour une entrée en section critique, `2.(n-1)` messages sont échangés : `(n-1)` requêtes et `(n-1)` acquittements. La complexité de cet algorithme est donc meilleure que celle de l'algorithme précédent.

Dans le scénario de la figure 4.4, les sites `1` et `2` désirent entrer en section critique. Ils diffusent donc leur requête et modifient leur état, mémorisent l'heure de leur requête et attendent les acquittements. Le site `3` acquitte les deux requêtes car son application n'est pas intéressée (au moment de la réception des requêtes) par une section critique. Lorsque le site `1` reçoit la requête du site `2`, il retarde sa réponse car sa propre requête est plus âgée. Par contre le site `2` lui renvoie immédiatement un acquittement. A la réception de l'acquittement du site `3`, le site `1` entre en section critique. Lorsque cette section critique est terminée, le site `1` envoie un acquittement aux sites qu'il a retardés (dans le cas présent au site `2`). A la réception de cet acquittement, le site `2` pénètre à son tour en section critique.

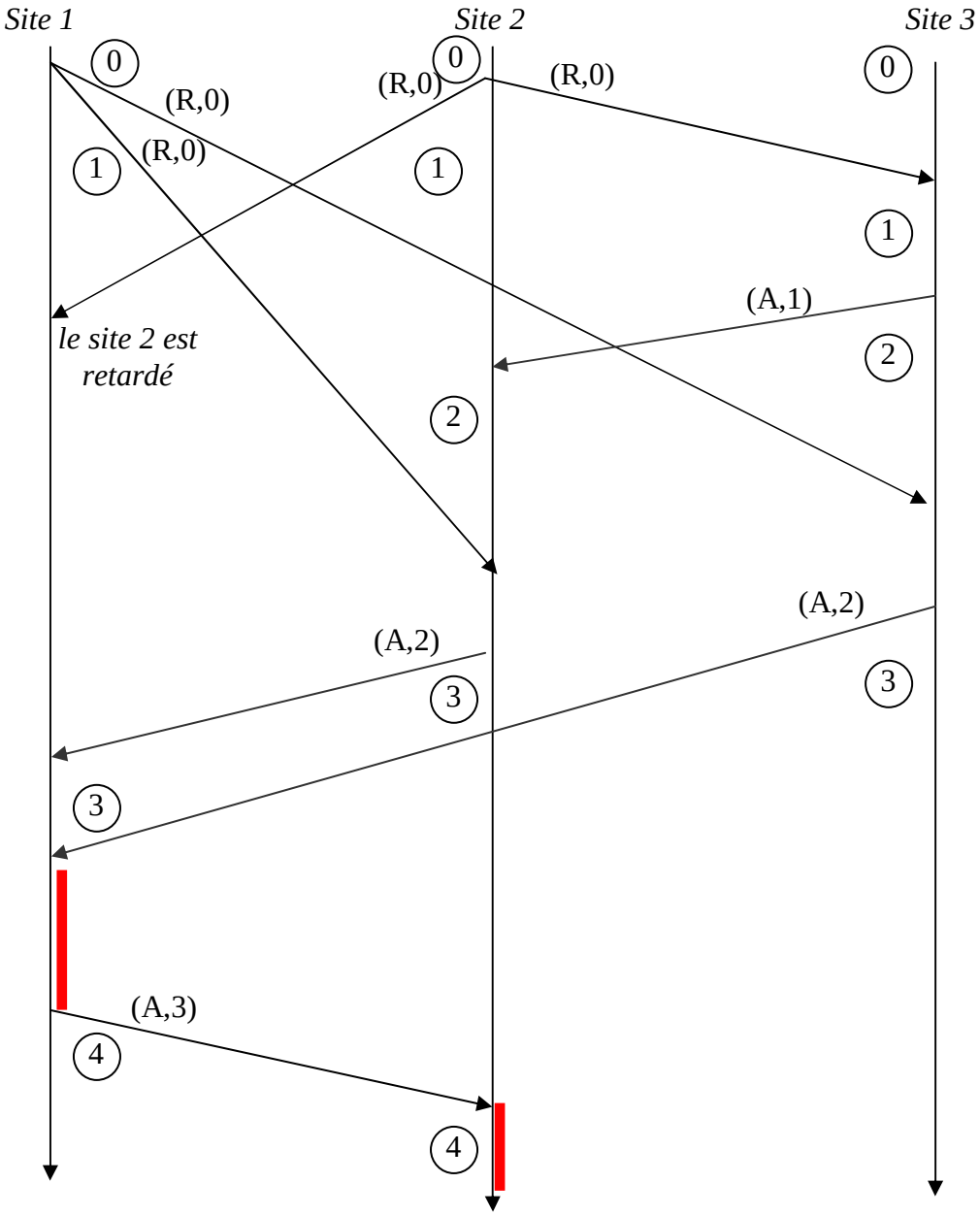



Figure 4.4 : Un scénario d'entrée en section critique

### *3.2 Description*

Variables du site `i`

- $h_i$ : valeur de l'horloge logique. Elle est initialisée à `0`.
- $état_i$ : état du site. Cette variable peut prendre l'une des deux valeurs {`repos`, `en_cours`}. Initialement, tous les sites sont dans l'état `repos`.
- $hreq_i$ : heure de la requête courante `i`
- $nbacq_i$ : compteur des acquittements reçus.
- $Retardé_i[1..N]$ : tableau de booléens. $Retardé_i[j]$ est vrai lorsque `i` retarde l'acquittement d'une requête `j`.
- $temp_i$ : variable temporaire.

Algorithme du site i

Lorsqu'un site désire entrer en section critique, il diffuse une requête, puis se met en attente de tous les acquittements.

```
prologue()
Début
     hreq_i = h_i;
     nbacq_i = 0;
     Pour temp_i de 1 à n faire // n est le nombre de sites
          Retardé_i[temp_i] = Faux;
     Finpour
     état_i = en_cours;
     Diffuser(req,h_i);
     Attendre(nbacq_i == (n-1));
Fin
```

A la sortie de la section critique, les site envoie des acquittements aux sites qu'il a retardés.

```
épilogue()
Début
     Pour tout temp_i de 1 à n faire
          Si Retardé_i[temp_i] Alors
               envoyer_à(temp_i,(acq,h_i));
          Fin si
     Finpour
état_i = repos;
Fin
```

La réception de la requête suit la politique décrite plus haut.

```
sur_réception_de(j,(req,h))
Début
     Si (état_i == en_cours)&&((hreq_i,i)<(h,j)) Alors
          Retardé_i[j] = Vrai;
     Sinon
          envoyer_à(j,(acq,h_i));
     Finsi
Fin
```

La réception d'un acquittement incrémente le compteur des acquittements reçus.

```
sur_réception_de(j,(acq,h))
Début
     nbacq_i++;
Fin
```

# 4 Algorithme de Carvalho et Roucairol [Car83]

## *4.1 Principe de l'algorithme*

Dans l'algorithme de Ricart et Agrawala, un acquittement s'interprète comme le fait d'autoriser l'entrée en section critique. Ici aussi l'apport de ce nouvel algorithme consiste à modifier la sémantique de l'acquittement.

A présent, un acquittement du site `j` à une requête du site `i` signifie que `j` autorise `i` à pénétrer en section critique pour la section critique courante **mais aussi pour les sections critiques suivantes**. L'acquittement s'interprète alors comme l'envoi d'une permission partagée entre `i` et `j`. Si `j` veut par la suite pénétrer en section critique, il doit réclamer la permission qu'il a concédée à `i`. La condition d'entrée en section critique devient maintenant **la possession des permissions partagées avec chacun des autres sites**. On remarque que les permissions sont similaires aux jetons multiples associés à l'algorithme du rendez-vous (vu au chapitre 2).

Pour aboutir au développement de l'algorithme, il faut cependant résoudre quelques problèmes liés à cette idée. Tout d'abord il s'agit de répartir initialement les jetons entre les sites. Une idée simple à réaliser consiste à ce que le jeton du couple de sites `{i,j}` soit possédé par le site d'identité `max(i,j).`

Une première modification importante est que lors du `prologue()` seules les permissions manquantes seront réclamées. Ce qui signifie qu'un site peut **entrer immédiatement en section critique sans échanger un seul message** !

Illustrons maintenant les spécificités de cet algorithme par deux scénarios d'exécution. Examinons la figure 4.5. Les sites `1` et `2` désirent exécuter une section critique. Il manque au site `2` le jeton `(2,3)` qu'il réclame au site `3`. Il manque au site `1`, les jetons `(1,3)` et `(2,3)` qu'il réclame aux sites correspondants. Notons que la requête du site `1` est plus âgée que celle du site `2`. Le site `2` ayant obtenu le jeton qui lui manquait entre en section critique. En cours de section critique, il reçoit la requête du site `1`. Bien que celle-ci soit plus âgée que sa propre requête, il ne peut donner le jeton `(1,2)` car l'exclusion mutuelle ne serait plus garantie (la preuve de la correction de l'algorithme de Lamport implique que ceci ne peut arriver dans les algorithmes précédents). Il faut donc que l'algorithme distingue trois états pour mettre en oeuvre sa politique de choix. Soit l'application n'est pas intéressée par une section critique (`repos`), soit elle attend dans le prologue (`attente`), soit elle exécute la section critique (`en_SC`).

La figure 4.6 est une variation du scénario précédent dans lequel la requête du site `1` arrive au site avant l'entrée en section critique. Dans ce cas, la priorité donnée à la requête la plus âgée s'applique et le site `2` donne son jeton au site `1`. Il doit cependant le lui réclamer aussitôt car il ne pourrait entrer en section critique. Il est évident que sa requête sera retardée (excepté si entre l'arrivée des deux messages le site `1` a eu le temps d'exécuter sa section critique) mais l'important est ici de récupérer le jeton perdu. Attention, toutes les requêtes sont datées de la même heure qu'elles soient émises immédiatement, ou sur une perte ultérieure du jeton. Ceci permet d'éviter la famine d'un site.

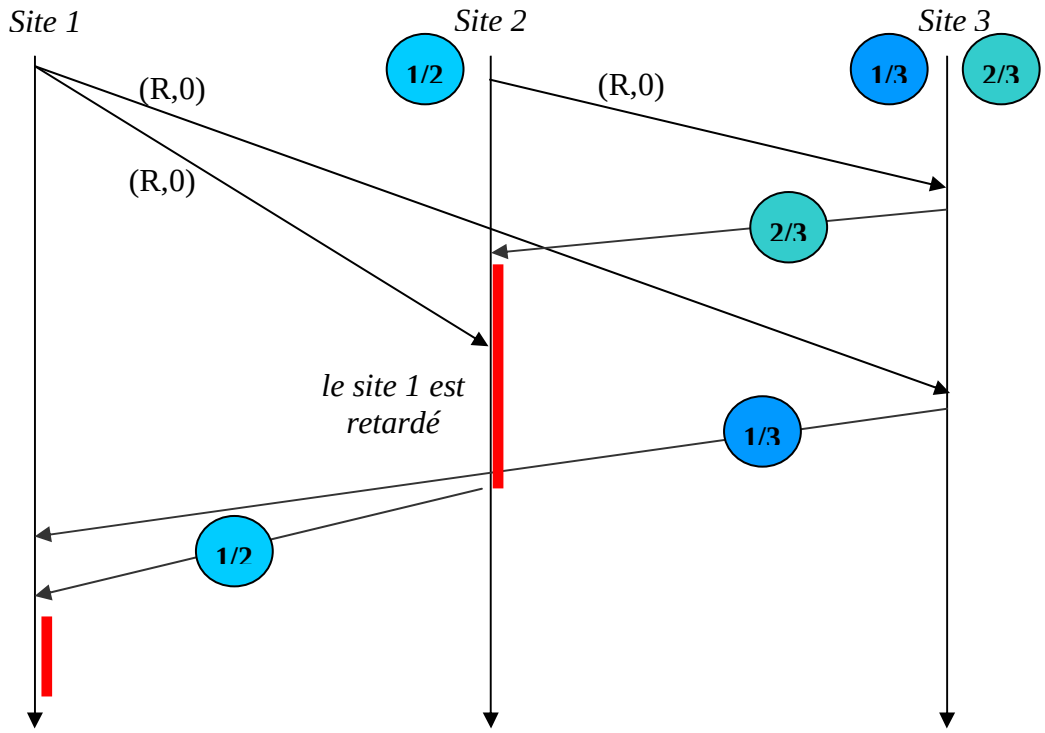



Figure 4.5 : Une requête retardée par une requête plus jeune

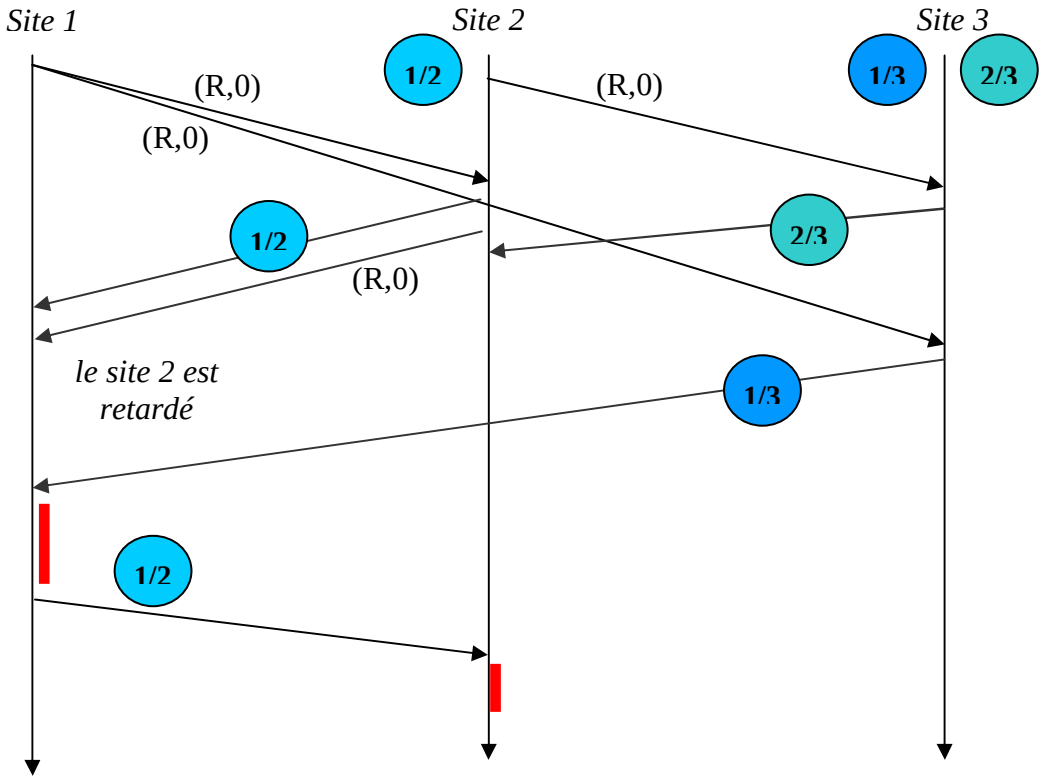



Figure 4.6 : Un site qui donne un jeton et le réclame aussitôt

### *4.2 Description*

Variables du site `i`

- $h_i$ : valeur de l'horloge logique. Elle est initialisée à `0`.
- $état_i$ : état du site. Cette variable peut prendre l'une des deux valeurs {`repos`, `attente`, `en_SC`}. Initialement, tous les sites sont dans l'état `repos`.
- $hreq_i$ : heure de la requête courante `i`
- $Jeton_i[1..N]$ : tableau de booléens indiquant la présence des jetons. $Jeton_i[j]$ est initialisé à la valeur $(i \geq j)$.
- $Retardé_i[1..N]$ : tableau de booléens. $Retardé_i[j]$ est vrai lorsque `i` retarde l'acquittement d'une requête `j`.
- $temp_i$ : variable temporaire.

Algorithme du site `i`

Le site réclame les jetons qui lui manquent et attend de posséder tous les jetons pour entrer en section critique.

```
prologue()
Début
     hreq_i = h_i;
     Pour temp_i de 1 à n faire // n est le nombre de sites
          Retardé_i[temp_i] = Faux ;
          Si Jeton_i[temp_i] == Faux Alors
               envoyer_à(j,(req,hreq_i));
          Finsi
     Finpour
     état_i = attente;
     Attendre(∀ j,  Jeton_i[j] == Vrai);
     état_i = en_SC;
Fin
```

A la sortie de la section critique, le site envoie les jetons aux sites qu'il a retardés.

```
épilogue()
Début
     Pour temp_i de 1 à n faire
          Si Retardé_i[j] Alors
               envoyer_à(j,(acq,h_i));
               Jeton_i[j] = Faux;
          Finsi
     Finpour
     état_i = repos;
Fin
```

Lors de la réception par le site `i` d'une requête émise par le site `j`, la décision prise par `i` dépend de son état :

- Si le site `i` n'est pas intéressé par une section critique, il envoie le jeton à `j`.
- Si le site `i` exécute sa section critique ou si sa requête est plus âgée que la requête de `j`, il retarde l'envoi du jeton jusqu'à la fin de sa section critique.
- Sinon il envoie le jeton à `j` et le lui réclame aussitôt.

```
sur_réception_de(j,(req,h))
Début
     Si état_i == repos Alors
          envoyer_à(j,(acq,h_i));
          Jeton_i[j] = Faux;
     Sinon si  (état_i==en_SC)||((hreq,i)<(h,j)) Alors
          Retardé_i[j] = Vrai;
     Sinon
          envoyer_à(j,(acq,h_i));
          Jeton_i[j] = Faux;
          envoyer_à(j,(req,hreq_i));
     Finsi
Fin
```

Un jeton arrive ...

```
sur_réception_de(j,(acq,h))
Début
     Jeton_i[j] = Vrai;
Fin
```

### *4.3 Complexité de l'algorithme*

Si un site possède tous ses jetons, il entre immédiatement en section critique. Dans le meilleur des cas, il n'y a pas de messages échangés pour entrer en section critique. Démontrons maintenant qu'un site ne réclame un jeton particulier qu'**au plus une fois**.

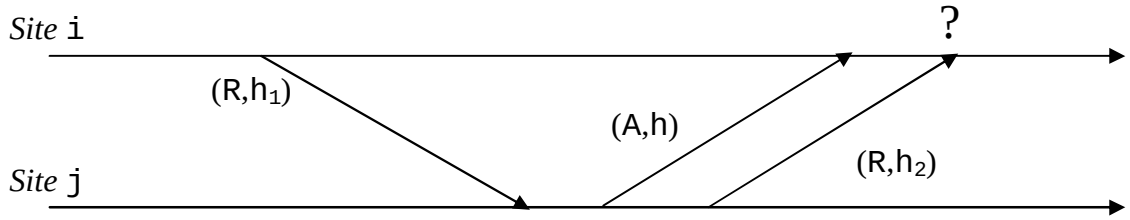



Figure 4.7 : Un jeton récupéré et réclamé une nouvelle fois

Sur la figure 4.7, le site `i` a récupéré le jeton `(i,j)` et le site `j` le lui réclame avant que `i` ne soit entré en section critique. Deux cas se présentent :

- Soit l'émission de la requête correspond à l'appel à `prologue()` et donc $h_2>h>h_1$ auquel cas le site `j` sera retardé.
- Soit l'émission de la requête correspond à la perte du jeton lors de la réception de la requête de `i` et donc (voir la réception d'une requête) $(h_2,j)>(h_1,i)$ auquel cas le site `j` sera ici aussi retardé.

Par conséquent, le nombre de messages nécessaire à une section critique varie entre `0` et `2.(n-1)`puisqu'il y a au plus `n-1` requêtes et donc `n-1` acquittements.

En fonctionnement normal durant une période relativement longue, un petit nombre `p` parmi `n` de sites est actif (comportement sporadique du réseau local). Passée la première entrée en section critique pour chacun des `p` sites, les jetons échangés ne le seront qu'entre ces sites conduisant à une complexité de `2.(p-1)<< 2.(n-1)`. Autrement dit, cet algorithme est très efficace dans la pratique même si sa complexité au pire des cas n'est pas meilleure que celle de l'algorithme précédent.

## 5 Exercices

***Sujet 1***
On se propose de définir un nouvel algorithme d'accès à une section critique n'utilisant pas les horloges logiques. Cet algorithme est une adaptation de l'algorithme de Carvahlo et Roucairol. Deux sites quelconques partagent une permission. A la différence de l'algorithme précedent, chaque site tient à jour pour une permission qu'il possède le fait qu'il est prioritaire ou non pour cette permission. Initialement la permission `(i,j)` est possédée par le site d'identité `max(i,j)` et ce site est prioritaire pour l'utilisation de cette permission.

Pour entrer en section critique un site doit posséder toutes les permissions dont il est copropriétaire : il demande donc celles qui lui manquent.

Lorsqu'un site `i` reçoit une requête de `j` , il réagit selon les règles suivantes :
• `i` est au repos,
    il renvoie immédiatement la permission
• `i` est en attente de section critique,
    si `i` a la permission et qu'il n'est pas prioritaire pour cette permission
        il renvoie immédiatement la permission
        il la réclame aussitôt
    sinon
        il diffère sa réponse
• `i` est en section critique,
    il diffère sa réponse

Lorsqu'un site sort de sa section critique, il devient non prioritaire pour toutes les permissions qu'il possède et il renvoie les permissions à tous les sites pour lesquels il avait différé sa réponse.

L'indication du site prioritaire d'une permission est envoyée avec la permission.

**Question 2** Décrire les variables de chaque site nécessaires à cet algorithme et leur initialisation éventuelle.

**Question 3** Décrire les réceptions de requête et de permission ainsi que la l'entrée et la sortie de section critique.

**Question 4** Avec trois sites, déroulez le scénario où le site `1` et le site `2` demandent simultanément la section critique. Vous devrez représenter les variables de chaque site et l'échange des messages.

**Question 5** Expliquez **informellement** comment cet algorithme assure qu'un site qui demande la section critique finira par l'obtenir.

***Sujet 2***

On se propose de définir un algorithme d'accès à un fichier partagé par des sites. Ce fichier peut être accédé en lecture et en écriture. Les deux contraintes à respecter sont les suivantes :

*Les lectures et les écritures sont exclusives*

*Les écritures sont mutuellement exclusives*

Cet algorithme est une adaptation de l'algorithme de Ricart et Agrawala. Lorsqu'un site veut effectuer une opération (de lecture ou d'écriture) sur le fichier il envoie sa demande d'opération à tous les autres sites. Sur réception d'une demande d'opération, un site accorde sa permission si :
- soit il n'est pas intéressé par le fichier
- soit sa propre demande d'opération est compatible avec celle du site distant
- soit sa demande est plus récente que la demande du site distant

Dans le cas contraire, il diffère son accord. A la fin d'une opération, un site accorde toutes les permissions aux sites qu'il a retardé.

**Question 1** Décrire les variables de chaque site nécessaires à cet algorithme.

**Question 2** Décrire les réceptions de requête et de permission, les demandes de lecture et d'écriture et les fins de lecture et d'écriture.

**Question 3** Avec trois sites, déroulez le scénario où le site `1` , le site `2` et le site `3` demandent simultanément un accès - le site `1` et le site `3` en lecture, le site `2` en écriture - . Vous devrez représenter les variables de chaque site et l'échange des messages.

**Question 4** Expliquez **informellement** pourquoi les contraintes sont satisfaites.

***Sujet 3***

La diffusion atomique `Diffuser_Atomique(m)` est une primitive de diffusion utilisée par des applications qui a la propriété suivante :
"Si `m` et `m'` sont deux messages diffusés par cette primitive (pas nécessairement par le même émetteur) alors ils seront reçus par l'application de chaque site dans un même ordre"

Cette propriété permet d'ordonner les messages diffusés selon leur instant de traitement par l'application (et ceci indépendamment du site).
***Attention, il ne faut pas confondre cette primitive avec la primitive de la couche réseau diffuser(m) utilisée par les algorithmes du cours !***

**Question 1** Donnez un exemple d'application informatique où il y a necessité de diffusion atomique.

Une manière simple de gérer la diffusion atomique consiste à séquentialiser les diffusions atomiques. Autrement dit, lorsqu'un site veut effectuer une diffusion atomique :
• son service demande la section critique,
• lorsqu'il l'a obtenue, il diffuse avec la primitive de bas niveau le message de l'application,
• il libère la section critique, une fois reçus les acquittements de la diffusion envoyés par les autres services.

On vous demande d'adapter l'algorithme de Carvahlo et Roucairol pour cette gestion.

**Question 2** Décrire les variables de chaque site nécessaires à cet algorithme (elles incluent les variables de l'algorithme originel).

**Question 3** Décrire l'implémentation de `Diffuser_Atomique(m)` et des réceptions de message.

**Question 4** Avec trois sites, déroulez le scénario où le site `1` et le site `2` décident simultanément de diffuser un message. Vous devrez représenter les variables de chaque site et l'échange des messages.

**Question 5** Quels sont les inconvénients d'une telle gestion ?

***Sujet 4 : Implémentation d'un sémaphore sur un anneau virtuel***

Dans ce problème, tous les sites sont interconnectés (réseau totalement maillé). Ils sont de plus organisés en un anneau logique (sites numérotés `0`,…,`n-1`) sur lequel circule un jeton.

On désire maintenant fournir une gestion d'un sémaphore à une application répartie. L'implémentation que l'on désire doit être équitable : tout site qui appelle `P()` ne restera pas bloqué indéfiniment. Tous les sites connaissent la valeur initiale du sémaphore $cpt_0$. La couche application n'émet qu'une requête à la fois (`P()` ou `V()` sans paramètre car il n'y a qu'un sémaphore).

On rappelle que pour qu'un `P()` soit passant, il faut que la condition suivante soit réalisée :

`(C)` nombre de `V()` effectués + $cpt_0$ > nombre de `P()` satisfaits

Le principe de la solution est le suivant :

(1) Chaque site tient à jour le membre gauche de `(C)` initialement égal à $cpt_0$

(2) Le jeton contient le membre droit de `(C)`

(3) Un `V()` sur le sémaphore par un site entraine une diffusion de cette information aux autres sites.

(4) Un `P()` sur le sémaphore est bloquant jusqu'à l'arrivée du jeton. Lorsque le jeton arrive sur le site, deux cas sont possibles :
• la condition `(C)` est satisfaite, le jeton est alors relaché après une mise à jour et le `P()` devient passant.
• la condition `(C)` n'est pas satisfaite, le jeton est alors conservé jusqu'à la satisfaction de la condition.

(5) Si le jeton arrive sur un site qui n'est pas en cours de `P()`, le jeton est aussitôt retransmis.

*Attention ! Toutes les actions ne sont décrites ci-dessus.*

**Question 1** Définir les variables de la couche service.

**Question 2** Définir les procédures `P()` , `V()` ainsi que la réception du jeton et du message "V".

**Question 3** Décrire le scénario suivant :
• Il y a quatre sites,
• le sémaphore est initialisé à `0`,
• le site `0` effectue un `V()`,
• puis les sites `1` et `2` effectuent un `P()`,
• enfin le site `3` effectue un `V()`.

**Question 4** Pourquoi cette gestion est-elle équitable ?

**Question 5** Un grand nombre d'algorithmes sont basés sur le principe d'un jeton circulant sur un anneau. Quels sont les avantages et les inconvénients de ce mécanisme ?

***Sujet 5 : Gestion de sections critiques par partage "semi-global" de jetons***

Dans tout ce qui suit, l'ensemble des sites est {1,2,...,n}. On se propose de définir un nouvel algorithme d'accès à une section critique adapté de l'algorithme de Carvahlo et Roucairol à partir des observations suivantes sur cet algorithme :

- Il y a un ensemble de jetons partagés $\{\{i,j\}\}_{i,j \text{ de 1 à n},i\neq j}$ entre l'ensemble des sites.
- Le site i doit posséder le sous-ensemble de jetons $\{\{i,j\}\}_{j \text{ de 1 à n},j\neq i}$ pour entrer en section critique.

L'objectif est de diminuer les tailles de l'ensemble des jetons partagés et des sous-ensembles nécessaires à chaque site pour entrer en section critique. On appellera $R_i$ le sous-ensemble de jetons nécessaires à i pour entrer en section critique.

**Question 1** Quelle condition doivent vérifier les sous-ensembles $R_i$ et $R_j$ pour être assuré que les sites i et j ne rentrent pas simultanément en section critique ?

On décide de partager n jetons numérotés de 1 à n entre l'ensemble des sites. Le site i décide de l'attribution du jeton i en fonction des différentes requêtes qu'il reçoit. On l'appelle le propriétaire du jeton. Pour définir les sous-ensembles $R_i$, on range d'abord les identités des sites dans le tableau carré de taille suffisante. Voici deux exemples de tableau pour n=7 et n=11.

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | | |

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | |
| | | | |

Pour un site i, l'ensemble des jetons $R_i$ qu'il doit obtenir correspond à la ligne et à la colonne où il apparaît. Ainsi dans le premier tableau on a $R_5=\{2,4,5,6\}$ et $R_7=\{1,4,7\}$. Dans le deuxième tableau, on a $R_8=\{4,5,6,7,8\}$ et $R_{11}=\{3,7,9,10,11\}$.

**Question 2** Montrez que la condition de la question 1 est toujours satisfaite. Donnez la taille maximale et la taille minimale d'un ensemble $R_i$ en fonction de n. Indication : les expressions de ces bornes font intervenir les fonctions $\sqrt{x}$ et $\lceil x \rceil$ qui désigne le plus petit entier supérieur ou égal à x.

*Description de l'algorithme*

Initialement, un site i est au repos.

Lorsqu'il désire entrer en section critique, il envoie à tous les sites de $R_i$ (y compris à lui-même) une requête datée de son heure logique courante puis il passe en attente.

Lorsque le site reçoit tous les jetons, il entre en section critique. Pour tester la condition d'entrée, il dispose d'un tableau de booléens indicé par $R_i$.

A la sortie de la section critique, il renvoie les jetons à leurs propriétaires respectifs à l'aide d'un message de libération et repasse au repos.
Quelque soit son état, chaque site maintient l'état du jeton dont il est le propriétaire (présent, prêté, réclamé) – initialement présent - et une file des requêtes qu'il a reçues. La file est triée selon l'âge des requêtes (c'est à dire le couple <heure logique, identité>). Cette file est initialement vide. Lorsqu'un site reçoit une requête, il l'insère dans sa file. Plusieurs cas se présentent alors :

1. Le jeton est présent (la file était vide), il envoie le jeton à l'émetteur de cette requête à l'aide d'un acquittement. Le jeton est prêté.
2. La file n'était pas vide et la requête n'est pas rangée en tête de la file (autrement dit, ce n'est pas la requête la plus ancienne), il ne fait rien de plus.
3. La file n'était pas vide et la requête est rangée en tête de la file. Si le jeton est prêté, il envoie une réclamation à l'émetteur de la seconde requête de la file (c'est à dire celui qui possède actuellement le jeton). Le jeton passe à l'état réclamé. Si le jeton était déjà réclamé, le site ne fait rien.

A la réception d'une réclamation, si un site est en attente et s'il possède le jeton, il renvoie le jeton à son propriétaire par un message de retour. Sinon il ignore la réclamation, car un message de libération a été ou sera envoyé.

A la réception d'un message de libération, le propriétaire extrait la requête de sa file et si la file est non vide, le jeton est prêté à la tête de la file par un acquittement. Sinon le jeton reste présent. A la réception d'un message de retour, le jeton est prêté à la tête de la file par l'envoi d'un acquittement.

**Question 3** Décrivez les variables de chaque site nécessaires à cet algorithme et leur initialisation éventuelle. $R_i$ sera considéré comme une constante calculée par le mécanisme indiqué plus haut.

**Question 4** Ecrivez les primitives suivantes :

- prologue()
- épilogue()
- sur_réception_de(j,(req,h))
- sur_réception_de(j,(acq,h))
- sur_réception_de(j,(lib,h))
- sur_réception_de(j,(reclam,h))
- sur_réception_de(j,(retour,h))

On supposera que la file est dotée de méthodes d'insertion, d'extraction, de recherche de tête et de test de file vide. De plus, l'insertion préserve l'ordre des âges des requêtes.

**Question 5** Lorsqu'un site demande un jeton, cela provoque au pire l'envoi d'une requête, d'une réclamation, d'un retour, de deux acquittements (vers le site demandeur et ultérieurement vers le site auquel on retire le jeton) et d'une libération. En vous servant du majorant de la taille d'un $R_i$ trouvé en question 2, donnez une borne sur le nombre de messages échangés pour entrer en section critique.

## 6 Références

[Car83] O.S.F. Carvalho, G. Roucairol "On mutual exclusion in computer networks" Communication of ACM vol 26,2 février 1983 pp 146-147.

[Lam78] L. Lamport "Time, clocks, and the ordering of events in a distributed system" Communications of the ACM 21,7 juillet 1978 pp 558-565.

[Ric81] G. Ricart, A.K. Agrawala "An optimal algorithm for mutual exclusion in computer networks" Communication of ACM vol 24, janvier 1981 pp. 9-17.

# CHAPITRE V

# L'OBSERVATION

*version du 15 janvier 2004*

## 1 Problématique

Au cours des chapitres précédents, nous avons perçu la nécessité de raisonner sur l'état d'une application répartie. Ainsi une gestion de tampons "optimiste" peut conduire à un interblocage et il est important de le détecter afin d'entreprendre les mesures adéquates. De même, lors de la construction d'un arbre de plus courts chemins, il est indispensable de détecter la terminaison de la construction pour débuter la transaction répartie au dessus de cet arbre.

D'autre part, une manière de rendre tolérante aux fautes une application répartie consiste à construire des points de reprise de telle sorte qu'après une panne, l'application puisse redémarrer dans un état le plus proche possible de celui qu'elle avait au moment de la panne.

Autrement dit, les services proposés dans ce chapitre sont de l'ordre de l'observation. Il s'agit d'obtenir une information sur l'état de l'application **sans perturber, contraindre ou stopper le déroulement de l'application**.

La deuxième section de ce chapitre sera consacrée à la définition d'un état global. En effet, en raison de l'échange asynchrone des messages et des exécutions parallèles sur chaque site, il n'est pas évident de caractériser ce qu'est un état global de l'application. Une fois ce point établi, cette section se terminera par un algorithme de construction d'un état global et son application à la construction de points de reprise.

La troisième section se focalisera sur la détection de propriétés telles que la terminaison ou l'interblocage. Ce type de propriétés a une caractéristique qui facilite sa détection : une fois vérifiée, cette propriété le reste à moins d'une intervention extérieure à l'application. On a affaire à **des propriétés stables**. A partir de la construction d'un état global de la section précédente, on montrera comment obtenir un algorithme de détection de telles propriétés. Cependant cette nouvelle construction n'est que partiellement répartie. Aussi en spécialisant le concept de propriété stable à l'aide de celui **de propriété paisible**, on sera en mesure d'établir un algorithme de construction entièrement réparti qui, de plus, ne nécessite pas la construction d'un état global.

# 2 Etat global d'une application répartie [Cha85]

## *2.1 Exemple introductif*

Supposons que l'application soit composée de trois sites dont le code respectif est le suivant :

| Site 1 | Site 2 | Site 3 |
|---|---|---|
| `e1:Emettre_vers(2,m) ; e5:Recevoir_de(3,x);` | `e3:Recevoir_de(1,y) ;` | `e2:Action_locale; e4:Emettre_vers(1,m') ;` |

Nous adoptons ici comme mode de communication pour l'application : l'émission asynchrone et la réception synchrone. Le mode de fonctionnement du service reste identique. Les primitives `émettre_vers()` et `recevoir_de()` correspondent à ce mode de communication. `x` et `y` sont deux variables qui contiendront des messages.

Afin de cerner le comportement global de l'application, nous faisons l'hypothèse d'un **démon** qui choisirait la prochaine instruction à exécuter en ayant une vision globale de l'application, c'est à dire :

- la prochaine instruction à exécuter pour le processus applicatif de chaque site,
- le contenu de chaque canal de communication (une suite de messages).

Ce démon débute l'application par un premier choix entre `e1` et `e2`. Il ne peut choisir `e3` car initialement les canaux sont vides. S'il choisit `e1`, alors il peut continuer par `e2` ou `e3`, il ne peut choisir `e5` car le canal de `3` vers `1` est vide. S'il choisit `e2`, il peut continuer par `e3` ou `e4` mais il ne peut toujours pas choisir `e5`. Une fois choisi `e3`, le seul choix possible est `e4` suivi de `e5`. La figure 5.1 exhibe l'ensemble des états possibles de l'exécution de l'application sous l'action de ce démon. On a représenté pour chaque état, les instructions exécutées de chaque site suivies de l'état des canaux de `1` vers `2` et de `2` vers `3`.

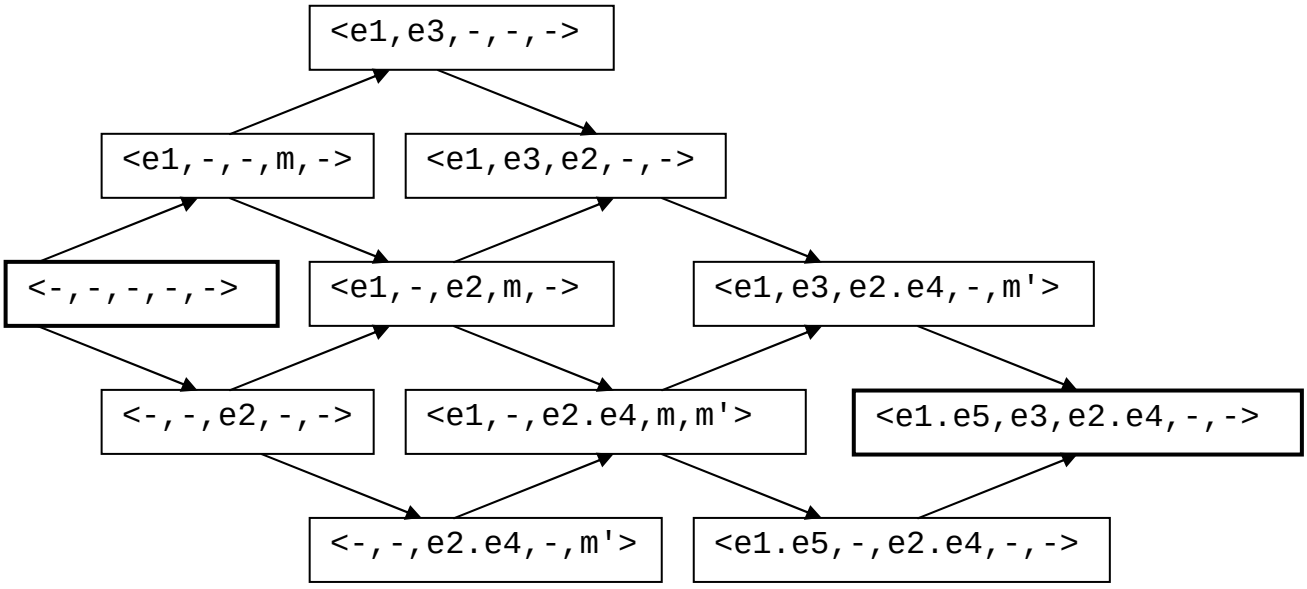



Figure 5.1 : Les états possibles de l'application répartie

Une exécution de l'application peut correspondre à n'importe quel chemin depuis l'état initial jusqu'à l'état final. Cependant, on ne vise pas à retrouver l'un des états rencontrés sur le chemin mais plutôt n'importe quel état de la figure précédente, car l'un quelconque de ses états aurait pu être le résultat de l'exécution de l'application dans des conditions d'exécution différentes (selon les vitesses d'exécution sur chaque site et le délai de transit des messages). Il reste malgré tout à caractériser ce qu'est un état global cohérent.

### *2.2 Définition d'un état global cohérent*

Un état global est donné par un état de l'application sur chaque site i à un instant $t_i$ et un état des canaux. Un état global sera cohérent si et seulement si :

- Tout message émis par le site `j` et reçu par le site `i` avant $t_i$ a été émis avant $t_j$.
- L'état du canal de `i` vers `j` contient - dans l'ordre d'émission – les messages émis par `j` avant $t_j$ et reçus par `i` après $t_i$.

La première condition exprime le fait qu'un message doit être émis pour être reçu et la deuxième condition exprime le fait que les canaux contiennent les messages émis et non encore reçus.

Ainsi `<-,e3,-,-,->` est impossible car un message a été reçu par le site 2 alors qu'il n'a pas été émis par le site 1. De même `<e1,-,-,-,->` est impossible car l'état du canal de `1` vers `2` devrait contenir le message m émis par le site `1` et non encore reçu par le site `2`.
Dans le cadre d'un état global, appelons *arc entrant* un message qui est émis par `i` après l'instant correspondant à son état local et qui est reçu par `j` avant l'instant correspondant à son état local. Appelons de même *arc sortant* un message qui est émis par `i` avant l'instant correspondant à son état local et qui est reçu par `j` après l'instant correspondant à son état local.

Cette dénomination s'explique de la façon suivante. Sur un chronogramme, traçons une ligne qui joint les points correspondant aux états locaux de chaque application. Cette ligne divise le plan en une partie supérieure et une partie inférieure. Le tracé d'un message correspondant à un arc entrant quitte la partie inférieure pour atteindre dans la partie supérieure et le tracé d'un message correspondant à un arc sortant quitte la partie supérieure pour atteindre dans la partie inférieure. Dans ces conditions les deux conditions pour que l'état global soit cohérent peuvent être reformulées ainsi :

- Il n'y a pas d'arc entrant.
- L'état du canal de `j` vers `i` contient - dans l'ordre d'émission – les arcs sortants correspondant aux messages émis par `j` à destination de `i`.

### *2.3 Construction d'un état global cohérent*

D'après la définition d'un état global, l'application doit fournir une primitive qui permet d'enregistrer son état local ou plus précisément l'abstraction qui sera suffisante pour ce à quoi servira l'état global, une fois construit. Nous reviendrons sur ce point lors de la section suivante. Nous appellerons `enregistrer()` la primitive qui renvoie l'état local de l'application.

Pour ce qui est de l'état d'un canal, il y a deux possibilités : soit stocker l'état d'un canal sur le récepteur, soit sur l'émetteur. Dans les deux cas, le site choisi ne dispose que d'une vue partielle sur le canal. Cependant à mesure que le temps s'écoule, le récepteur pourra "compléter" son information sur le passé du canal ce qui ne sera jamais le cas de l'émetteur. Nous choisissons de stocker l'état du canal sur le récepteur.

Enfin bien que nous visions un algorithme réparti nous supposerons qu'un site appelé l'initiateur démarre la construction de l'état global.

Décrivons informellement une première construction fautive qui nous conduira à la solution correcte. Dans cette construction, l'initiateur enregistre son état, puis diffuse un message de construction (`cons`) aux autres sites. Chacun d'entre eux à la réception de ce message enregistre à son tour son état local (nous oublions pour l'instant l'état des canaux).

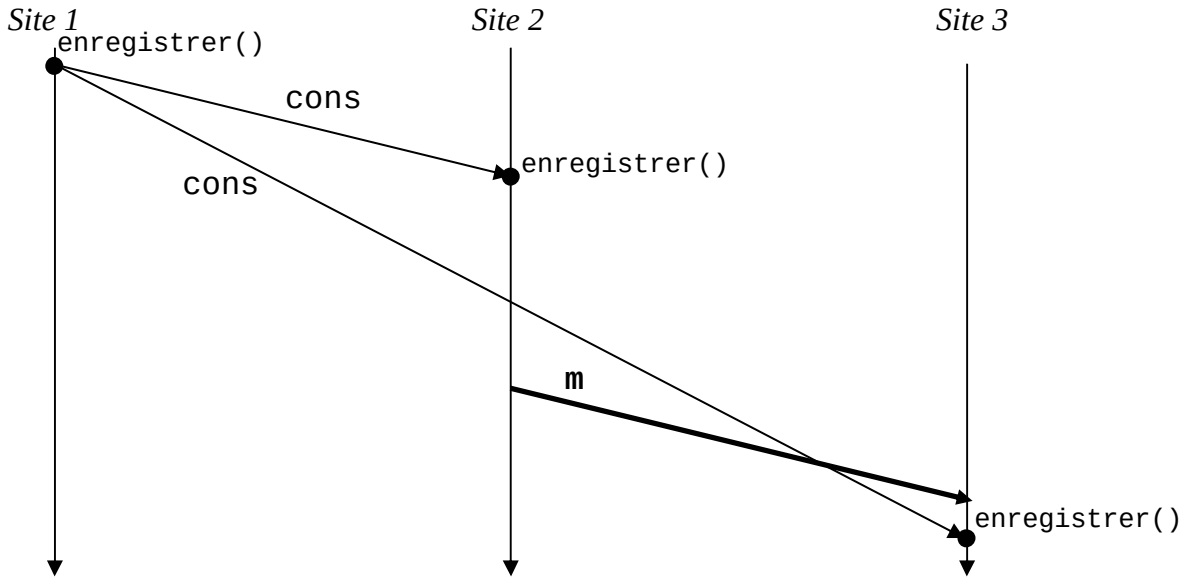



Figure 5.2 :Une construction fautive

Sur la figure 5.2, le message d'application `m` est émis après l'enregistrement de l'état sur le site `2` et reçu avant l'enregistrement de l'état sur le site `3`. Quelque soit l'état des canaux, l'état global construit ne sera donc pas cohérent (il viole la première condition).

Pour remédier à ce problème, chaque site collabore à la construction en diffusant un message de construction lorsqu'il enregistre son état et il enregistre celui-ci à la première réception d'un message de construction (qui n'est plus nécessairement celui provenant de l'initiateur).

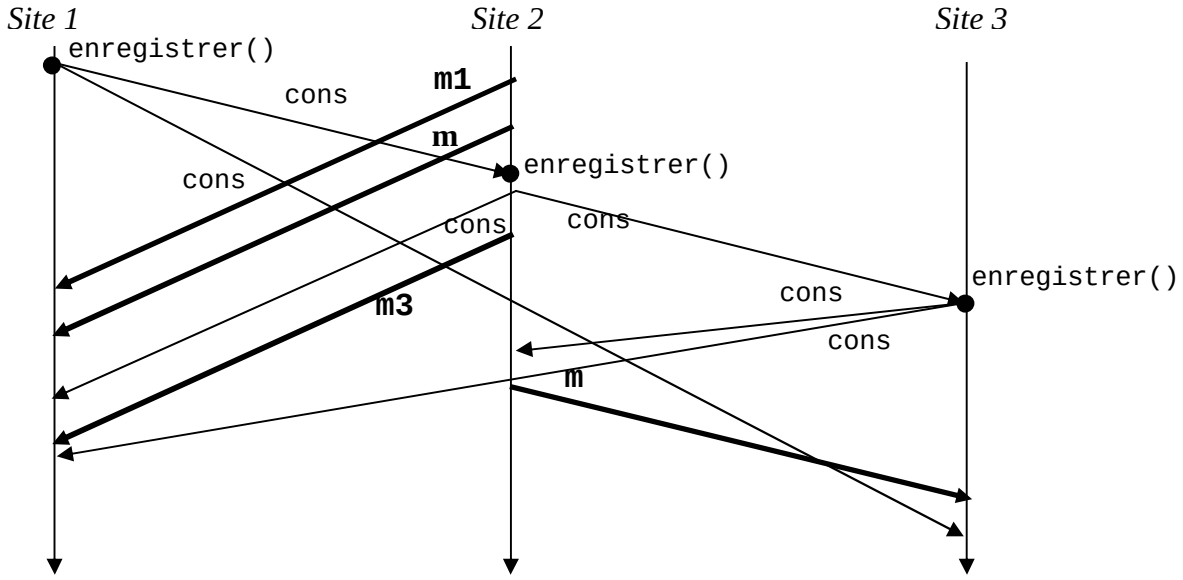



Figure 5.3 :Une deuxième construction

On voit sur la figure 5.3 que le message `m` est maintenant reçu après l'enregistrement de l'état sur le site 3. Généralisons : tout message émis après l'enregistrement de l'état sur le site émetteur est émis après l'émission du message de construction correspondant. Puisque les canaux sont fifo, il est reçu après ce même message de construction, donc par conséquent après l'enregistrement de l'état sur le récepteur. La première contrainte de l'état global est donc vérifiée.

Intéressons-nous maintenant à construire l'état des canaux et pour cela reportons-nous à la figure 5.3 et plus particulièrement au canal de `2` vers `1`. Le site 1 doit reconstituer ce canal comme étant la suite des messages reçus après l'enregistrement de l'état sur `1` et avant l'enregistrement de l'état sur `2`. Après l'enregistrement de l'état, le site `1` voit "passer" les messages `m1`, `m2` et `m3`. Puisque le canal est fifo, tant que le site 1 ne voit pas passer le message de construction de 2, il conclut que les messages ont été émis avant l'enregistrement de l'état sur `2`. Autrement dit, `m1` suivi de `m2` sera exactement l'état du canal.

Une fois reçus tous les messages de construction, un site sait qu'il a terminé sa construction.

Il reste maintenant à synthétiser ces remarques pour obtenir l'algorithme. L'interface application-service est constituée des primitives suivantes :

- `enregistrer()` appelée par le service pour enregistrer l'état courant de l'application.
- `délivrer_de(j,m)` appelée par le service pour délivrer un message d'application `m` provenant de `j`.
- `construire()` appelée sur l'initiateur par l'application (ou une autre couche) pour construire un état global de l'application.
- `émettre_vers(j,m)` appelée par l'application pour émettre un message `m` vers l'application de `j`.

Variables du site `i`

- $s_i$ : variable contenant l'état local de l'application du site i.
- $canal_i[1..N]$ : tableau de suites de messages. Chaque cellule `j` de ce tableau (exceptée la cellule `i`) correspond à l'état du canal de `j` vers `i` dans l'état construit.
- $état_i$ : (`avant`,`pendant`,`après`) état local du service initialisé à `avant`.
- $àenregistrer_i[1..N]$ : tableau de booléens. Indique si un message venant d'un canal doit être enregistré pour compléter l'état global. (`N` est le nombre de sites)
- $nbcons_i$ : nombre de messages de construction reçus.
- $temp_i$ : variable temporaire

Algorithme du site i

La construction consiste à initialiser les variables et diffuser un message de construction aux autres sites.

```
construire()
Début
     s_i = enregistrer();
     état_i = pendant;
     nbcons_i=0;
     Pour temp_i de 1 à N faire
          canal_i[temp_i]= ∅;
          àenregistrer_i[temp_i]= vrai;
     Fin pour
     Diffuser(cons);
Fin
```

L'émission d'un message d'application est accompagnée d'une encapsulation.

```
émettre_vers(j,m)
Début
     envoyer_à(j,(app,m));
Fin
```

A la réception d'un message de construction, on débute éventuellement la construction puis on incrémente le nombre de messages de construction reçus et on "clôt" l'enregistrement du canal correspondant. A la réception du dernier message de construction, la construction est terminée.

```
sur_réception_de(j,cons)
Début
     Si état_i == avant Alors
          construire();
     Fin si
     nbcons_i++;
     àenregistrer_i[j]= faux;
     Si nbcons_i == N-1 Alors
          état_i=après;
     Fin si
Fin
```

La réception d'un message d'application provoque sa délivrance éventuellement accompagnée d'un enregistrement dans le canal correspondant.

```
sur_réception_de(j,(app,m))
Début
     Si àenregistrer_i[j] == vrai Alors
          canal_i[j]=canal_i[j]& m; // concaténation du message
     Fin si
     délivrer_de(j,m);
Fin
```

### *2.4 Construction de points de reprise*

Une application importante de la construction de l'état global est la construction de points de reprise. Cette construction a pour objectif de redémarrer une application répartie sujette aux fautes dans un état global le plus proche possible de la panne éventuelle.

On pourrait dans un premier temps, construire périodiquement un état global et à la suite d'une panne redémarrer dans l'état construit. On voit ici un autre avantage à stocker les messages sur le récepteur puisqu'au redémarrage, le service les délivre directement à son application (pas de circulation sur le réseau). Cependant que faire si la panne survient en cours de construction de l'état global ?

Il nous faut donc gérer un historique de deux états globaux. Pour les distinguer, le service de chaque site affecte un numéro (incrémenté) à chaque état global qui est stocké avec l'état. De plus, il faut savoir si la construction de la partie de l'état a pu être menée à bien avant la panne et pour cela on ajoutera une deuxième variable ($\texttt{validé}_i$) à chaque partie locale de l'état global. En début de construction, $\texttt{validé}_i$ est positionnée à faux ; elle passe à `vrai` lorsque la variable $\texttt{état}_i$ passe à `après`. On suppose qu'au départ, on dispose de l'état global correspondant à l'état initial de l'application.

En conséquence, chaque site dispose à tout instant d'au moins un état global cohérent (et éventuellement deux). Supposons qu'au redémarrage le site, `i` ait le numéro (`k`) d'état global validé le plus grand et soit `j` un autre site qui n'ait pas validé la partie de cet état. Puisque `i` a construit sa partie d'état global `k`, le site `j` a du lui envoyer un message de construction correspondant à cet état. Par conséquent, `j` a construit sa partie d'état global `k-1`. Puisque j n'a pas achevé la construction `k`, nécessairement i n'a pu débuter la construction k+1 et par conséquent, `i` dispose encore de l'état global `k-1`.

Nous venons donc de démontrer que :
- soit tous les sites ont le même état global validé de plus grand indice,
- soit les sites les plus avancés ont un état global d'avance sur les autres et ont conservé leur état précédent.

Le mécanisme de redémarrage consiste alors pour chaque site à diffuser son plus grand indice d'état global validé. Puis, à la réception de tous les messages, chaque site redémarre à partir de l'état correspondant au minimum de ces indices.

# 3 Détection de propriétés stables et paisibles [Cha86]

## *3.1 Propriétés stables*

Une propriété stable évolue de la manière suivante :
- Elle est initialement fausse.
- Soit elle reste indéfiniment fausse, soit elle devient vrai et le reste indéfiniment.

La figure 5.4 décrit ces deux comportements. Nous appelons $\pi$ la propriété stable et pour marquer la dépendance de la propriété en fonction de l'exécution et du temps, nous notons $\pi(t)$ la valeur de $\pi$ à l'instant $t$ et $\pi(s)$ la valeur de $\pi$ dans l'état $s$.

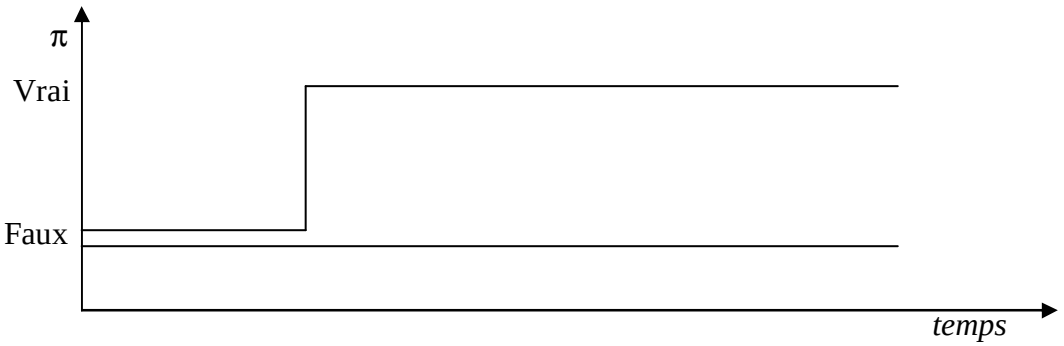



Figure 5.4 : Les deux comportements d'une propriété stable

Afin de mieux cerner ce que recouvre ce concept, nous allons l'illustrer par trois exemples.

1. Les sites de l'application répartie se partagent des ressources à accès exclusif. Un site qui demande une ressource en cours d'utilisation attend sa libération. Si des sites s'attendent mutuellement, ceci conduit à un **interblocage de ressources**. Veuillez noter que ce type de problème n'est pas lié à la répartition mais plutôt au parallélisme. Pour reconnaître cet interblocage, une abstraction de l'état local de chaque site est suffisante : les ressources possédées par le site et l'éventuelle ressource qu'attend le site pour reprendre son exécution. A partir de cette abstraction, on construit un graphe biparti dont les nœuds sont les sites et les ressources, un arc d'un site vers une ressource signifie que le site attend la ressource et un arc d'une ressource vers un site signifie que la ressource est utilisée par le site. L'interblocage est alors équivalent à l'existence d'un circuit dans ce graphe.
2. Les sites de l'application répartie communiquent par réception synchrone. Ainsi des sites peuvent attendre mutuellement des messages alors que les canaux sont vides. On a alors affaire à un **interblocage de communication**. Pour reconnaître cet interblocage, une abstraction de l'état local de chaque site est suffisante : l'éventuel site distant dont le site attend un message pour reprendre son exécution. A partir de cette abstraction, on construit un graphe dont les nœuds sont les sites, un arc d'un site vers un autre site signifie que le premier site attend un message du second et que le canal correspondant est vide. L'interblocage est alors équivalent à l'existence d'un circuit dans ce graphe.

3. Une application répartie peut comporter plusieurs phases. Il est alors important de détecter lorsqu'une phase est terminée pour débuter la suivante. Ce problème (déjà soulevé au chapitre 3) s'appelle la **terminaison**. Pour reconnaître la terminaison, une abstraction de l'état local de chaque site est suffisante : est-il actif ou passif vis à de la phase courante. La terminaison est alors équivalente à ce que tous les sites soient passifs et qu'aucun message ne circule sur les canaux qui relient les sites de l'application.

### *3.2 Un algorithme de détection de propriétés stables*

Nous allons tout d'abord exprimer les objectifs d'un algorithme de détection de propriétés stables. Cet algorithme peut renvoyer à la demande une évaluation $\pi^*$ de la valeur de $\pi$. Cette évaluation doit satisfaire deux exigences :

- Elle ne doit pas conduire à de fausses détections. Si $\pi$ est fausse à l'instant $t$ et l'évaluation est demandée à cet instant, alors $\pi^*$ est fausse.
- Elle doit finir par détecter l'occurrence de la propriété. Si $\pi$ devient vraie à l'instant $t$ alors il existe $t_1 \geq t$ telle qu'après $t_1$, $\pi^*$ est toujours vraie.

Montrons comment adapter l'algorithme de construction d'état global pour détecter une propriété stable. Tout d'abord, nous allons spécialiser un site $i_0$ qui se chargera de l'évaluation de la propriété.

Le site $i_0$ lance périodiquement une construction d'état global. Une fois celle-ci terminée, les sites transfèrent leur partie d'état global vers $i_0$. Appelons s* l'état global construit, calcule $\pi$(s*) pour mettre à jour $\pi^*$. Si $\pi^*$ devient vrai, il n'est plus nécessaire de continuer à construire l'état global.

Algorithme

```
Début
     Répeter périodiquement
          (1)  io initie une construction d'état global s*
          (2)  les sites transfèrent leur partie de s* vers i0
          (3)  io calcule π(s*) et π*=π(s*)
     Jusqu'à π*==vrai;
Fin
```

Prouvons que cet algorithme répond aux deux exigences formulées plus haut. On note tout d'abord $s_{deb}$ l'état au début de la construction de $s^*$ et $s_{fin}$ l'état à la fin de la construction. Rappelons que même si $s^*$ n'est pas un état rencontré, il est un futur possible de $s_{deb}$ et un passé possible de $s_{fin}$.

Supposons qu'à un instant $t$ où $\pi(t)$ est fausse, on examine $\pi^*$. $s_{fin}$ est un état qui précède l'état courant et par conséquent $s^*$ est un passé possible de l'état courant. Puisque $\pi$ est stable, nécessairement $\pi(s^*)$ est fausse.

Examinons maintenant un instant $t$ où $\pi(t)$ devient vraie ; s'il n'y a plus de construction après $t$ ceci signifie que $\pi^*$ est vrai. Dans le cas contraire, $s_{deb}$ est un état qui suit l'état atteint en $t$ et par conséquent $s^*$ est un futur possible de cet état. Puisque $\pi$ est stable, nécessairement $\pi(s^*)$ est vraie.

### *3.3 Un algorithme de détection de propriétés paisibles*

L'algorithme précédent n'est pas entièrement satisfaisant du point de vue de la répartition puisque l'information est centralisée vers le site qui évalue la propriété stable. Ceci entraîne une augmentation du trafic du réseau vers ce site et une charge de calcul supplémentaire sur ce site. Pour obtenir un algorithme entièrement réparti (i.e. où tous les sites jouent le même rôle à l'exception de l'initiation) nous devons posséder plus d'information sur la propriété $\pi$. Ceci nous conduit à la définition suivante.

Définition Une propriété $\pi$ est dite paisible si et seulement si :

1. Il existe un ensemble de sites I*
2. Il existe un ensemble de canaux C* reliant des sites de I*
3. Il existe une propriété $b_i$ dépendant uniquement de l'état local de $i$ et l'une (au moins) des $b_i$ est initialement fausse.
4. $\pi(s) =_{def} Et_{i \in I^*} b_i(s_i)\ Et_{c \in C^*} c(s) == \emptyset$
5. Si $b_i$ est vrai, alors seule la réception par $i$ d'un message d'un canal de C* positionne $b_i$ à faux.
6. Si $b_i$ est vrai, alors $i$ ne peut envoyer de message sur un canal de C*.

Montrons immédiatement qu'une propriété paisible est stable. $\pi$ est initialement fausse puisque l'un des $b_i$ est initialement fausse. Supposons maintenant que $\pi(s)$soit vraie, qu'un événement $e$ conduise de $s$ en $s'$ et que $\pi(s')$soit fausse.

Alors soit il existe un $i$ tel que $b_i(s')$ devienne fausse soit il existe un $c$ tel que $c(s')$ ne soit plus vide. Dans le premier cas en vertu du point 5, e doit être une réception d'un message sur un canal de C* ce qui est impossible puisque tous les canaux sont vides. Dans le deuxième cas, e doit être une émission d'un message sur un canal de C* ce qui est impossible en vertu du point 6.

Les propriétés paisibles constituent ainsi une spécialisation des propriétés stables. Heureusement cette catégorie recouvre les propriétés paisibles les plus usuelles. Ainsi la terminaison est une propriété paisible ; il suffit pour cela de prendre pour I* l'ensemble des sites, C* l'ensemble des canaux et de considérer que $b_i$ est vraie si et seulement si $i$ est passif. On vérifiera facilement que les points 4, 5 et 6 sont alors vérifiés. De même, supposons que l'on veuille détecter l'existence d'un circuit d'interblocage de communication. On prendra alors pour I* l'ensemble des sites de ce circuit, C* l'ensemble des canaux sous-jacents à ce circuit et $b_i$ vrai si et seulement si le site $i$ attend un message du site qui le suit sur le circuit. On vérifiera encore que les points 4, 5 et 6 sont vérifiés.

Nous décrivons maintenant le principe d'un algorithme de détection de propriétés paisibles. Ici encore, il s'agit d'une estimation périodique jusqu'à ce que l'estimation devienne vraie. Chaque site $i$ observe durant un intervalle $[deb_i..fin_i]$ (à préciser ultérieurement) la propriété $b_i$. Si elle vraie durant tout l'intervalle alors le site positionne un estimateur local

$b^*_i$ à vrai. L'estimation globale $\pi^*$ est la conjonction des estimations locales. On appellera le début de l'observation $deb =_{def} \min(deb_i)$ et la fin de celle-ci $fin =_{def} \max(fin_i)$.

Sans plus de précisions, cet algorithme satisfait l'une des exigences d'une détection. En effet supposons qu'au début d'une observation la propriété $\pi$ soit vraie. Elle le demeure indéfiniment puisque elle est stable. En conséquence chaque site i durant son intervalle observe $b_i$ à vrai. Par conséquent $\pi^*$ sera vraie.

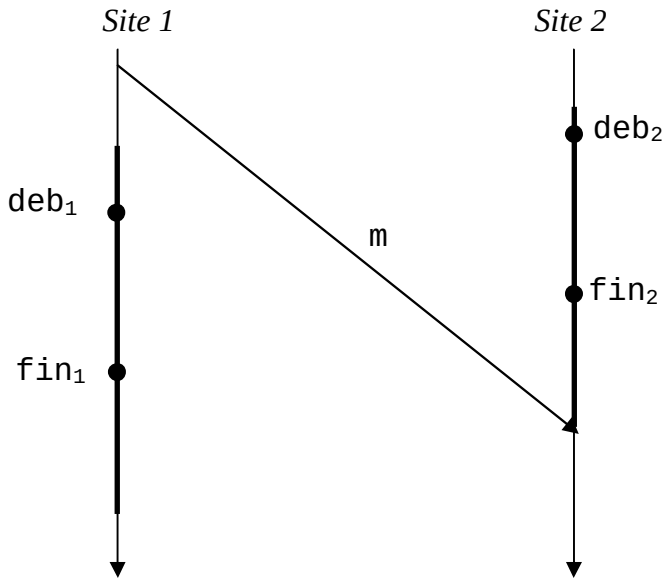



Figure 5.5 : Une observation incorrecte

La deuxième exigence n'est pas satisfaite comme le montre le contre-exemple de la figure 5.5. Sur cette figure nous avons représenté en gras les instants où la variable locale $b_i$ est vraie. L'estimation $\pi^*$ est vraie. Pourtant à la fin de la construction, $\pi^*$ est faux car le message m est présent dans le canal (d'ailleurs $b_2$ repassera plus tard à faux). Le problème provient du fait que m a été émis avant le début de l'observation sur le site 1 et reçu après la fin de l'observation sur le site 2.

Nous allons donc contraindre (par un mécanisme à préciser ultérieurement) les intervalles de telle sorte que tout message émis par i vers j avant $deb_i$ est reçu avant $fin_j$. Il se trouve que cette contrainte suffit à rendre cet algorithme correct. Supposons que cela ne soit pas le cas. Autrement dit l'estimation $\pi^*$ est vraie et la propriété $\pi$(fin) est fausse.

Puisque $\pi$(fin) est fausse,

- soit il existe i tel que $b_i(fin)$ soit fausse
- soit il existe un message m à destination de i à l'instant fin. Lorsque ce message arrivera alors $b_i$ sera vraie.

Dans les deux cas de figure, il existe i, il existe $t \geq fin_i$ tel que $b_i(t)$ soit fausse. Posons $t_0$ le premier instant pour lequel il existe i avec $fin_i \leq t_0$ et $b_i(t_0)$ fausse. Avant $t_0$, $b_i$ était fausse donc i a reçu un message d'un site j à cet instant. D'après la contrainte sur les intervalles, ce message n'a pu être émis avant $deb_j$, d'après le résultat de l'estimation il n'a pu

être émis durant l'intervalle $[deb_j..fin_j]$, donc il a été émis en un instant $t_1 \geq fin_j$. Evidemment $t_1$ est inférieur à $t_0$. D'autre part en vertu de la définition d'une propriété paisible, $b_j(t_1)$ est fausse. Mais l'existence de $t_1$ contredit la définition de $t_0$. Ce qui achève la démonstration.

Il reste maintenant à assurer la contrainte sur les intervalles. Nous supposons comme précédemment qu'un initiateur lance l'observation en la débutant localement et en diffusant un message d'observation (`obs`). A la réception du premier message `obs`, tout autre site démarre son observation et diffuse à son tour ce même message.

Chaque site termine son observation lorsqu'il a reçu un message d'observation de chacun des autres sites. En, effet tout message émis avant la période d'observation précéde la diffusion du message `obs` et sera reçu avant celui-ci (hypothèse des canaux fifo). Ce qui garantit la contrainte.

A la fin de l'observation tout site diffuse un message de fin accompagné de son estimation locale. La fin de l'algorithme est détecté par chaque site à réception d'un message de fin de chacun des autres sites. Nous formalisons maintenant l'algorithme.

L'interface application-service est constituée des primitives suivantes :

- `set_bi()` appelée par l'application pour indiquer que la propriété locale $b_i$ devient vraie.
- `délivrer_de(j,m)` appelée par le service pour délivrer un message d'application `m` provenant de `j`.
- `observer()` appelée sur l'initiateur par l'application (ou une autre couche) pour débuter une observation.
- `émettre_vers(j,m)` appelée par l'application pour émettre un message `m` vers l'application de `j`.

Variables du site `i`

- $b_i$ : état de l'application du site i. Initialisée selon le fonctionnement de l'application
- $b^*_i$ : état "cumulé" de l'application durant la période d'observation.
- $PI_i$ : estimation de la propriété
- $nbobs_i$ : nombre de messages d'observation reçus.
- $nbfin_i$ : nombre de messages de fin reçus.
- $état_i$ : (`avant`,`pendant`,`après`) état local du service initialisé à `avant`.
- $temp_i$ : variable temporaire.

Algorithme du site $i$

Le début de l'observation consiste à initialiser les variables et diffuser un message d'observation aux autres sites.

```
observer()
Début
     b*_i = b_i;
     PI_i = vrai;
     nbobs_i = 0;
     nbfin_i = 0;
     état_i = pendant;
     Diffuser(obs);
Fin
```

Cette primitive a pour unique rôle de positionner à vrai la variable $b_i$.

```
set_bi()
Début
     b_i=vrai;
Fin
```

L'émission d'un message d'application est accompagnée d'une encapsulation.

```
émettre_vers(j,m)
Début
     envoyer_à(j,(app,m));
Fin
```

A la réception d'un message d'observation, on débute éventuellement l'observation puis on incrémente le nombre de messages d'observation reçus. A la réception du dernier message d'observation, l'observation est terminée. On met à jour l'estimation de la propriété globale avec l'estimateur local et on diffuse son estimateur global à l'aide d'un message de `fin`.

```
sur_réception_de(j,obs)
Début
     Si état_i == avant Alors
          observer();
     Fin si
     nbobs_i++;
     Si nbobs_i==N-1 Alors
          PI_i = PI_i ET b*_i;
          diffuser(fin,b*_i);
     Fin si
Fin
```

A la réception d'un message de fin, on met à jour l'estimation de la propriété globale avec la valeur reçue. Si ce message est le dernier alors l'estimation est définitive.

```
sur_réception_de(j,(fin,v))
Début
     PIi = PIi ET v;
     nbfini++;
     Si nbfini==N-1 Alors
          étati=après;
     Fin si
Fin
```

La réception d'un message d'application provoque sa délivrance. Les deux variables (instantanée et cumulée) de l'état de l'application sont remises à faux.

```
sur_réception_de(j,(app,m))
Début
     bi=faux;
     b*i=faux;
     délivrer_de(j,m);
Fin
```

### *3.4 Cas des canaux non fifo*

Nous allons maintenant modifier l'algorithme précédent dans le cas où les canaux ne sont plus fifo. Nous procéderons de manière informelle en laissant au lecteur le soin de développer l'algorithme correspondant.

Nous cherchons à contraindre les intervalles de telle sorte qu'un message émis avant le début de l'observation sur l'émetteur soit reçu avant la fin de l'observation chez le récepteur. Nous élaborons la solution en plusieurs étapes.

Tout d'abord, chaque site envoie un acquittement à l'émetteur. Cet acquittement a pour objectif d'informer le service de l'émetteur que le message d'application n'est plus dans le réseau. Afin d'exploiter cette information, chaque site gère un compteur de messages émis qu'il incrémente à chaque émission et qu'il décrémente à chaque réception d'acquittement. Ainsi ce compteur est une borne supérieure du nombre de messages en transit émis par le site.

Un site ne débute son observation que si ce compteur est nul. Soit un message émis avant le début de l'observation, il est donc reçu avant le début de l'observation (sur l'émetteur) donc aussi avant la fin de l'observation sur le récepteur. En effet, le début de l'observation sur un site précède toujours la fin de l'observation sur un autre en raison de la gestion des messages d'observation.

Il reste à éclaircir un dernier point. Telle que nous l'avons présenté, il se peut qu'une observation ne se termine jamais si le compteur d'un site ne repasse jamais à zéro. Mais dans ce cas, cela signifie qu'un site envoie indéfiniment des messages et par conséquent que la propriété paisible est indéfiniment fausse. La réponse de l'estimateur (faux) sera donc correcte.

# 5 Exercices

***Sujet 1***
On se propose d'adapter l'algorithme de construction de Chandy et Misra d'un état global cohérent dans le cas où les canaux ne sont pas FIFO. On rappelle qu'un état global est cohérent si étant donné un enregistrement des états locaux sur chaque site, on a les deux propriétés suivantes :

- Il n'y a pas d'arc entrant, c'est à dire pas de message émis après l'enregistrement de l'état par l'émetteur et reçu avant l'enregistrement de l'état par le récepteur.
- L'état des canaux est constitué des arcs sortants , c'est à dire des messages émis avant l'enregistrement de l'état par l'émetteur et reçus après l'enregistrement de l'état par le récepteur.

**Question 1** Montrer que même pour deux sites, aucune de ces propriétés n'est satisfaite par l'algorithme de Chandy et Misra si les canaux ne sont pas FIFO. Vous pourrez donner un schéma de déroulement pour chaque contre-exemple.

*Adaptation n°1*
Pour remédier à ces problèmes, on décide de "colorer" chaque site et chaque message d'application :

- Un site est vert s'il n'a pas débuté sa construction, rouge dans le cas contraire.
- Un message prend la couleur de l'émetteur, au moment de son émission.

**Question 2** Caractériser en termes de couleur d'un message et du site récepteur ce qu'est un arc entrant et un arc sortant.

*Adaptation n°2*
D'après cette caractérisation, il est facile à un site récepteur de reconnaître un arc qui pourrait être entrant. Aussi un site autre que l'initiateur commence sa construction lors de la première réception d'un message qui soit :

- un message de construction,
- ou un message d'application de couleur rouge

*Adaptation n°3*
D'après cette même caractérisation, il est facile à un site qui a commencé sa construction de reconnaître un arc sortant. Aussi chaque site en cours de construction enregistrera les messages verts indépendamment des réceptions de message de construction.

Puisque les canaux ne sont pas FIFO, veuillez noter que l'état de chaque canal n'est plus une suite ordonnée de messages en transit mais un ensemble.

Il reste cependant un problème à résoudre à savoir la terminaison de l'algorithme puisque des messages verts peuvent arriver après la réception du dernier message de construction.

*Adaptation n°4*
Aussi, chaque site maintient un compteur (à valeurs positives ou négatives) initialisé à `0`.

- Ce compteur est incrémenté à chaque émission de message vert et décrémenté à chaque réception de message vert.
- Lors du début de la construction, le site attache au message de construction la valeur de son compteur puis en cours de construction, à chaque réception d'un message vert diffuse un message de décrémentation.
- A la réception du message de construction, le site ajoute à son compteur, la valeur transmise.
- A la réception d'un message de décrémentation, le site décrémente son compteur.

Une fois tous les messages de construction reçus, le compteur d'un site représente alors une borne supérieure sur les messages verts en transit. La condition de terminaison s'exprime donc comme suit : "Réception de tous les messages de construction et compteur nul".

**Question 3** Décrivez les variables de l'algorithme et leur initialisation.

**Question 4** Écrivez les primitives de l'algorithme :

- `construire()` // correspondant au début de la construction
- `émettre_vers(j,m)` // appelée par l'application pour envoyer un message
- `sur_réception_de(j,(app,couleur,m))`
- `sur_réception_de(j,(cons,valeur))`
- `sur_réception_de(j,(décrémentation))`

Le service dispose de la primitive `délivrer_de(j,m)` pour transmettre les messages à son application.

**Question 5** La diffusion des messages de décrémentation est coûteuse. Proposez de manière informelle un mécanisme vu en cours qui permet de diminuer les messages de service envoyés sur le réseau. Quelle variable faut-il alors ajouter à l'algorithme ?
Dans tout ce qui suit, l'ensemble des sites est `{1,2,...,n}`.

***Sujet 2***

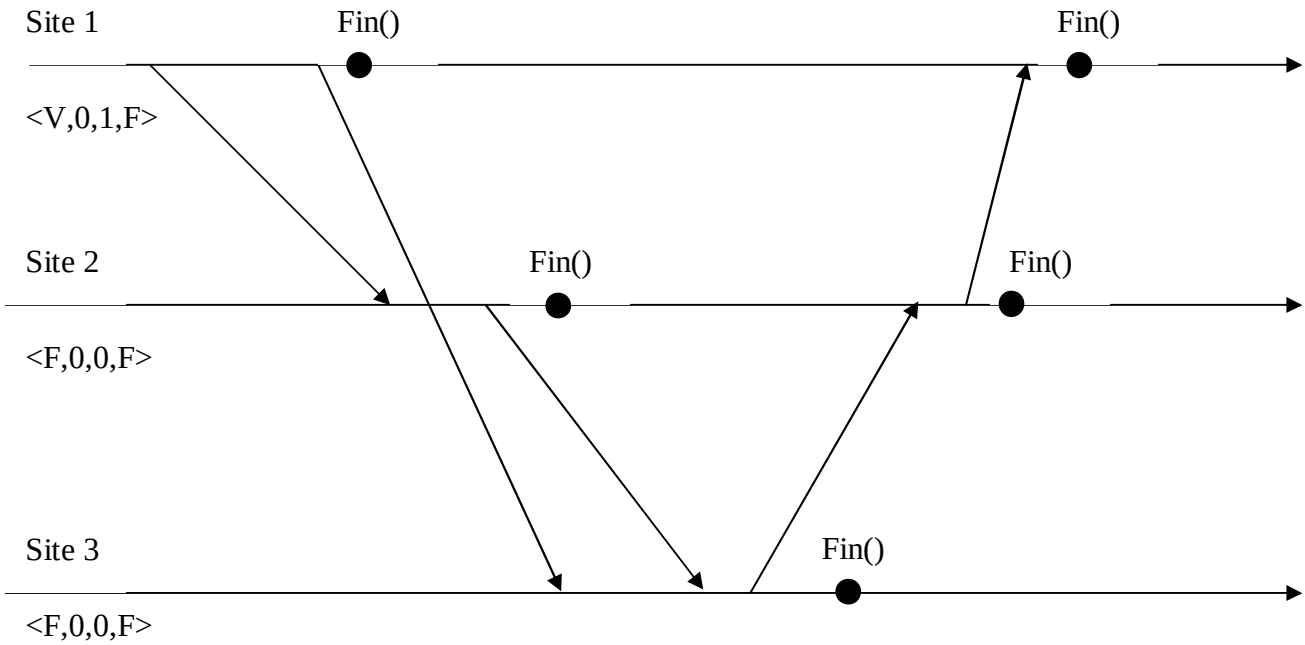
**Question 1** Indiquez en fonction de `n` le nombre de messages de service envoyés lors d'une observation associée à la détection d'une propriété paisible.

On remarque que ce nombre est indépendant de l'activité d'une application. Aussi, on décide de concevoir un algorithme plus efficace de détection de la terminaison d'une application.

On fait l'hypothèse que l'application est démarrée par un site particulier $i_0$, dit l'initiateur.

- L'activité d'un site est traduite par une variable booléenne $actif_i$ initialement vraie pour $i=i_0$ et fausse pour les autres sites.
- Le service de chaque site maintient un compteur $cpt_i$ de messages d'application envoyés dont il n'a pas reçu d'acquittement et une variable $père_i$ initialisée à $i_0$ pour l'initiateur et à `0` pour les autres sites.
- A la réception d'un message d'application (qui sera délivré) si l'application était passive celle-ci s'active
    - et si $père_i$ est nul, le service adopte comme père l'émetteur du message d'application à l'aide de sa variable $père_i$
    - sinon le service envoie un acquittement à l'émetteur.
- A la réception d'un acquittement, le service décrémente le compteur.
- Lorsque une application a terminé son travail courant, elle appelle la primitive de service `Fin()`.
- Dans les deux cas précédents, si le compteur est nul et si l'application est passive alors
    - soit le site est l'initiateur, le service détecte la terminaison et met à jour la variable $terminé_i$ (initialisée à faux) et diffuse un message de terminaison,
    - soit le site n'est pas l'initiateur, il remet à zéro sa variable $père_i$ et il envoie un acquittement à son père.
- A la réception d'un message de terminaison, le service met à jour la variable $terminé_i$.

**Question 2** Recopiez le scénario ci-dessous et complétez-le avec l'état des différentes variables et les messages de service envoyés. Sur la figure sont représentés les échanges de messages d'application et les appels à la primitive `Fin()`. L'initiateur est le site 1.




Les variables d'un site sont représentées par un quadruplet :
$\texttt{<actif}_i\texttt{,cpt}_i\texttt{,père}_i\texttt{,terminé}_i\texttt{>}$.

**Question 3** Ecrivez les primitives suivantes :

- `émettre_vers(j,m)`
- `Fin()`
- `sur_réception_de(j,(app,m))`
- `sur_réception_de(j,acq)`
- `sur_réception_de(j,term)`

Vous utiliserez la primitive `délivrer_de(j,m)`.

**Question 4** Supposez que l'application envoie `m` messages avant de se terminer. Exprimez le nombre de messages de service envoyés en fonction de `m` et de `n`.

**Question 5** Indiquez les adaptations à effectuer sur l'algorithme dans le cas où plusieurs sites peuvent démarrer simultanément l'application ?

## 6 Références

[Cha85] K.M. Chandy, L. Lamport "Distributed snapshots : determining global states of distributed systems" ACM TOCS, vol. 3,1 (1985) pp 63-75.

[Cha86] K.M. Chandy, J. Misra "An example of stepwise refinment of distributed program : quiescence detection" ACM Toplas vol. 8,3 (1986) pp 326-343.

# CHAPITRE VI

## L'ELECTION

*version du 16 janvier 2004*

## 1 Problématique

Il arrive que même si les stations d'une application répartie ont des fonctionnalités similaires, certaines tâches spécifiques doivent être réalisées par une station unique. Aussi avant de démarrer l'application, les sites doivent se mettre d'accord sur l'identité de cette station. Cette opération est communément appelée élection et fait l'objet du présent chapitre.

Donnons deux exemples de l'intérêts de l'élection :

- Dans une application de type maître-esclaves, tous les serveurs sont redondants pour le traitement des requêtes de lecture des clients. Cependant les requêtes d'écriture ne sont traitées que par un seul serveur (le maître) qui transmet ensuite les modifications aux autres sites (les esclaves). On pourra, par exemple, imaginer une gestion de mots de passe qui permet d'ouvrir une session sans passer par le réseau.
- Certaines applications nécessitent une phase d'initialisation exécutée par un seul site. L'exemple le plus simple est celui de la circulation d'un jeton unique entre les sites. Dans ce cas l'initiateur est le premier possesseur du jeton.

Il est naturellement possible de fixer l'identité de ce site dans le code ou mieux dans un fichier de configuration. Cependant une telle solution présente des inconvénients. En cas de changement de configuration du réseau local, l'administrateur doit modifier le (ou les) fichier(s) de configuration. Un arrêt temporaire mais durable de la machine choisie nécessite une intervention manuelle.

Aussi, nous présentons ici des solutions qui sont dynamiques au sens où l'élection a lieu uniquement lorsqu'un site de l'application a besoin d'utiliser cette identité. Autrement dit, le service que nous réaliserons a une interface constituée d'une seule fonction `leader()` qui renvoie l'identité du site élu. La contrainte à vérifier est que l'identité renvoyée soit toujours la même quelque soit l'instant ou le lieu de l'appel.

La suite du chapitre est organisée ainsi. Nous développons trois algorithmes selon le type de graphe de communication : anneau unidirectionel, anneau bidirectionnel et graphe quelconque. Ces trois algorithmes ont de plus l'avantage de ne nécessiter de chaque site que la connaissance de ses voisins dans le graphe. Autrement dit, l'insertion d'un nouveau site ne requiert aucune intervention au niveau des sites qui ne sont pas ses voisins (moyennant le fait que le type de graphe est inchangé dans le cas de graphes spécifiques).

# 2 Algorithme de Chang et Roberts [Cha79]

## *2.1 Principe et réalisation de l'algorithme*

Cet algorithme s'applique à un anneau unidirectionnel. Chaque site n'émet des messages que vers le site suivant sur l'anneau.

Lorsque l'application « réclame » l'identité du leader, trois cas se présentent :

- Le processus d'élection est terminé et le service renvoie immédiatement l'identité du site élu.
- Le processus d'élection est en cours et connu du service. Dans ce cas, le service attend la terminaison du processus afin de renvoyer l'identité du leader.
- Le service n'est pas au courant d'un processus d'élection engagé. Dans ce cas, il envoie une requête circuler sur l'anneau afin de devenir le leader. Si la requête lui revient, il conclut que son offre est acceptée et envoie un message de confirmation circuler sur l'anneau pour indiquer aux autres sites qu'il est le leader.

Lorsqu'une requête parvient à un autre site, deux cas se présentent :

- Le service du site était au repos ou avait provisoirement choisi un leader d'identité supérieure; il adopte l'initiateur de la requête comme nouveau leader potentiel et retransmet la requête au site suivant de l'anneau.
- Le service du site avait provisoirement choisi un leader d'identité inférieure; il ne donne pas suite à la requête.

La figure 7.1 décrit les phases successives d'une exécution possible de l'algorithme. Sur la figure 7.1.a, l'application du site 4 a appelé la fonction `leader()`ce qui a provoqué l'envoi d'une requête du service pour devenir « leader ». Sur la figure 7.1.b, trois autres applications ont appelé la fonction `leader()` : les sites 2,3,5. Puisqu'une requête a déjà été retransmise par le service du site 5, celui-ci n'envoie pas de requête et attend la fin de l'élection (il a adopté provisoirement 4 comme leader). Les deux autres sites envoient des requêtes. Sur la figure 7.1.c, seules deux requêtes circulent encore puisque la requête du site 4 a été « détruite » par le site 2. Tous les sites sont maintenant au courant du processus d'élection. Sur la figure 7.1.d, seule la requête du site 2 circule encore car celle du site 3 a aussi été « détruite » par le site 2. Notez que tous les sites ont adopté provisoirement le site 2 comme leader. Sur la figure 7.1.e, le site 2 sait qu'il est élu puisque sa requête lui est revenue. Il envoie un message de confirmation aux autres sites. Sur la figure 7.1.f, tous les sites ont adopté définitivement le site 2 comme leader et le message de confirmation sera détruit à la réception par le site 2.

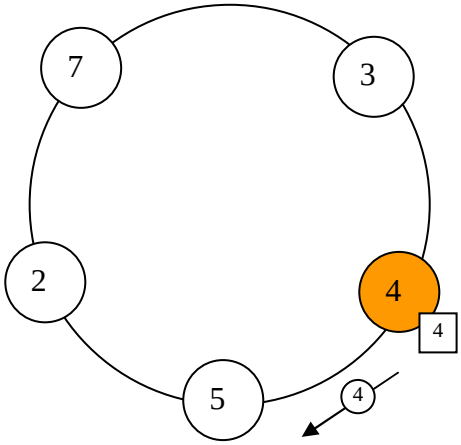



Figure 7.1.a

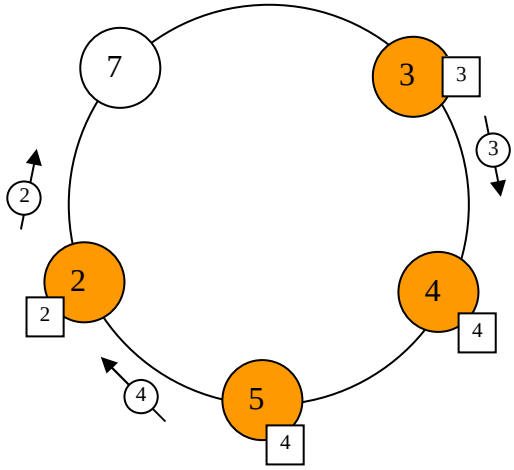



Figure 7.1.b

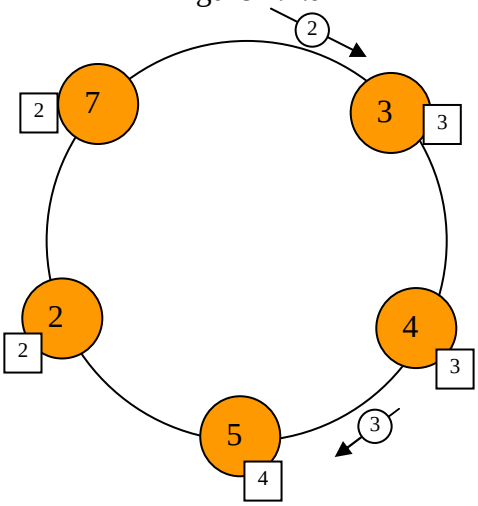



Figure 7.1.c

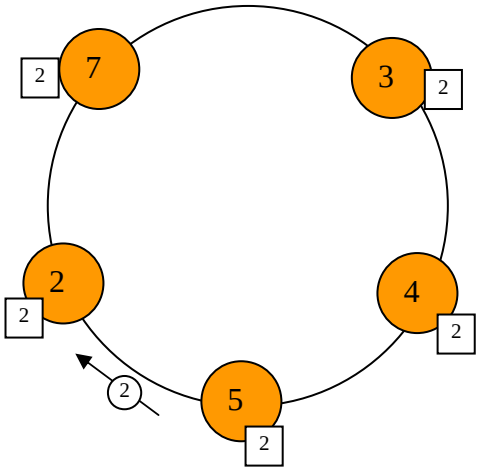



Figure 7.1.d

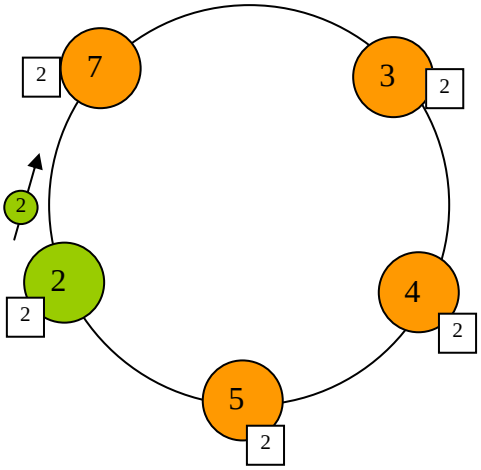



Figure 7.1.e

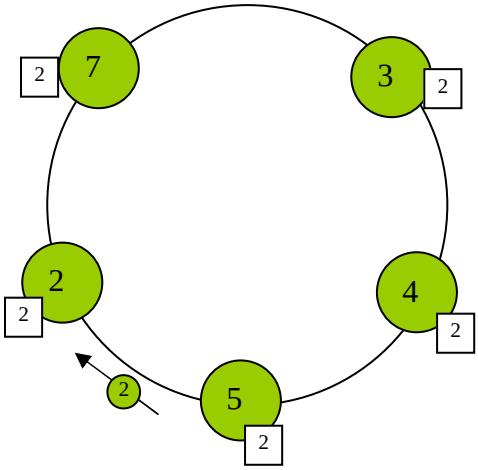



Figure 7.1.f

Nous sommes maintenant en mesure d'énoncer l'algorithme.

Variables du site `i`

- $\texttt{suivant}_i$ : constante contenant l'identité du site successeur de `i` sur l'anneau.
- $\texttt{état}_i$ : état du service. Cette variable prend une valeur parmi l'ensemble des valeurs (`repos`,`en_cours`,`terminé`). Cette variable est initialisée à `repos`.
- $\texttt{chef}_i$ : identité du site élu.

Algorithme du site `i`

Si aucun processus d'élection n'a atteint le site `i`, le service initialise un tel processus. Dans tous les cas, il attend que le processus d'élection soit terminé pour renvoyer l'identité du site élu.

```
leader()
Début
     Si (état_i==repos) Alors
          état_i=en_cours;
          chef_i=i;
          envoyer_à(suivant_i,(req,i));
     Finsi
     Attendre(état_i==terminé);
     renvoyer(chef_i);
Fin
```

Si le site est au repos, la réception d'une requête provoque le changement d'état et la retransmission de la requête. Dans le cas où le processus reçoit une "meilleure" requête, il en tient compte et retransmet aussi la requête. Enfin si une requête parvient à son initiateur, ce site est élu et il avertit les autres sites.

```
sur_réception_de(j,(req,k))
Début
     Si (état_i==repos || k<chef_i) Alors
          état_i=en_cours;
          chef_i=k;
          envoyer_à(suivant_i,(req,k));
     Sinon si (i==k) Alors
          état_i=terminé;
          envoyer_à(suivant_i,(conf,i));
     Finsi
Fin
```

La confirmation du site élu fait un tour de l'anneau.

```
sur_réception_de(j,(conf,k))
Début
     Si (i!=k) Alors
          envoyer_à(suivant_i,(conf,k));
          état_i=terminé;
     Finsi
Fin
```

### *2.2 Preuve de l'algorithme*

Un site dont l'application appelle `leader()` alors que son service est au repos sera désigné comme un initiateur. Soit $i_0$ l'initiateur d'identité minimale, la requête $(req, i_0)$ circule de bout en bout et atteint $i_0$. Le site $i_0$ passera donc à l'état `terminé`, enverra son message de confirmation qui entrainera à sa réception le passage de tous les autres sites à ce même état et l'adoption de $i_0$ comme leader.

Il reste à démontrer qu'aucun autre message de confirmation ne circulera sur le réseau. Soit $i_1$ un autre initiateur, si la requête de $i_1$ parvient à $i_0$, elle sera détruite puisque l'émission de la requête de $i_0$ précède cette réception (par définition d'un initiateur). En conséquence, aucun autre message de confirmation ne sera émis.

### *2.3 Complexité de l'algorithme*

2.3.1 Complexité au pire des cas

Nous désirons obtenir un ordre de grandeur sur `Pire(n)`, le nombre maximum de messages échangés durant une élection sur un anneau composé de `n` sites. Notons d'abord que chaque site envoie au plus une requête. Chaque requête donne lieu à au plus `n` envois de message. Enfin, le message de confirmation donne lieu à exactement `n` envois de message. En cumulant ces bornes nous obtenons :

$$Pire(n) \leq n^2 + n$$

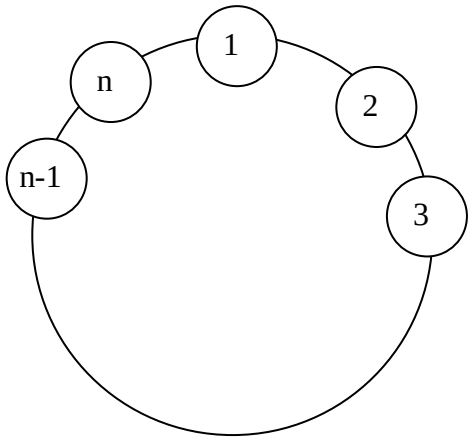



Figure 7.2 : Une configuration défavorable

Examinons le cas de la figure 7.2. où tous les sites sont initiateurs, la requête issue de i est envoyée exactement `n-i+1` fois. Ce qui nous donne pour cette exécution particulière un nombre de messages égal à :

$$\sum_{i=1 \text{ à } n} (n-i+1) + n = n(n+1)/2 + n \leq Pire(n)$$

Cet encadrement nous fournit l'ordre de grandeur recherché :

$$Pire(n) = \theta(n^2).$$

2.3.2 Complexité en moyenne

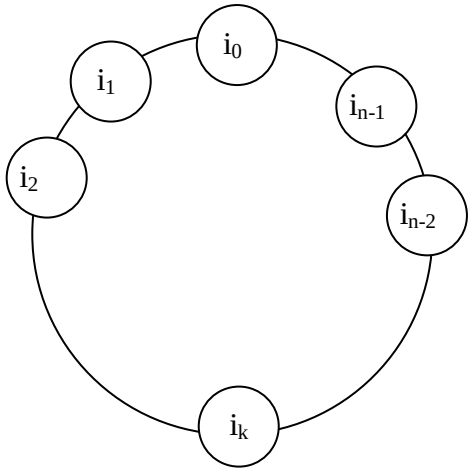



Figure 7.3 : Une configuration aléatoire

Nous cherchons maintenant à évaluer $Moyenne(n)$ qui représente le nombre moyen de messages échangés sur l'ensemble des exécutions possibles (nous supposons dans la suite que tous les sites sont initiateurs). Posons $i_0$ le site d'identité minimale et considérons la configuration aléatoire de la figure 7.3. Soit $X_k$ la variable aléatoire représentant le nombre de messages issus de la requête de $i_k$ (pour $k \neq 0$). Puisque cette requête ne pourra être relayée par $i_0$, nous avons : $Prob(X_k \geq k+1)=0$.

Pour que cette requête soit relayée au moins $t$ fois pour $t \leq k$ il faut et il suffit que $i_k$ soit l'identité minimale dans l'ensemble des identités $\{i_k,...,i_{k-t+1}\}$. Nous supposons que toutes les configurations sont équiprobables, donc cette minimalité est satisfaite avec une probabilité $1/t$. Autrement dit, $Prob(X_k \geq t)= 1/t$.

Nous appliquons maintenant un résultat élémentaire sur l'espérance d'une variable aléatoire à valeurs entières :

$E(X) =_{def} \sum_{t=1 \text{ à } \infty} t.Prob(X=t)$
$= \sum_{t=1 \text{ à } \infty} \sum_{s=1 \text{ à } t} Prob(X=t)$ *(remplacement du produit par la somme)*
$= \sum_{s=1 \text{ à } \infty} \sum_{t=s \text{ à } \infty} Prob(X=t)$ *(inversion des sommes)*
$= \sum_{s=1 \text{ à } \infty} Prob(X \geq s)$

D'où : $E(X_k) = \sum_{t=1 \text{ à } k} 1/t$

Nous sommons ces différentes espérances en ajoutant les deux tours (requête et confimation) dûs à $i_0$ :

$$\text{Moyenne}(n) = \sum_{k=1 \text{ à } n-1} \sum_{t=1 \text{ à } k} 1/t + 2.n$$
$$= \sum_{t=1 \text{ à } n-1} \sum_{k=t \text{ à } n-1} 1/t + 2.n$$
$$= \sum_{t=1 \text{ à } n-1} (n-t)/t + 2.n$$

*(inversion des sommes et remplacement de la somme par le produit)*

$$= \sum_{t=1 \text{ à } n-1} n/t + \sum_{t=1 \text{ à } n-1} -t/t + 2.n$$
$$= n.\sum_{t=1 \text{ à } n-1} 1/t + n + 1$$
$$= n.\sum_{t=1 \text{ à } n-1} 1/t + n + n.(1/n)$$

Finalement :

$$\text{Moyenne}(n) = n.\sum_{t=1 \text{ à } n} 1/t + n$$

Il nous reste à obtenir un ordre de grandeur de cette quantité. Nous nous servons de l'encadrement immédiat :

$$\int_t^{t+1} 1/x \, dx \leq 1/t \leq \int_{t-1}^{t} 1/x \, dx \text{ pour } t \geq 2$$

ce qui nous donne par sommation :

$$\log(n+1) = \int_1^{n+1} 1/x \, dx \leq \sum_{t=1 \text{ à } n} 1/t \leq 1 + \int_1^{n} 1/x \, dx = 1+\log(n)$$

D'où :

$$\text{Moyenne}(n) = \theta(n.\log(n))$$

# 3 Algorithme de Franklin [Fra82]

## *3.1 Principe et évaluation de l'algorithme*

Cet algorithme s'applique à un anneau bidirectionnel. Il se décompose en "tours" où les compétiteurs sont les initiateurs (au même sens que dans l'algorithme précédent). A chaque tour, un initiateur survivant envoie à ses initiateurs voisins (à gauche et à droite) sa candidature. De ce fait, chaque initiateur reçoit les requêtes des initiateurs voisins gauche et droite et ne survit au tour suivant que s'il a la plus petite identité. Le tournoi s'achève quand un initateur sait qu'il est ou sera au prochain tour le seul survivant. Il acquiert cette certitude soit en recevant l'un des messages qu'il a envoyé (ou éventuellement les deux) soit en recevant deux messages d'un même site. Les sites "éliminés" participent à l'algorithme en transmettant les messages qu'ils reçoivent. Comme dans l'algorithme précédent un message de confirmation achève l'algorithme.

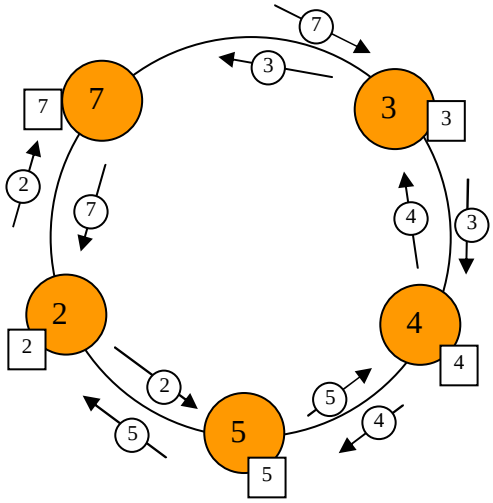



Figure 7.5.a

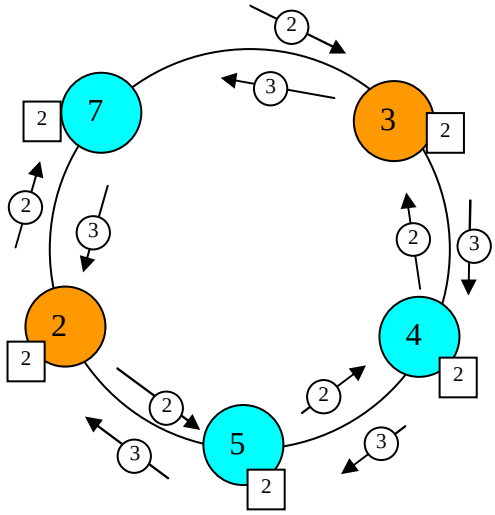



Figure 7.5.b

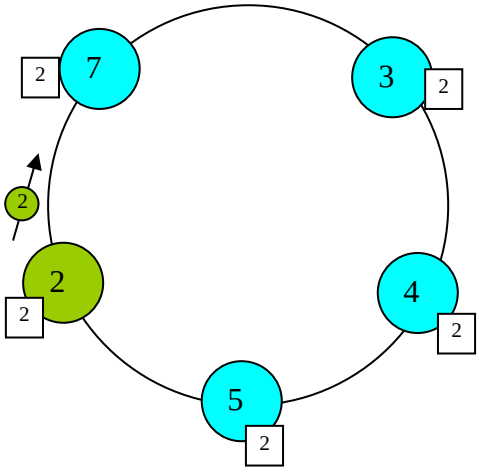



Figure 7.5.c

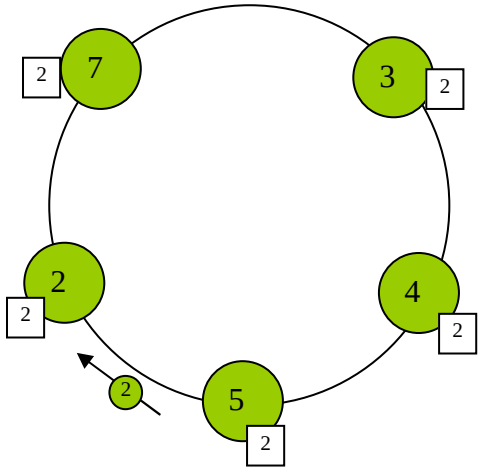



Figure 7.5.d

Nous avons représenté sur la figure 7.5, une exécution possible de l'algorithme. Sur la figure 7.5.a, toutes les applications ont appelé la fonction `leader()`. Par conséquent, chaque site envoie sa candidature à gauche et à droite. Seuls le site 2 et 3 survivent et se retrouvent au deuxième tour. Aussi, ils s'envoient à nouveau leur candidature qui sont relayés par les sites éliminés (figure 7.5.b). Lors de ce tour le site 2 est le seul survivant et de plus il le sait car il a reçu deux messages issus du site 3. Il envoie donc un message de confirmation (figure 7.5.c). Lorsque ce message de confirmation a fait le tour de l'anneau (figure 7.5.d), le processus d'élection est terminé.

Evaluons cet algorithme dans le pire des cas. Nous remarquons d'abord qu'à chaque tour exactement $2.n$ messages sont échangés (un tour dans chaque sens). Nous majorons le nombre de tours de la manière suivante. A chaque initiateur survivant d'un tour on associe son voisin initiateur de droite éliminé lors de ce tour. Cette fonction est injective. Donc le nombre d'éliminés durant un tour est au moins égal au nombre de survivants. Autrement dit, le nombre de survivants est au moins divisé par 2 à chaque tour. Sachant qu'il peut y avoir un tour supplémentaire (inutile), on obtient que le nombre de tours est majoré par $ENT(\log_2(n))+1$.

En ajoutant le message de confirmation, on obtient :

$$Pire(n) \leq 2.n.(ENT(\log_2(n))+1)+ n = \theta(n.\log(n))$$

Nous laissons le soin au lecteur d'exhiber un exemple d'exécution démontrant que l'on a :

$$Pire(n) = \theta(n.\log(n))$$

Indication On placera $n=2^k$ sites de la manière itérative suivante. On place d'abord les sites 1 et 2. Une fois placés les $2^i$ sites de plus basses identités, on intercale les $2^i$ sites suivants entre les sites déjà placés. On vérifie par récurrence que $k$ tours sont nécessaires.

Notons que cet algorithme a été adapté pour fonctionner sur un anneau unidirectionnel avec la même complexité de messages [Pet82] et [Dol82].

### *3.2 Description*

Il y a une difficulté supplémentaire due à l'asynchronisme du réseau. Un site peut être en avance d'un tour sur un candidat voisin. Par exemple si $i_0$, $i_1$, $i_2$, $i_3$ et $i_4$ sont des candidats "voisins" à un tour donné, $i_3$ peut avoir éliminé $i_2$ et $i_4$ et avoir envoyé ses messages du tour suivant alors que $i_1$ attend encore la réponse de $i_0$. Il faut donc que $i_1$ conserve ce message en avance pour le traiter au tour suivant. Notons qu'un site ne peut recevoir qu'un message en avance et qu'il le reçoit dans la même direction que le premier message reçu.

Variables du site `i`

- $suivant_i$ : constante contenant l'identité du site successeur de `i` sur l'anneau.
- $précédent_i$ : constante contenant l'identité du site prédécesseur de `i` sur l'anneau.
- $état_i$ : état du service. Cette variable prend une valeur parmi (`repos`, `en_cours`, `attente`, `terminé`). Cette variable est initialisée à `repos`.
- $nbreq_i$ : nombre de requêtes reçues au cours d'un tour

- $conc_i$ : identité du site dont on reçoit la première des deux requêtes du tour courant. Lorsqu'elle vaut `i`, cela signifie qu'aucune requête en avance n'est reçue.
- $dir_i$ : booléen indiquant la direction d'où vient la première des deux requêtes du tour courant
- $concav_i$ : identité du site dont on reçoit la première des deux requêtes du tour suivant
- $chef_i$ : identité du site (provisoirement) élu.

Algorithme du site `i`

Si aucun processus d'élection n'a pas atteint le site `i`, le service initialise un tel processus. Au cours de ce processus, il propose sa candidature dans les deux sens de l'anneau tant qu'il n'est pas éliminé ou qu'il ne sait pas qu'il est le seul survivant. Dans tous les cas, il attend que le processus d'élection soit terminé pour renvoyer l'identité du site élu. La boucle `répéter` correspond à la participation du site aux tours successifs. A chaque changement de tour, il faut prendre garde à traiter l'éventuelle requête en avance.

```
leader()
Début
     Si (état_i==repos) Alors
          état_i=en_cours;
          chef_i=i;concav_i=i;
          Répéter
               nbreq_i=0;
               Si concav_i!=i Alors
                    nbreq_i=1;
                    conc_i=concav_i;
                    Si (conc_i<chef_i) Alors chef_i=conc_i; Finsi
                    concav_i=i;
               Finsi
               envoyer_à(suivant_i,(req,i));
               envoyer_à(précédent_i,(req,i));
               Attendre(nbreq_i==2);
          Jusqu'à(état_i !=en_cours);
          Si concav_i!=i Alors
               Si (dir_i) Alors
                    envoyer_à(suivant_i,(req,concav_i));
               Sinon
                    envoyer_à(précédent_i,(req,concav_i));
               Finsi
          Finsi
     Finsi
     Attendre(état_i==terminé);
     renvoyer(chef_i);
Fin
```

A la réception d'une requête, on met à jour si nécessaire l'identité provisoire du leader. Dans le cas où on est un initiateur survivant (état à `en_cours`), on enregistre les deux requêtes voisines en prenant garde à mémoriser une requête en avance. Sur la réception de la deuxième, on teste si on a survécu puis si on est le seul survivant. Dans le cas où on est en `attente`, on retransmet les messages (fonction proxy).

```
sur_réception_de(j,(req,k))
Début
     Si (état_i == repos || k<chef_i) Alors
          chef_i = k;
     Finsi
     Si (état_i == en_cours) Alors
          Si (nbreq_i==0) Alors
               conc_i=k;
               nbreq_i=1;
               dir_i=(j==précédent_i);
          Sinon si ((dir_i&&j==précédent_i)||
          (!dir_i&&j!=précédent_i)) Alors
               concav_i=k;
          Sinon
               nbreq_i=2;
               Si (chef_i<i) Alors
                    état_i=attente;
               Sinon si (conc_i==i||k==conc_i) Alors
                    état_i = terminé;
                    envoyer_à(suivant_i,(conf,i));
               Finsi
          Finsi
     Sinon
          état_i=attente;
          Si (j==précédent_i) Alors
               envoyer_à(suivant_i,(req,k));
          Sinon
               envoyer_à(précédent_i,(req,k));
          Finsi
     Finsi
Fin
```

La confirmation du site élu fait un tour de l'anneau.

```
sur_réception_de(j,(conf,k))
Début
     Si (i!=k) Alors
          envoyer_à(suivant_i,(conf,k));
          état_i=terminé;
     Finsi
Fin
```

# 4 Généralisation de l'algorithme de Chang et Roberts

## *4.1 Algorithme de parcours du graphe de communication [Tar1895]*

4.1.1 Présentation informelle et preuve

On désire généraliser l'algorithme de Chang et Roberts à un graphe de communication quelconque. La difficulté de cette généralisation réside dans le fait que la connaissance du graphe par un site se limite à ses voisins. Nous allons maintenant établir que cette connaissance minimale est suffisante pour gérer un parcours du graphe de communication qui vérifie les propriétés suivantes :

1. Ce parcours se termine chez l'initiateur.
2. Lorsqu'il se termine, ce parcours a visité au moins une fois tous les sites et a traversé exactement une fois chaque canal de communication dans les deux directions.

Attention, ***le parcours dépend de l'initiateur*** mais ceci n'affecte pas la correction de l'algorithme d'élection dont nous verrons la preuve plus loin.

Nous allons expliquer l'algorithme sur un exemple décrit à la figure 7.6.

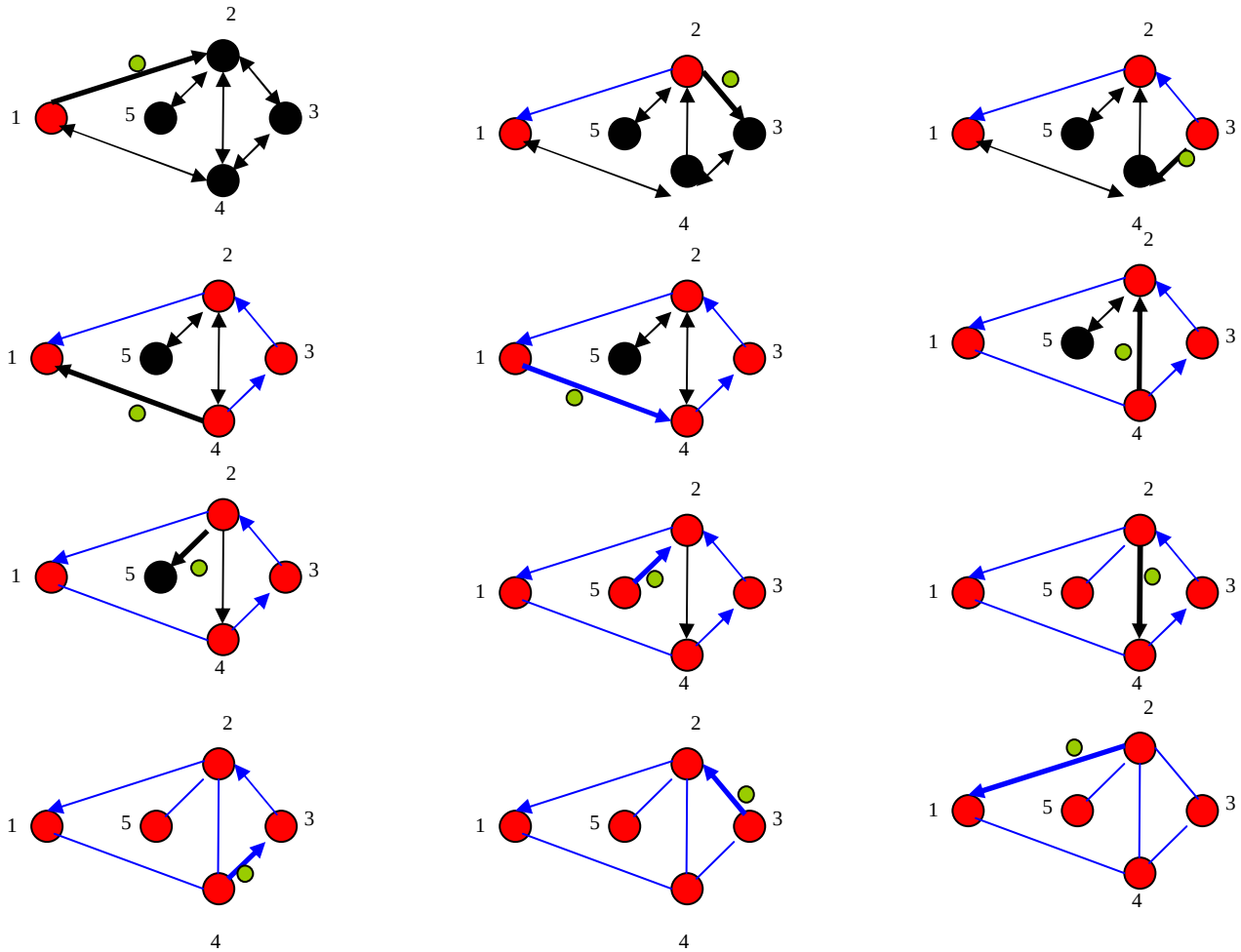



Figure 7.6 Un parcours de graphe

Chaque canal sortant est considéré comme initialement "ouvert" (ce qui est représenté sur la figure par une flèche à l'extrémité concernée). Lorsque le message associé au parcours est envoyé sur un canal ouvert, ce canal se "ferme" et ne pourra plus être utilisé par le parcours. Un site déjà visité par ce message est indiqué sur la figure en rouge et dans le cas contraire en noir. De plus, lors de la première visite par le message (excepté pour l'initiateur), le site mémorise le canal comme étant celui de son père, indiqué en bleu sur la figure. Il n'utilisera ce canal qu'après avoir utilisé tous les autres canaux.

Déroulons maintenant l'exemple sur les 12 schémas successifs de la figure :

1. Le site initie un parcours en envoyant le message au site 2. Par conséquent, il clôt ce canal.
2. A la réception, le site 2 mémorise que le canal de 2 vers 1 est le canal de "retour". Il a le choix entre les trois autres canaux et décide de l'envoyer au site 3 (canal fermé).
3. A la réception, le site 3 mémorise que le canal de 3 vers 2 est le canal de "retour". Il ne dispose que d'un autre canal et l'envoie au site 4 (canal fermé).
4. A la réception, le site 4 mémorise que le canal de 4 vers 3 est le canal de "retour". Il a le choix entre les deux autres canaux et décide de l'envoyer au site 1 (canal fermé).
5. A la réception, le site 1 ne dispose plus que d'un autre canal et le renvoie au site 4 (canal fermé).
6. A la réception, le site 4 ne dispose plus que d'un canal différent du canal de retour et le renvoie au site 2 (canal fermé).
7. A la réception, le site 2 a le choix entre deux canaux différents du canal de retour et décide de l'envoyer au site 5 (canal fermé).
8. A la réception, le site 5 mémorise que le canal de 5 vers 2 est le canal de "retour". Il ne dispose d'aucun autre canal et le renvoie au site 2 (canal fermé).
9. A la réception, le site 2 ne dispose plus que d'un canal différent du canal de retour et le renvoie au site 4 (canal fermé).
10. A la réception, le site 4 ne dispose que du canal de retour et le renvoie au site 3 (canal fermé).
11. A la réception, le site 3 ne dispose que du canal de retour et le renvoie au site 2 (canal fermé)
12. A la réception, le site 2 ne dispose que du canal de retour et le renvoie au site 1 (canal fermé). Lorsque ce message arrive au site 1, celui-ci n'ayant plus de canal ouvert conclut que le parcours est terminé.

Avant de développer cet algorithme, nous établissons sa correction. Appelons pour un site $i$, le nombre de canaux sortants (et donc entrants) $n_i$.

Tout d'abord, le parcours se termine nécessairement car un canal n'est emprunté qu'au plus une fois et le nombre de canaux est fini. Supposons qu'il se termine ailleurs que chez l'initiateur. Ceci signifie que ce site $i$ n'a plus de canal sortant ouvert, lors d'une visite. Or puisqu'il décrémente le nombre de canaux sortants à chaque visite. On en conclut qu'il a été visité plus de $n_i$ fois. Mais puisque $n_i$ est également le nombre de canaux entrants, cela signifie qu'un canal entrant a été emprunté plus d'une fois. Ce qui est impossible. Le parcours se termine donc chez l'initiateur.

Démontrons maintenant que lorsque l'algorithme se termine, alors tous les noeuds visités ont envoyé le message sur tous leurs canaux sortants. On le démontre par récurrence sur l'ordre des visites. Pour l'initiateur c'est évident, puisque c'est la condition d'arrêt de l'algorithme. Supposons par l'absurde que `i` soit le premier site visité à ne pas emprunter tous ses canaux sortants. Dans ce cas, il n'envoie pas le message à son "père", que nous notons `j`. Par hypothèse de récurrence, `j` a emprunté tous ses canaux sortants soit $n_j$ canaux, mais il n'a pas été visité par la canal venant de i, donc il a été visité moins $n_j$ de fois. Ce qui est contradictoire.

Démontrons finalement que tous les sites sont visités. Si ce n'est pas le cas, on partitionne l'ensemble des sites entre ceux qui sont visités et les autres. Puisque le graphe est connexe, il existe au moins une arête reliant un site visité à un site non visité. D'après le paragraphe précédent, ce canal a été emprunté d'où la contradiction.

### 4.1.2 Réalisation de l'algorithme

Tout d'abord, nous modifions légérement l'algorithme pour permettre plus plusieurs parcours successifs initiés par un même site. Notre algorithme se compose de deux fonctions `initier(type)` qui initie un parcours avec un certain type de message et `faire_suivre(j,type,k)` qui fait suivre un type de message venant de `j` dans un parcours initié par `k`. Cette fonction est appelée à la réception dudit message.

Pour gérer un parcours initié par un quelconque des sites, on utilise un tableau $T_i$ indicé par les sites. Ceci peut sembler contradictoire avec l'hypothèse de minimalité sur la connaissance des sites, mais il est facile de remplacer $T_i$ par une allocation dynamique au prix d'une programmation plus "lourde".

L'algorithme travaille avec des ensembles de canaux (ou voisins). La fonction `extraire(ensemble)` renvoie un élement de l'ensemble (s'il est non vide) et le retire de celui-ci. On s'autorise bien entendu à tester si un ensemble est vide.

Variables du site `i`

- $Voisins_i$: constante contenant l'ensemble des voisins de `i` dans le graphe de communication.
- $T_i$`[1..N]`: tableau dont la cellule `j` contribue au parcours d'un jeton initié par `j`. Chaque cellule a deux champs :
  - $T_i$`[j].voisins`
  - $T_i$`[j].père` initialisé à `i` qui signifie dans le cas où `j`$\neq$`i` que le parcours de `j` n'est pas encore parvenu à `i`.
- $prochain_i$ : variable contenant l'identité d'un voisin de `i`.

Algorithme du site `i`

Pour initier un parcours du graphe, on initialise le champ voisins aux voisins du site et on extrait un voisin de cet ensemble pour débuter le parcours.

```
initier(type)
Début
     T_i[i].voisins = Voisins_i;
     prochain_i = extraire(T_i[i].voisins);
     envoyer_à(prochain_i,(type,i));
Fin

sur_réception_de(j,(type,k))
Début
     Si (!faire_suivre(j,type,k)) Alors
          afficher("parcours terminé");
     Finsi
Fin
```

S'il s'agit de la première visite de ce parcours, on initialise les champs père et voisins de la cellule concernée en extrayant le "père" des voisins. S'il reste des voisins, on extrait l'un d'entre eux pour continuer le parcours. Sinon si `i` n'est pas l'initiateur du parcours, on emprunte le canal de son père. Le parcours étant terminé sur ce site, on réinitialise le champ `père` pour un éventuel nouveau parcours. Si `i` est l'initiateur, la fonction renvoie l'indication que le parcours est terminé.

```
faire_suivre(j,type,k)
Début
     Si (k!=i) && (T_i[k].père==i) Alors
          T_i[k].père = j;
          T_i[k].voisins = Voisins_i \{j};
     Finsi
     Si T_i[k].voisins != ∅ Alors
          prochain_i = extraire(T_i[k].voisins);
          envoyer_à(prochain_i,(type,k));
          renvoyer(VRAI);
     Sinon si (k!=i) Alors
          envoyer_à(T_i[k].père,(type,k));
          T_i[k].père = i;
          renvoyer(VRAI);
     Sinon
          renvoyer(FAUX);
     Finsi
Fin
```

### *4.2 Description de l'algorithme d'élection*

Variables du site `i`

Aux variables nécessaires au parcours, on ajoute les variables nécessaires à l'élection.

- $\texttt{état}_\texttt{i}$ : état du service. Cette variable prend une valeur parmi l'ensemble des valeurs (`repos`,`en_cours`,`terminé`). Cette variable est initialisée à `repos`.
- $\texttt{chef}_\texttt{i}$ : identité du site élu.

Algorithme du site `i`

Si aucun processus d'élection n'a atteint le site `i`, le service initialise un tel processus par un parcours du graphe avec une requête. Dans tous les cas, il attend que le processus d'élection soit terminé pour renvoyer l'identité du site élu.

```
leader()
Début
     Si (état_i==repos) Alors
          état_i=en_cours;
          chef_i=i;
          initier(req);
     Finsi
     Attendre(état_i==terminé);
     renvoyer(chef_i);
Fin
```

Si le site est au repos, la réception d'une requête provoque le changement d'état. Dans le cas où le processus reçoit une "meilleure" requête, il en tient aussi compte. Dans le cas contraire, le parcours est avorté (disparition implicite de la requête) et retransmet aussi la requête. Si le parcours n'est pas avorté alors à l'aide de la fonction `faire_suivre()`, soit on continue le parcours soit si le parcours est teminé (`i` est alors le site élu) on initie un parcours d'un confirmation.

```
sur_réception_de(j,(req,k))
Début
     Si (état_i==repos || k<=chef_i) Alors
          état_i=en_cours;
          chef_i=k;
          Si !faire_suivre(j,req,k) Alors
               état_i=terminé;
               initier(conf);
          Finsi
     Finsi
Fin
```

La confirmation du site élu fait un parcours du graphe de communication.

```
sur_réception_de(j,(conf,k))
Début
     état_i=terminé;
     faire_suivre(j,conf,k);
Fin
```

### *4.3 Correction et complexité de l'algorithme d'élection*

Supposons que deux requêtes soient envoyées par deux initiateurs. Alors le parcours de l'initiateur de plus grande identité ne pourra se terminer car il visitera l'autre initiateur d'après la propriété d'exhaustivité du parcours.

Donc seul le message de confirmation de l'iniateur de plus petite identité est émis et parcourt le graphe.

Chaque initiateur initie un parcours de graphe et l'élu initie un deuxième parcours pour le message de confirmation. En notant E le nombre d'arêtes du graphe de communication, on obtient :

$$Pire(n) \leq (n+1).(2.E) = \theta(n.E)$$

L'exemple d'exécution de l'algorithme de Chang et Roberts établit que l'on a :

$$Pire(n) = \theta(n.E)$$

On peut adapter cet algorithme avec les idées de l'algorithme de Franklin afin d'obtenir un algorithme qui se comporte plus efficacement dans le cas où des parcours intelligents du graphe de communication sont possibles (e.g. grille, tore, etc.) [Kor90].

# 5 Exercices

***Sujet 1***

Dans ce problème, `n` désigne le nombre de sites et `E` le nombre d'arêtes du graphe de communication. On suppose que l'exécution d'instructions est instantanée (i.e. en un temps négligeable devant les délais de transit).

**Question 1** En supposant qu'il y ait un seul initiateur et que la transmission de chaque message prenne exactement `1` unité de temps, calculez le temps nécessaire à l'élection effectuée par l'adaptation (vue en cours) de l'algorithme de Chang et Roberts pour un graphe de communication quelconque. En déduire l'inconvénient majeur de cet algorithme.

Afin de remédier à cet inconvénient, on se propose de définir un nouvel algorithme d'élection. Les variables de ce nouvel algorithme sont les suivantes :

- $\texttt{Voisins}_i$: constante contenant l'ensemble des voisins de `i` dans le graphe de communication ;
- $\texttt{état}_i$ : état du service. Cette variable prend une valeur parmi l'ensemble des valeurs (`repos`,`en_cours`,`terminé`). Cette variable est initialisée à `repos` ;
- $\texttt{chef}_i$ : identité du site élu ;
- $\texttt{père}_i$ : identité du site à qui envoyer une « dernière » requête ;
- $\texttt{nbreq}_i$ : nombre de requêtes à recevoir avant d'envoyer la dernière requête ;
- $\texttt{temp}_i$ : variable temporaire.

Lorsque l'application « réclame » l'identité du leader, trois cas se présentent :

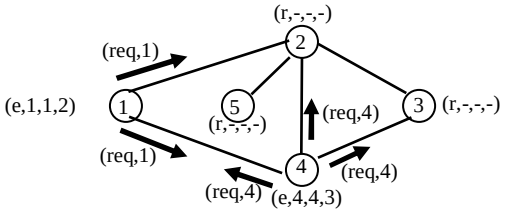
- Le processus d'élection est terminé et le service renvoie immédiatement l'identité du site élu.
- Le processus d'élection est en cours et connu du service. Dans ce cas, le service attend la terminaison du processus pour renvoyer l'identité du leader.
- Le service n'est pas au courant d'un processus d'élection engagé. Dans ce cas, il envoie à tous ses voisins une requête portant son identité afin de devenir le leader, se considère (provisoirement) comme l'élu et son propre père, initialise le nombre de requêtes attendues au nombre de voisins puis il attend la terminaison du processus pour renvoyer l'identité du leader.

Lorsqu'une requête parvient à un site, trois cas se présentent :

- Le service du site était au repos ou avait provisoirement choisi un leader d'identité supérieure. Il adopte l'initiateur de la requête comme nouveau leader potentiel et l'émetteur du message devient son nouveau père. S'il a des voisins différents de son père, il retransmet à tous ses voisins excepté son père la requête et (ré)initialise le nombre de requêtes attendues au nombre de voisins`-1`. Sinon il retransmet immédiatement la requête à son père.
- Le service du site avait provisoirement choisi un leader d'identité inférieure. Il ne donne pas suite à la requête.
- La requête correspond au leader actuel du site. Il décremente alors le nombre de requêtes attendues. Si celui-ci devient nul, soit le site n'est pas l'initiateur de la requête et la retransmet à son père, soit le site est l'initiateur, l'état du service est maintenant terminé et il envoie à ses voisins un message de confirmation.

La réception du message de confirmation est laissée à votre réflexion.

**Question 2** Déroulez l'algorithme sur le graphe de communication ci-dessous en supposant que le site `1` et le site `4` appellent simultanément `leader(),` que la transmission de chaque message prenne exactement `1` unité de temps et que si deux messages arrivent simultanément sur un même site, on traite d'abord le message correspondant à l'émetteur de plus grande identité. Vous présenterez l'état de chaque site sous la forme $(\text{état}_i, \text{chef}_i, \text{père}_i, \text{nbreq}_i)$ et les messages qui circulent aux instants `0,1,2,...` jusqu'à la terminaison de l'algorithme. La situation initiale est décrite sur le graphe.




**Question 3** Ecrivez les primitives suivantes :

- `leader()`
- `sur_réception_de(j,(req,k))`
- `sur_réception_de(j, conf)`

**Question 4** Donnez une borne supérieure du nombre de messages échangés pour une élection en fonction de `n` et de `E`.

**Question 5** On appelle distance entre deux sites, la longueur en nombre d'arêtes du plus court chemin entre deux sites. Le diamètre du graphe `D` est la plus grande distance entre deux sites quelconques. En supposant qu'une transmission de message prenne au plus `1` unité de temps, indiquez en fonction de `D`, le temps maximum entre le premier appel à `leader()` et le moment où tous les sites sont dans l'état terminé.

## 6 Références

[Cha79] E.J.-H. Chang, R. Roberts "An improved algorithm for decentralized extrema finding in circular arrangements of processes" Communication of ACM vol 22 (1979) pp 281-283.

[Dol82] D. Dolev, M. Klawe, M. Rodeh " An O(n log n) unidirectional distributed algorithm for extrema-finding in a circle." Journal of algorithms 3 (1982) pp 245-260.

[Fra82] W.R. Franklin "On an improved algorithm for decentralized extrema finding in circular configurations of processors" Communications of the ACM 25,5 (1982) pp 336-337.

[Kor90] E. Korach, S. Kutten, S. Moran "A modular technique for the design of efficient leader finding algorithms" ACM Transactions on Programming Languages and Systems 12 (1990) pp 84-101.

[Pet82] G.L. Peterson "An O(n log n) unidirectional algorithm for the circular extrema problem." ACM transctions on programming languages and systems 4 (1982) pp 758-762.

[Tar1895] G. Tarry "Le problème des labyrinthes" Nouvelles Annales de Mathématiques 14. (1895)

# CHAPITRE VII

## MÉMOIRE VIRTUELLE RÉPARTIE

*version du 13 novembre 2003*

## 1 Problématique

Dans le chapitre sur la communication, nous avons examiné des alternatives au modèle de programmation répartie par envoi de messages (e.g. le rendez-vous). Dans ce chapitre, nous étudierons l'un de ces modèles alternatifs : la mémoire virtuelle répartie. L'idée sous-jacente est de fournir au programmeur l'illusion de travailler avec des processus partageant un espace mémoire unique.

Pour simplifier notre propos, nous supposons que cet espace est structuré en objets (ou variables partagées) et que le programmeur dispose des seules opérations de lecture et d'écriture. Le lecteur se convaincra que ces simplifications ne restreignent pas la généralité des solutions algorithmiques de ce chapitre.

Il y a en réalité deux problèmes distincts [Ray93].

- D'une part, la sémantique des accès concurrents à la mémoire doit être formellement définie. En effet, en raison de l'activité parallèle des processus, plusieurs exécutions d'un même programme sont possibles. La sémantique de ces accès permet au programmeur de savoir quelles synchronisations il doit introduire afin d'obtenir le comportement escompté.
- Une fois cette sémantique définie, il faut construire un "protocole" qui garantit que cette sémantique est préservée. Il s'agit en fait d'un algorithme réparti associé au service d'accès à la mémoire virtuelle répartie.

Le service que nous réaliserons a une interface constituée de deux fonctions : `read(x)` qui renvoie le contenu de l'objet `x` et `write(x,v)` qui met à jour l'objet `x` avec la valeur `v`. La structure des objets est sans importance pour les algorithmes. Aussi nous ne préciserons pas le type des objets de la mémoire répartie. Par contre, nous supposerons que ces objets sont statiques (c'est à dire connus à l'initialisation de l'application). Enfin nous ferons abstraction des opérations internes (mettant uniquement en jeu les variables privées d'un processus).

Deux sémantiques sont définies et mises en oeuvre dans la suite du chapitre : la linéarisabilité et la consistance séquentielle. Nous proposons pour la première sémantique plusieurs protocoles dont l'un d'entre eux ne s'applique qu'aux objets propriétaires : chaque objet est modifié par un seul processus et mais peut être lu par tous. Ce cas particulier se rencontre assez fréquemment dans les technologies logicielles (CORBA, JAVA RMI,...). Dans le cas de la consistance séquentielle, nous nous limitons à un protocole s'appliquant aux objets propriétaires. On trouvera dans les références des solutions plus générales.

## 2 La linéarisabilité [Her90]

Dans l'exemple représenté à la figure 6.1 (ainsi que dans la suite) on note $r_i(x):v$ une instance d'appel à `read(x)` qui a renvoyé la valeur `v` et $w_i(x,v)$ une instance d'appel à `write(x,v)`. L'indice nous sert à identifier de manière unique l'instance de l'appel.

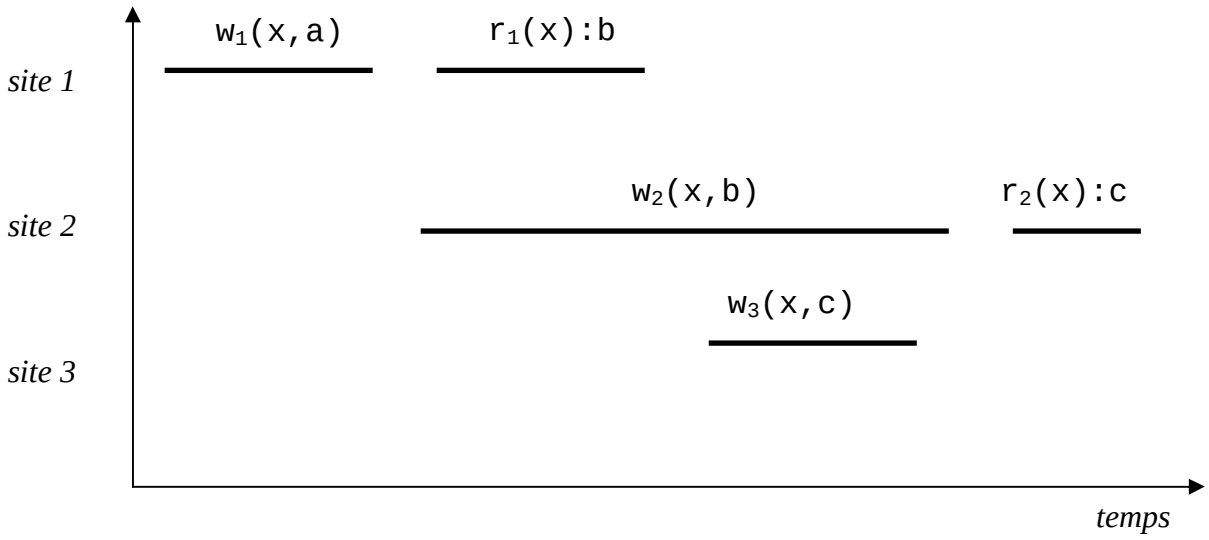



Figure 6.1 : Une exécution linéarisable

Afin d'étudier à quelles conditions l'exécution représentée ci-dessus pourrait être une exécution "acceptable", nous introduisons une définition préalable qui prend en compte le fait que les actions ne sont pas instantanées.

**Définition** *(précédence temporelle)*

Soient deux opérations sur la mémoire répartie $op_1$ et $op_2$. On dit que $op_1$ précède temporellement $op_2$ si et seulement si la fin de l'opération $op_1$ précède le début de l'opération $op_2$. On notera cette précédence temporelle par $op_1 <_t op_2$.

Appelons deux opérations, des opérations concurrentes si aucune d'entre elles ne précède temporellement l'autre. Ainsi il y a concurrence (au sens de cette définition) dès qu'il y a un instant du temps où les deux opérations étaient simultanément en cours d'exécution.

Il est immédiat de vérifier que cette relation est une relation d'ordre et que c'est un ordre partiel. Dans notre exemple on a $r_1(x):b <_t w_3(x,c)$ mais $w_2(x,b)$ et $w_3(x,c)$ sont concurrentes.

Nous sommes maintenant en mesure de définir la notion de linéarisabilité.

**Définition** *(linéarisabilité)*

Une exécution répartie est linéarisable si et seulement si il existe une relation d'ordre total sur les opérations notée < qui étend la précédence temporelle et qui vérifie :

Pour toute lecture $r_i(x):v$, il existe une écriture $w_j(x,v) < r_i(x):v$ et toute autre écriture $w_k(x,v')$ avec $v' \neq v$ vérifie :

- soit $w_k(x,v') < w_j(x,v)$
- soit $r_i(x):v < w_k(x,v')$

Autrement dit, une exécution est linéarisable, si on peut ordonner l'ensemble des opérations en respectant la précédence temporelle et de telle sorte qu'une lecture renvoie la dernière valeur écrite selon cet ordre. L'exécution est de la figure 6.1 est linéarisable et équivalente à l'exécution séquentielle : $w_1(x,a);w_2(x,b);r_1(x):b;w_3(x,c);r_2(x):c$

Nous donnons encore deux exemples d'exécution pour bien fixer les idées.

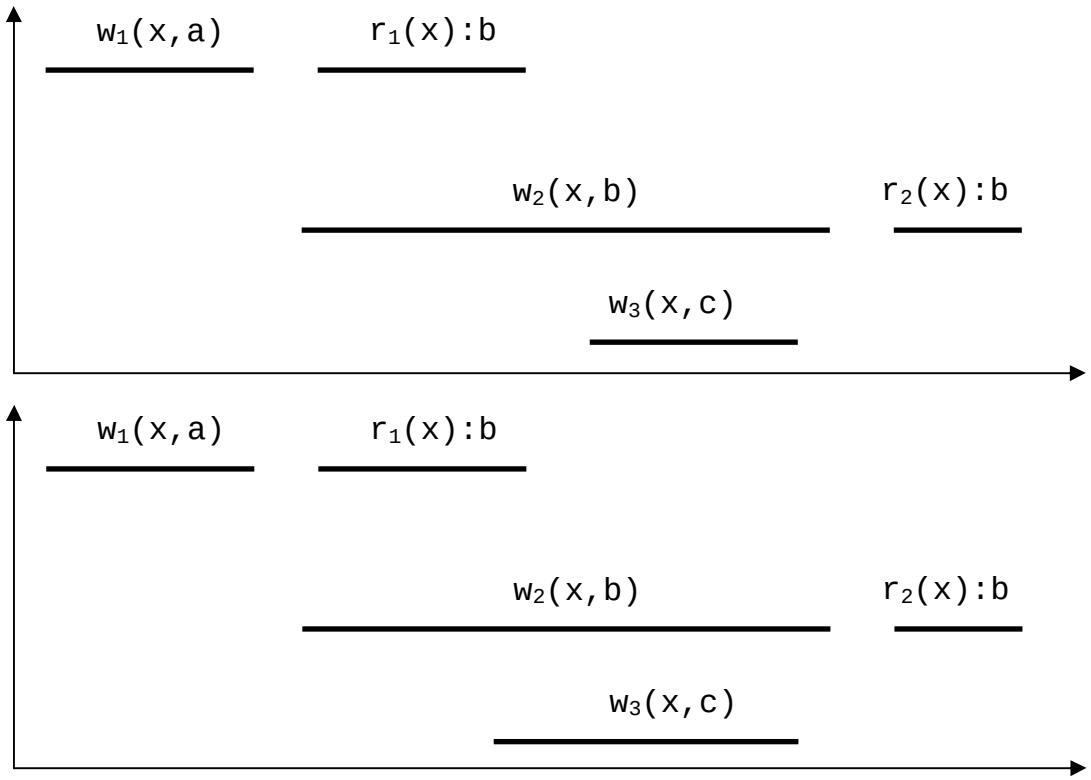



Figure 6.2 : Deux autres exécutions

La première exécution ne peut être linéarisée : $w_2(x,b)$peut être placé avant $r_1(x):b$ et $r_2(x):b$. mais puisque $r_1(x):b <_t w_3(x,c) <_t r_2(x):b$, il n'y a pas de placement possible pour $w_3(x,c)$. La deuxième exécution presque identique à la première n'implique pas la précédence temporelle $r_1(x):b <_t w_3(x,c)$. En conséquence elle est équivalente à l'exécution séquentielle : $w_1(x,a);w_3(x,c);w_2(x,b);r_1(x):b;r_2(x):b$. Il semble paradoxal qu'un "retard non significatif" d'exécution sur un site modifie la possibilité de linéarisation. Ceci nous conduira à une deuxième sémantique de mémoire virtuelle.

# 3 Linéarisabilité par exclusion mutuelle

## *3.1 Principe de l'algorithme*

Avant de présenter le premier protocole qui garantit la linéarisabilité de l'exécution, nous soulignons d'abord une propriété essentielle de cette sémantique qui facilite le développement de protocoles associés. Supposons que nous ayons un protocole qui garantit que, relativement aux opérations sur chaque objet, une linéarisation est possible, alors ce protocole garantit la linéarisation globale.

En effet, si on pose $<_x$ la relation d'ordre total relative aux opérations sur un objet $x$ obtenue par le protocole, on construit une relation d'ordre total sur toutes les opérations comme suit :

- on définit une relation d'ordre total (arbitraire) entre tous les objets $x_1,...,x_n$
- appelons $E$, l'ensemble des opérations, on sélectionne $E_0$ l'ensemble des éléments minimaux de $(E,<_t)$ qu'on ordonne selon deux critères :
    - l'objet sur lequel porte l'opération (d'abord les opérations sur $x_1$, puis sur $x_2$, etc.)
    - pour trier les opérations sur un objet $x$, on suit l'ordre total $<_x$
- on sélectionne ensuite les éléments minimaux de $(E\backslash E_0,<_t)$ et on réitère le procédé.

On construit ainsi par récurrence une relation d'ordre total (car toute opération n'a qu'un nombre fini de prédécesseurs pour $<_t$). Nous laissons au lecteur le soin de vérifier que cette relation vérifie les contraintes de la linéarisation.

La conséquence de ce résultat est qu'en vue de l'obtention de la linéarisation, il suffit de se focaliser sur la gestion de chaque objet de manière indépendante.

Le premier algorithme garantit la linéarisabilité de manière triviale en assurant l'exclusion mutuelle entre les opérations sur un même objet. Ainsi la relation $<_t$ est une relation d'ordre totale sur les opérations de chaque objet et la propriété précédente nous assure la linéarisabilité. N'importe quel algorithme d'exclusion mutuelle convient (voir le chapitre 4). Par souci de simplicité, nous organisons les sites en anneau, et nous utilisons l'algorithme à jetons avec un jeton qui circule par objet et contient son identité et sa valeur.

## *3.2 Description*

Variables du site `i`

- $\texttt{suivant}_i$ : constante contenant l'identité du site successeur de `i` sur l'anneau.
- $\texttt{val}_i$ : variable contenant une valeur transmise par un jeton
- $\texttt{attendu}_i$ : variable contenant l'identité de l'objet dont attend le jeton. Cette variable est initialisée à `NULL` indiquant qu'on n'attend pas de jeton.

Algorithme du site `i`

Pour lire la valeur d'un objet, on attend le jeton et on renvoie la valeur après avoir envoyé le jeton au suivant sur l'anneau.

```
read(x)
Début
     attendu_i=x;
     Attendre(attendu_i==NULL);
     envoyer_à(suivant_i,(x,val_i));
     renvoyer(val_i);
Fin
```

Pour modifier un objet, on attend le jeton et on envoie le jeton au suivant sur l'anneau après avoir modifié sa valeur.

```
write(x,v)
Début
     attendu_i=x;
     Attendre(attendu_i==NULL);
     envoyer_à(suivant_i,(x,v));
Fin
```

A la réception d'un jeton, si l'objet associé n'était pas attendu, on envoie le jeton au suivant sur l'anneau sinon on indique que le jeton est arrivé et la valeur de cet objet.

```
sur_réception_de(j,(x,v))
Début
     Si (attendu_i != x) Alors
          envoyer_à(suivant_i,(x,v));
     Sinon
          val_i = v;
          attendu_i=NULL;
     Finsi
Fin
```

# 4 Lecture à la demande

## *4.1 Principe et réalisation de l'algorithme*

Dans cet algorithme, nous faisons l'hypothèse que les objets ne sont modifiés que par un seul site appelé le propriétaire de l'objet. Dans ce cas particulier, il existe une solution particulièrement simple pour assurer la linéarisabilité. Lorsqu'un site appelle la primitive `read()` sur un objet distant, il envoie une demande de valeur au site propriétaire qui à la réception lui envoie la valeur demandée. La mise en oeuvre de cet algorithme décrite ci-dessous ne présente pas de difficultés majeures. Cet algorithme est présenté afin de servir de base à une version auto-stabilisante (voir chapitre 8).

Variables du site `i`

- $val_i$ : variable temporaire qui contient la valeur demandée lors d'un `read()` distant.
- $x_i$ : variable associée à l'objet partagée `x`. Le site `i` a autant de variables de ce type que d'objets dont il est propriétaire.
- $état_i$ : état du service à valeurs dans (`repos`, `attente`) initialisé à `repos`.
- $prop_i$`[Objets]` : constante qui contient l'identité du propriétaire de chaque objet.

Algorithme du site `i`

A la lecture d'un objet, soit on est le propriétaire et on renvoie immédiatement la valeur, soit on envoie une demande au propriétaire et on renvoie la valeur fournie par le propriétaire.

```
read(x)
Début
     Si prop_i[x]==i Alors
          renvoyer(x_i);
     Sinon
          état_i=attente;
          envoyer_à(prop_i[x],(req,x));
          Attendre(état_i==repos);
          renvoyer(val_i);
     Finsi
Fin
```

La modification d'un objet est purement locale.

```
write(x,v)
Début
     x_i = v;
Fin
```

A la réception d'une requête, on renvoie la valeur demandée.

```
sur_réception_de(j,(req,x))
Début
     envoyer_à(j,(acq,x_i));
Fin
```

A l'acquittement, on enregistre la valeur et on modifie l'état du service.

```
sur_réception_de(j,(acq,v))
Début
     val_i = v;
     état_i=repos;
Fin
```

### *6.2 Correction de l'algorithme*

La correction de l'algorithme s'établit en construisant la relation d'ordre total < qui permet la linéarisation. Cette relation se définit à l'aide d'instants associés à chaque opération :

- Soit `op` une opération d'écriture, `t(op)` est l'instant de fin de l'opération
- Soit `op` une opération de lecture locale, `t(op)` est l'instant de fin de l'opération
- Soit `op` une opération de lecture distante, `t(op)` est l'instant où le résultat est renvoyé par le site propriétaire.

Appelons aussi `i(op)` l'identité du site qui exécute l'opération `op`. Comme nous l'avons fait à plusieurs reprises dans ce cours, on pose :

$$(t,i)<(t',i') \Leftrightarrow t<t' \text{ ou } (t=t') \text{ et } (i<i').$$

La relation précédente est un ordre total. Nous sommes maintenant en mesure de définir la relation d'ordre total entre opérations.

$$op_1<op_2 \Leftrightarrow (t(op_1),i(op_1))<(t(op_2),i(op_2))$$

Vérifions que les contraintes sont bien respectées par cet ordre :

1. C'est un ordre total car la fonction $op \rightarrow (t(op),i(op))$ est injective.
2. Il étend la précédence temporelle car l'instant choisi pour une opération est compris entre le début et la fin de l'exécution de celle-ci.
3. La consistance d'une lecture locale est trivialement assurée puisqu'elle suit temporellement l'écriture dont elle renvoie la valeur et précède temporellement l'écriture suivante.

La consistance d'une lecture distante s'obtient en considérant l'instant choisi. En effet, il s'agit d'un instant mesuré sur le site propriétaire qui suit la fin de l'écriture dont la lecture renvoie la valeur (en raison de l'ininterruptibilité des primitives de service) et qui précède le début - et par conséquent la fin - de l'écriture suivante (pour la même raison).

# 5 Linéarisabilité avec gestionnaires statiques [Li89]

## *5.1 Principe de l'algorithme*

On constate que l'algorithme précédent interdit des lectures simultanées alors que ceci n'empêche en rien la linéarisabilité. Nous allons donc mettre en œuvre une gestion des lectures concurrentes.

Pour simplifier, nous supposons qu'un site peut s'envoyer un message à lui-même (c'est d'ailleurs les cas des primitives de programmation réseau). Dans cet algorithme et le suivant, les objets sont mobiles et éventuellement dupliqués (cas des lectures simultanées). Il faut donc un moyen pour un site de repérer l'objet. Ici nous supposons qu'à chaque objet est associé un gestionnaire statique par objet connu de tous. Son identité peut se calculer par exemple à l'aide d'une fonction de hachage appliquée à l'identité de l'objet. Comme ce point n'est pas essentiel à l'algorithme, nous supposons que chaque site dispose d'une table qui lui indique pour chaque objet son gestionnaire ($gest_i$).

Les informations qu'un gestionnaire possède sur un objet `x` sont contenues dans la cellule $infos_i$`[x]`. Elle comporte d'abord l'identité du site qui acquis l'objet le plus anciennement appelé abusivement le propriétaire ($infos_i$`[x].prop`), puis l'ensemble des autres sites qui ont une copie de l'objet ($infos_i$`[x].ontcopie`). Enfin puisque des requêtes concurrentes peuvent parvenir au gestionnaire, celui dispose d'une file qui contient les requêtes à traiter dont la requête en cours ($infos_i$`[x].àtraiter`).

Chaque site dispose d'une mémoire virtuelle locale qui peut potentiellement contenir l'ensemble des objets du système ($mem_i$`[x].val`). Cependant chaque site doit connaître le statut de la cellule relatif à chaque objet : objet invalide, accès en lecture seule ou accès en lecture-écriture ($mem_i$`[x].statut`).

Le lecteur aura compris que la linéarisabilité est assurée par le recours au gestionnaire (rappelons qu'il suffit que les accès à chaque objet soient linéarisables indépendamment des accès aux autres objets). Détaillons maintenant le flux des messages engendrés par une lecture et par une écriture lorsque le site ne dispose pas du droit correspondant.

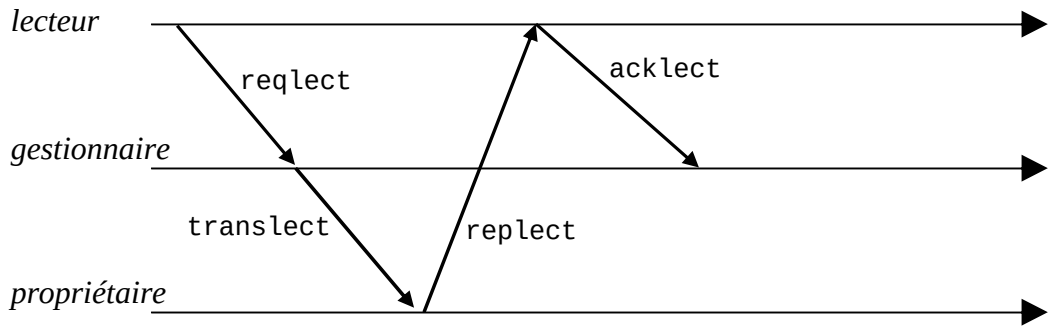



Figure 6.3 : Flux des messages lors d'une requête de lecture

La figure 6.3 décrit le cas d'une lecture. Un site qui désire obtenir le droit de lire (et incidemment la valeur à lire) envoie une requête de lecture (`reqlect`) au gestionnaire. Si celui-ci ne traite pas une autre requête sur cet objet, il transmet la requête au propriétaire qui modifie son accès à l'objet (lecture seule) et renvoie au lecteur l'objet (`replect`). A la réception de l'objet, celui-ci indique au gestionnaire qu'il dispose de la copie (`acklect`). Après mise à jour des informations, le gestionnaire peut éventuellement traiter une autre requête sur cet objet.

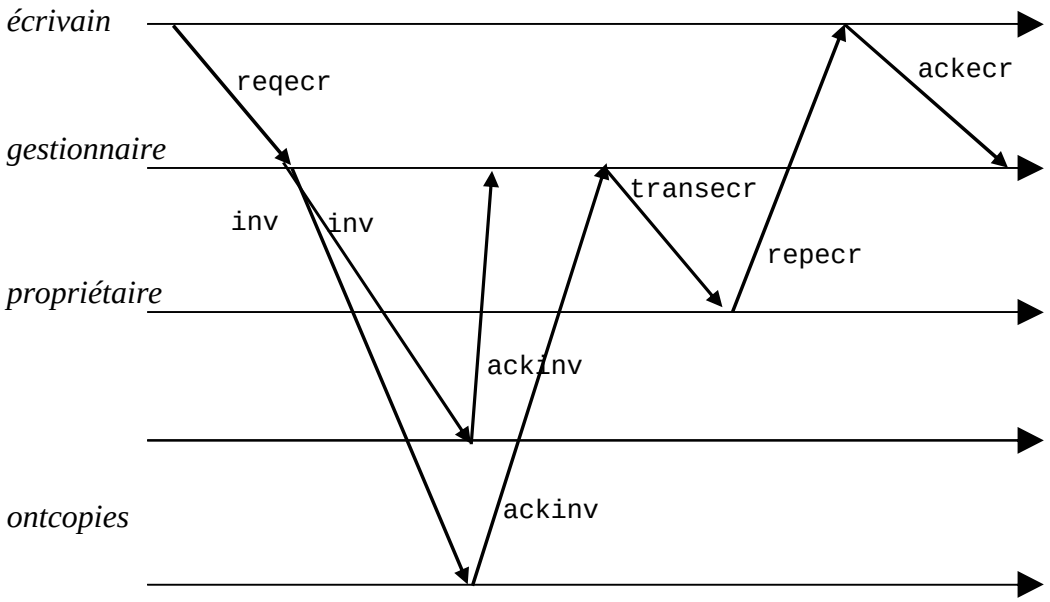



Figure 6.4 : Flux des messages lors d'une requête d'écriture

La figure 6.4 décrit le cas d'une écriture. Un site qui désire obtenir le droit d'écrire envoie une requête d'écriture (`reqecr`) au gestionnaire. Si celui-ci ne traite pas une autre requête sur cet objet, il envoie des invalidations aux sites qui possèdent la copie (`inv`) et attend leur acquittement (`ackinv`). Puis il transmet la requête au propriétaire qui invalide son accès à l'objet et renvoie à l'écrivain le droit d'accès (`repecr`). A la réception de celui-ci, le demandeur indique au gestionnaire qu'il dispose du droit d'écrire (`ackecr`). Après mise à jour des informations, le gestionnaire peut éventuellement traiter une autre requête sur cet objet.

### *5.2 Description*

Variables du site `i`

- $\texttt{gest}_i\texttt{[Objets]}$ : constante qui contient l'identité du gérant de chaque objet.
- $\texttt{Objets}_i$ : ensemble constant des objets gérés par `i`.
- $\texttt{infos}_i\texttt{[Objets}_i\texttt{]}$ de
  - `prop` : propriétaire courant de l'objet initialement `i`
  - `ontcopie` : ensembles des autres sites qui ont une copie de l'objet initialement vide
  - `àtraiter` : file de couples `<site,requête>` initialement vide

  (primitives associées :`enfiler()`,`défiler()`,`tête()`)

- $mem_i$[Objets] de
  - `val` : valeur de l'objet initialisé à la valeur initiale de l'objet s'il est affecté à `i`
  - `statut` : (`invalide`,`lect`,`ecr`) statut de l'objet, `ecr` pour un objet affecté, `invalide` pour un objet non affecté.
- $temp_i$ : variable temporaire d'identité de site
- $req_i$ : variable temporaire de type de requête

Algorithme du site `i`

En cas de lecture d'un objet invalide, on émet une requête au gestionnaire de l'objet et on attend que l'objet soit disponible en lecture.

```
read(x)
Début
     Si (mem_i[x].statut==invalide) Alors
          envoyer_à(gest_i[x],(reqlect,x));
          Attendre(mem_i[x].statut==lect);
     Fin si
     renvoyer(mem_i[x].val);
Fin
```

A la réception d'une requête de lecture, si une requête n'est pas en cours on transfère la requête au propriétaire courant de l'objet. Dans tous les cas, on met à jour la file des requêtes à traiter.

```
sur_réception_de(j,(reqlect,x))
Début
     Si infos_i[x].àtraiter==vide Alors
          envoyer_à(infos_i[x].prop,(translect,j,x));
     Finsi
     enfiler(infos_i[x].àtraiter,<j,reqlect>);
Fin
```

A la réception d'un transfert d'une requête de lecture, le propriétaire de l'objet envoie une réponse au demandeur et met à jour le statut en lecture.

```
sur_réception_de(j,(translect,k,x))
Début
     mem_i[x].statut=lect ;
     envoyer_à(k,(replect,x,mem_i[x].val));
Fin
```

A la réception de la réponse de lecture, la valeur et le statut de cet objet sont mis à jour et un acquittement est envoyé au gestionnaire.

```
sur_réception_de(j,(replect,x,v))
Début
     mem_i[x].val=v;
     mem_i[x].statut=lect;
     envoyer_à(gest_i[x],(acklect,x));
Fin
```

A la réception de l'acquittement de lecture le nouveau lecteur est ajouté aux possesseurs d'une copie. Puis après avoir défilé la requête courante on traite la requête suivante s'il y en a une. à l'aide de la primitive `finrequête`.

```
sur_réception_de(j,(acklect,x))
Début
     infos_i[x].ontcopie = infos_i[x].ontcopie ∪ {j};
     finrequête(x);
Fin

finrequête(x)
Début
     défiler(infos_i[x].àtraiter);
     Si infos_i[x].àtraiter!=vide Alors
          <temp_i,req_i>=tête(infos_i[x].àtraiter);
          Si req_i==reqlect Alors
               envoyer_à(infos_i[x].prop,(translect,temp_i,x));
          Sinon
               traiterécriture(temp_i,x);
          Finsi
     Finsi
Fin
```

Pour modifier un objet dont on ne possède pas le droit en écriture, on envoie une requête d'écriture et on attend le droit d'écriture. Finalement on écrit la valeur prévue.

```
write(x,v)
Début
     Si (mem_i[x].statut!=ecr) Alors
          envoyer_à(gest_i[x],(reqecr,x));
          Attendre(mem_i[x].statut==ecr);
     Fin si
     mem_i[x].val=v;
Fin
```

A la réception d'une requête d'écriture, si une requête n'est pas en cours on appelle la primitive `traiterécriture()`. Dans tous les cas, on met à jour la file des requêtes à traiter.

```
sur_réception_de(j,(reqecr,x))
Début
     Si infos_i[x].àtraiter==vide Alors
          traiterécriture(j,x);
     Finsi
     enfiler(infos_i[x].àtraiter,<j,reqecr>);
Fin
```

Le traitement d'une écriture consiste d'abord à envoyer à un message d'invalidation à tous les possesseurs d'une copie excepté éventuellement le demandeur. S'il n'y en a pas, on transfère directement la requête d'écriture au propriétaire.

**traiterécriture(j,x)**

```
Début
     Si infos_i[x].ontcopie\{j} != ∅ Alors
          Pour temp_i ∈ infos_i[x].ontcopie\{j} faire
               envoyer_à(temp_i,(inv,x));
          Fin pour
     Sinon
          envoyer_à(infos_i[x].prop,(transecr,j,x));
     Finsi
Fin
```

A la réception d'une invalidation, l'objet est invalidé et un acquittement est envoyé au gestionnaire.

**sur_réception_de(j,(inv,x))**

```
Début
     mem_i[x].statut=invalide;envoyer_à(j,(ackinv,x));
Fin
```

A la réception d'un acquittement d'invalidation l'émetteur est extrait des possesseurs de copies. S'il n'y a plus de possesseurs autre qu'éventuellement le demandeur, la requête est transférée au propriétaire.

**sur_réception_de(j,(ackinv,x))**

```
Début
     <temp_i,req_i>=tête(infos_i[x].àtraiter);
     infos_i[x].ontcopie = infos_i[x].ontcopie\{j};
     Si infos_i[x].ontcopie\{temp_i} == ∅ Alors
          envoyer_à(infos_i[x].prop,(transecr,temp_i,x));
     Finsi
Fin
```

A la réception d'un transfert d'une requête d'écriture, le propriétaire de l'objet envoie une réponse au demandeur et met à jour le statut à invalide.

**sur_réception_de(j,(transecr,k,x))**

```
Début
     mem_i[x].statut=invalide;envoyer_à(k,(repecr,x));
Fin
```

A la réception de la réponse de lecture, le statut de cet objet est mis à jour et un acquittement est envoyé au gestionnaire.

**sur_réception_de(j,(repecr,x))**

```
Début
     mem_i[x].statut=ecr;envoyer_à(gest_i[x],(ackecr,x));
Fin
```

A la réception de l'acquittement d'écriture, le demandeur est le nouveau propriétaire de l'objet et personne n'en possède une copie. Puis après avoir défilé la requête courante, on traite la requête suivante s'il y en a une.

```
sur_réception_de(j,(ackecr,x))
Début
     infos_i[x].ontcopie = ∅;
     infos_i[x].prop = j;
     finrequête(x);
Fin
```

# 6 Linéarisabilité avec gestionnaires dynamiques [Li89]

Une alternative à l'utilisation de gestionnaires statiques consiste à identifier le propriétaire et le gestionnaire. Mais dans ce cas, comment atteindre ce gestionnaire qui varie au cours du temps ? L'astuce consiste à ce que chaque site ait une variable contenant l'identité supposée du gestionnaire. Cette identité est initialement correcte puis lorsqu'elle change, l'ancien propriétaire modifie de manière adéquate sa variable. Ainsi par indirection, on arrive toujours au propriétaire réel. Autrement dit, l'ensemble des liens induits par ces variables forme un arbre dont la racine est le propriétaire courant.

Les variables sont similaires à celles de l'algorithme précédent excepté que tout site peut être gestionnaire de tout objet et que l'identité du propriétaire n'a plus besoin d'être stockée.

Hormis la recherche du gestionnaire, le protocole se trouve simplifié.

## *6.1 Description*

Variables du site `i`

- $\texttt{gestprob}_i\texttt{[Objets]}$ : identité du gérant probable de chaque objet initialisé au propriétaire initial de chaque objet.
- $\texttt{infos}_i\texttt{[Objets] de}$
  - `ontcopie` : ensembles des autres sites qui ont une copie de l'objet initiale vide
  - `àtraiter` : file de couples `<site,requête>` initialement vide
  (primitives associées :`enfiler()`,`défiler()`,`tête()`)
- $\texttt{mem}_i\texttt{[Objets] de}$
  - `val` : valeur de l'objet initialisé à la valeur initiale de l'objet s'il est initialement possédé par `i`
  - `statut` : (`invalide`,`lect`,`ecr`) statut de l'objet, `ecr` pour un objet affecté, `invalide` pour un objet non affecté. On suppose que les trois valeurs sont ordonnées.
- $\texttt{temp}_i$ : variable temporaire d'identité de site
- $\texttt{req}_i$ : variable temporaire de type de requête

Algorithme du site `i`

En cas de lecture d'un objet invalide, on émet une requête au gestionnaire probable de l'objet et on attend que l'objet soit disponible en lecture.

```
read(x)
Début
     Si (mem_i[x].statut==invalide) Alors
          envoyer_à(gestprob_i[x],(i,reqlect,x));
          Attendre(mem_i[x].statut==lect);
     Fin si
     renvoyer(mem_i[x].val);
Fin
```

A la réception d'une requête de lecture, si une requête n'est pas en cours, alors soit on n'est pas le gestionnaire et on transfère la requête au gestionnaire probable de l'objet, soit on est le gestionnaire. Dans ce cas, on renvoie la copie de l'objet et on met à jour le champ `ont_copie` et le statut à `lect`. Sinon on met à jour la file des requêtes à traiter.

```
sur_réception_de(j,(k,reqlect,x))
Début
     Si infos_i[x].àtraiter==vide Alors
          Si gestprob_i[x]!=i Alors
               envoyer_à(gestprob_i[x],(k,reqlect,x));
          Sinon
               envoyer_à(k,(replect,x,mem_i[x].val));
               mem_i[x].statut=lect ;
               infos_i[x].ontcopie = infos_i[x].ontcopie ∪ {k};
          Finsi
     Sinon
          enfiler(infos_i[x].àtraiter,<k,reqlect>);
     Finsi
Fin
```

A la réception de la réponse de lecture, la valeur, le statut et le gestionnaire probable de cet objet sont mis à jour.

```
sur_réception_de(j,(replect,x,v))
Début
     mem_i[x].val=v; mem_i[x].statut=lect; gestprob_i[x]=j;
Fin
```

Pour modifier un objet dont on ne possède pas le droit en écriture, on envoie une requête d'écriture et on attend le droit d'écriture. Après l'obtention de ce droit, on écrit la valeur prévue. Si on était déjà proriétaire d'autres requêtes ont pu être mises en attente et on traite la première d'entre elles.

```
write(x,v)
Début
     Si (mem_i[x].statut!=ecr) Alors
          envoyer_à(gestprob_i[x],(i,reqecr,x));
          Attendre(mem_i[x].statut==ecr);
     Fin si
     mem_i[x].val=v;
     Si infos_i[x].àtraiter!=vide Alors
          <temp_i,req_i>=tête(infos_i[x].àtraiter);
          défiler(infos_i[x].àtraiter);
          Si req_i==reqlect Alors
               envoyer_à(temp_i,(replect,x,mem_i[x].val));
               mem_i[x].statut=lect ;
               infos_i[x].ontcopie=infos_i[x].ontcopie ∪ {temp_i};
          Sinon
               traiterécriture(temp_i,x);
          Fin si
     Fin si
Fin
```

A la réception d'une requête d'écriture, si une requête n'est pas en cours soit on transmet au gestionnaire probable (si différent du site) soit on appelle la primitive `traiterécriture()`. Sinon on met à jour la file des requêtes à traiter.

```
sur_réception_de(j,(k,reqecr,x))
Début
     Si infos_i[x].àtraiter==vide Alors
          Si gestprob_i[x]!=i Alors
               envoyer_à(gestprob_i[x],(k,reqecr,x));
          Sinon
               traiterécriture(k,x);
          Finsi
     Sinon
          enfiler(infos_i[x].àtraiter,<j,reqecr>);
     Finsi
Fin
```

Le traitement d'une écriture consiste d'abord à envoyer à un message d'invalidation à tous les possesseurs d'une copie excepté éventuellement le demandeur. S'il n'y en a pas, on change car la requête émane nécessairement d'un autre site (cf les commentaires du programme).

```
traiterécriture(j,x)
Début
     Si infos_i[x].ontcopie\{j} != ∅ Alors
          enfiler(infos_i[x].àtraiter,<j,reqecr>);
          Pour temp_i ∈ infos_i[x].ontcopie\{j} faire
               envoyer_à(temp_i,(inv,x));
          Fin pour
     Sinon
          // nécessairement j!=i car sinon le statut de
          // x pour i serait positionné à ecr
          // car ontcopie != ∅ est équivalent à statut==lect
          changeprop(j,x);
     Finsi
Fin
```

A la réception d'une invalidation, l'objet est invalidé et un acquittement est envoyé au gestionnaire.

```
sur_réception_de(j,(inv,x))
Début
     mem_i[x].statut=invalide;
     envoyer_à(j,(ackinv,x));
Fin
```

A la réception d'un acquittement d'invalidation, l'émetteur est extrait des possesseurs de copies. S'il n'y a plus de possesseur autre qu'éventuellement le demandeur, la requête d'écriture peut être satisfaite. Dans le cas d'une requête locale, il suffit de mettre à jour le statut. Dans le cas d'une requête distante, on change de propriétaire.

```
sur_réception_de(j,(ackinv,x))
Début
     <temp_i,req_i>=tête(infos_i[x].àtraiter);
     infos_i[x].ontcopie = infos_i[x].ontcopie\{j};
     Si infos_i[x].ontcopie\{temp_i} == ∅ Alors
          Si temp_i!=i Alors
               défiler(infos_i[x].àtraiter);
               changeprop(temp_i,x);
          Sinon
               mem_i[x].statut=ecr;
          Fin si
     Finsi
Fin
```

En cas de changement de propriétaire, on transmet la réponse au demandeur, on met à jour son statut puis on lui retransmet les requêtes mises en attente

```
changeprop(j,x);
Début
     envoyer_à(j,(repecr,x));
     mem_i[x].statut=invalide;
     gestprob_i[x]=j;
     // àtraiter peut être différent de vide si la
     // requête précédente (d'écriture) émane de i
     // alors qu'il était déjà propriétaire
     Tant que infos_i[x].àtraiter!=vide faire
          <temp_i,req_i>=tête(infos_i[x].àtraiter);
          défiler(infos_i[x].àtraiter);
          envoyer_à(j,(temp_i,req_i,x));
     Fin tant que
Fin
```

A la réception de la réponse de lecture, le statut de cet objet est mis à jour ainsi que les informations de gestion..

```
sur_réception_de(j,(repecr,x))
Début
     mem_i[x].statut=ecr;
     infos_i[x].ontcopie=∅;
     infos_i[x].àtraiter=vide;
Fin
```

## 7 La consistance séquentielle [Att91] [Miz92]

Si nous faisons le bilan des protocoles associés à la linéraisabilité, nous remarquons que :

- L'exclusion mutuelle entre les opération (e.g. par un jeton) a une conception simple mais est d'une grande inefficacité
- La lecture à la demande ne s'applique qu'aux objets propriétaires. Elle autorise le parallélisme des lectures. Mais chaque lecture requiert deux envois de message consécutifs.
- Les solutions avec gestionnaire permettent le parallélisme des lectures, autorisent le cache des données évitant tant que faire se peut les envois de message mais la gestion des écritures est relativement coûteuse.

D'autre part, l'exemple de la figure 6.2 montre que la linéarisabilité dépend de la vitesse relative d'exécution sur chacun des sites. Or (voir le chapitre 5), ce facteur n'est pas significatif dans une application répartie dont le modèle de communication est l'envoi de messages. On cherche donc à définir une sémantique de même nature pour les accès à la mémoire virtuelle. C'est l'objectif de la consistance séquentielle. De plus les protocoles associés devraient être plus efficaces puisque les contraintes sur l'exécution seront relachées.

Nous restreignons d'abord la précédence temporelle aux opérations exécutées sur un même site (appelée précédence locale).

**Définition** *(précédence locale)*

Soient deux opérations sur la mémoire répartie $op_1$ et $op_2$. On dit que $op_1$ précède localement $op_2$ si et seulement si les deux opérations sont exécutées sur le même site et $op_1 <_t op_2$. On note cette relation d'ordre $op_1 <_l op_2$.

Nous sommes maintenant en mesure de définir la consistance séquentielle.

**Définition** *(consistance séquentielle)*

Une exécution répartie est séquentiellement consistante si et seulement si il existe une relation d'ordre total sur les opérations notée < qui étend la précédence locale et qui vérifie :

Pour toute lecture $r_i(x):v$, il existe une écriture $w_j(x,v) < r_i(x):v$ qui vérifie :

1. toute autre écriture $w_k(x,v')$ avec $v' \neq v$ vérifie :
   - soit $w_k(x,v') < w_j(x,v)$
   - soit $r_i(x):v < w_k(x,v')$
2. NOT $r_i(x):v <_t w_j(x,v)$

Autrement dit, une exécution est séquentiellement consistante, si on peut ordonner l'ensemble des opérations en respectant la précédence locale et de telle sorte qu'une lecture renvoie la dernière valeur écrite selon cet ordre.
De plus, cette écriture ne peut suivre temporellement la lecture. En effet cela signifierait que la couche service mémoire dispose d'un devin capable de connaître le comportement futur de l'application !

On retrouve donc ici une relation analogue à la dépendance causale du chapitre 2. Un événement dépend causalement d'un autre événement par fermeture transitive de la relation suivante.
Le second événement ne suit pas temporellement le premier et

- soit ils sont exécutés sur le même site (auquel cas il y a précédence temporelle)
- soit le premier est une lecture et le second la dernière écriture sur le même objet avant cette lecture (pour la séquentialisation <)

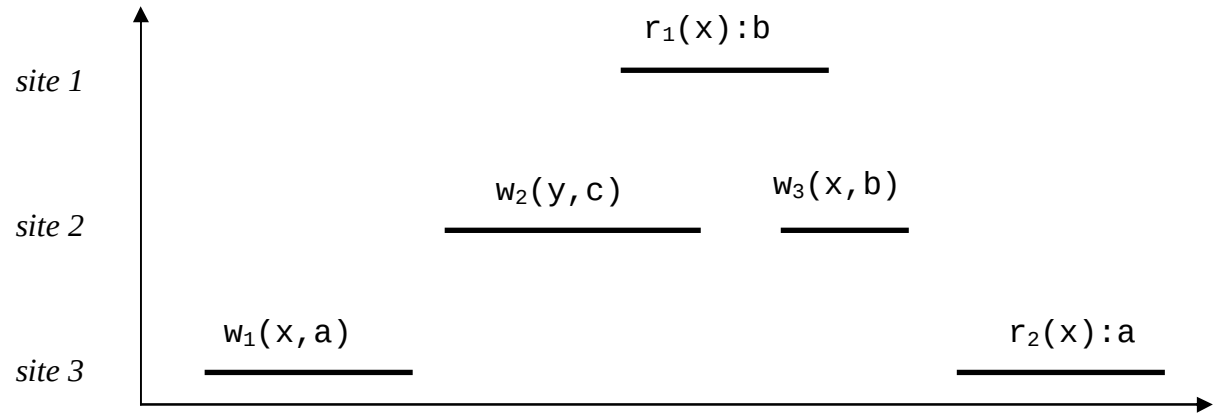



Figure 6.5 : Une exécution non linéarisable mais séquentiellement consistante

L'exécution de la figure 6.5 ne peut être linéarisée car :

$w_1(x,a) <_t w_3(x,b) <_t r_2(x):a$

Cependant cette exécution est séquentiellement consistante car on peut ordonner les opérations en préservant la précédence locale de la manière suivante :

$w_1(x,a);r_2(x):a;w_2(y,c);w_3(x,b);r_1(x):b$

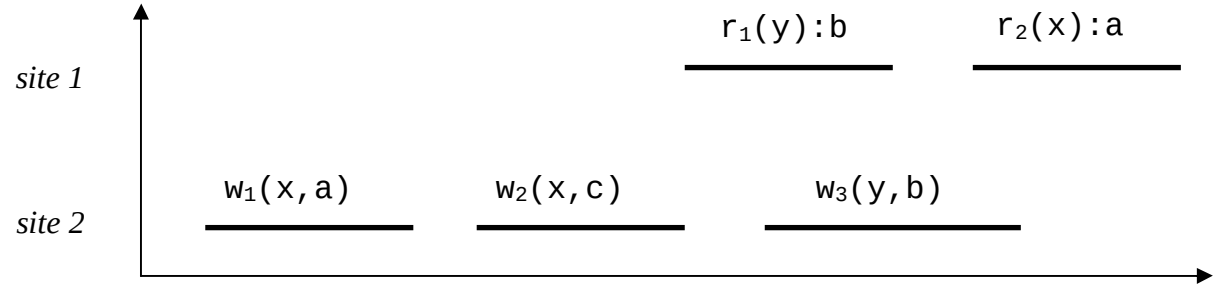



Figure 6.6 : Une exécution non séquentiellement consistante

L'exécution de la figure 6.6 n'est pas séquentiellement consistante car une exécution séquentielle équivalente devrait vérifier :

- $w_2(x,c) < w_3(y,b)$ *(précédence locale)*
- $w_3(y,b) < r_2(y):b$ *(consistance des lectures)*
- $r_2(y):b < r_2(x):a$ *(précédence locale)*

D'où par transitivité : $w_2(x,c) < r_2(x):a$
Mais en raison de la consistance des lectures on doit avoir : $r_2(x):a < w_2(x,c)$
Ce qui est contradictoire.

Il est intéressant de noter que, relativement à chaque objet, cette exécution est séquentiellement consistante. On peut ainsi ordonner $w_3(y,b);r_2(y):b$ et $w_1(x,a);r_2(x):a;w_2(x,c)$. Cependant la fusion de ces ordres relatifs est impossible. Il apparaît donc qu'un protocole de consistance séquentielle permettra des exécutions plus efficaces mais que le protocole sera plus complexe à définir car il devra gérer les interactions des opérations entre objets différents.

# 8 Propagation des écritures

## *8.1 Principe de l'algorithme*

Nous supposons à nouveau que les objets ne sont modifiés que par leur "propriétaire". Nous supposons de plus que chaque site commence son exécution par une écriture (au besoin la couche service peut ajouter une écriture sans effet au premier appel à `read()` si aucune écriture n'a été effectuée).

L'objectif de cet algorithme est d'anticiper les lectures distantes par diffusion de chaque modification d'un objet partagé par le propriétaire. Cependant il faut prendre des précautions pour obtenir une exécution répartie qui soit séquentiellement consistante. Ceci repose sur différents mécanismes.

Tout d'abord chaque site gère une horloge logique (voir le chapitre 3) pour dater les écritures sur les objets partagés. Conformément au fonctionnement de ces horloges, celles-ci s'incrémentent à chaque nouvelle écriture et d'autre part leur gestion garantit qu'une écriture distante dont un site est au courant a une heure plus petite que les futures écritures de ce site.

Le squelette de la séquentialisation repose sur un ordre total entre les écritures obtenu en comparant lexicographiquement les doublons (heure,identité du site) associés aux écritures. Les lectures sur un site qui s'intercalent entre deux écritures locales sont "placées" immédiatement après l'occurence de la première écriture.

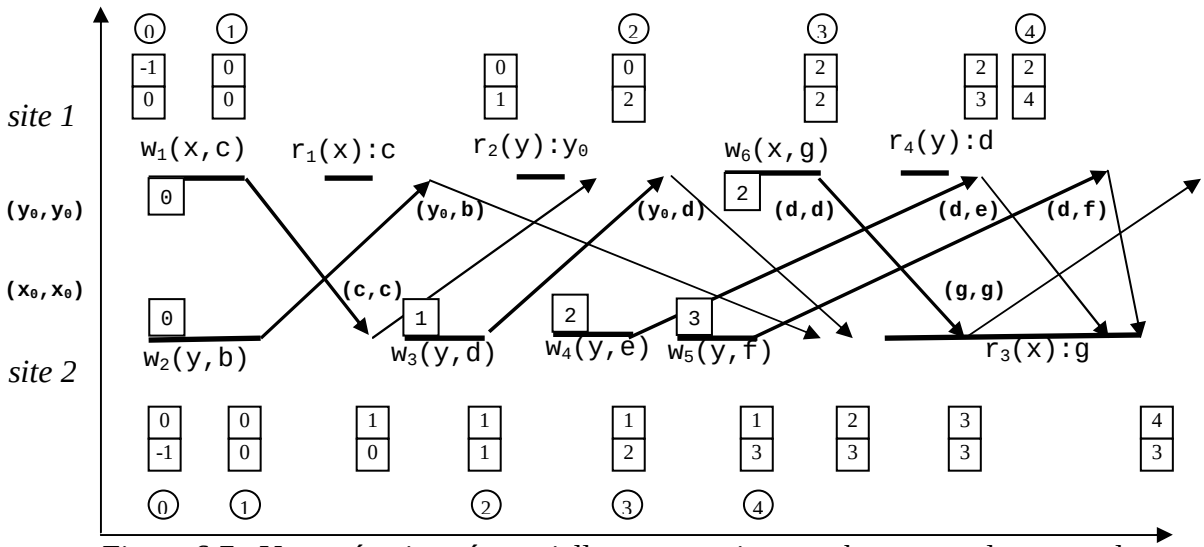



Figure 6.7 : Une exécution séquentiellement consistante obtenue par le protocole

Dans la figure 6.7, les heures sont notées en encadrés, les flèches grasses indiquent les propagations des mises à jour et les flèches fines les acquittements. L'ordre des écritures est le suivant :

$$w_1(x,c);w_2(y,b);w_3(y,d);w_6(x,g);w_4(y,e);w_5(y,f)$$

En intercalant les lectures immédiatement après la dernière écriture locale, on obtient :
$w_1(x,c);\ r_1(x):c;\ r_2(y):y_0;\ w_2(y,b);\ w_3(y,d);\ w_6(x,g);\ r_4(y):d;$
$w_4(y,e);\ w_5(y,f);\ r_3(x):g$

Il convient donc d'assurer la consistance des lectures. Ceci s'obtient de la façon suivante. Tout d'abord une lecture ne renvoie la valeur d'un objet distant que si elle sait que les écritures sur le site distant "suivront" la dernière écriture locale pour l'ordre précisé plus haut.

Ainsi dans l'exécution de la figure 6.7 :
$r_3(x)$ ne renvoie `g` qu'à la réception de l'acquittement de la modification $w_5(y,f)$ car le site `2` sait à cet instant que la prochaine écriture sur le site `1` sera datée d'au moins `(4,1)`.

Cette attente ne peut être indéfinie car tous les sites doivent acquitter la propagation de l'écriture qui a précédé la lecture et (au pire) à la réception de cet acquittement, on est assuré que la condition est vérifiée. C'est exactement le cas pour la lecture $r_3(x):g$.

Le dernier point délicat est donc la gestion des valeurs reçues lors de la propagation des écritures. A cet effet, deux variables sont utilisées par objet distant : l'une contient la dernière valeur reçue et l'autre la dernière valeur reçue qui peut être renvoyé comme résultat d'une lecture. A la réception d'une valeur, à l'aide des horloges logiques on sait si la valeur peut être renvoyée comme résultat.

Ainsi dans l'exécution de la figure 6.7 :
$w_2(y,b)$ puis $w_3(y,d)$ ne sont pas considérées comme des valeurs acceptables à leur réception sur le site `1` car leur heure est trop grande mais elles sont stockées successivement comme dernière valeur reçue.

D'autre part à chaque nouvelle écriture les dernières valeurs reçues deviennent "acceptables".

Ainsi dans l'exécution de la figure 6.7 :
$w_3(y,d)$ devient une valeur acceptable après l'écriture $w_6(x,g)$.

### *8.2 Réalisation de l'algorithme*

Variables du site `i`

- $val_i$`[Objets]` : tableau de valeurs indicé par les objets. Ces valeurs serviront comme résultat d'un appel à `read()`. Chaque cellule est initialisée à une valeur par défaut commune à tous les sites mais qui est spécifique à chaque objet.
- $der_i$`[Objets]`: tableau de valeurs indicé par les objets. Ces valeurs sont les dernières valeurs connues de l'objet correspondant. Ce tableau est initialisé de la même façon que le tableau précédent. Il ne sert en fait qu'aux objets "distants".
- $h_i$ : heure locale de `i` initialisée à 0.
- $hor_i$`[1..N]` : tableau d'heures indicé par les sites initialisé à $hor_i$`[j]=0` pour `j`$\neq$`i` et $hor_i$`[i]=-1`. Dans le cas d'un site différent de i la cellule fournit un minorant de l'heure de la prochaine écriture. Dans le cas de i, la cellule fournit l'heure de la dernière écriture.
- $prop_i$`[Objets]` : constante qui contient l'identité du propriétaire de chaque objet.

Algorithme du site `i`

A la lecture d'un objet, soit on est le propriétaire et on renvoie immédiatement la valeur, soit on attend d'être sûr que les prochaines écritures du propriétaire de l'objet "suivent" la dernière écriture sur `i`.

```
read(x)
Début
     Si (prop_i[x]!=i)Alors
          Attendre((hor_i[prop_i[x]],prop_i[x])>(hor_i[i],i));
     Finsi
     renvoyer(val_i[x]);
Fin
```

A la modification d'un objet, on met aussi à jour les variables dont on n'est pas propriétaire au vu des dernières valeurs reçues. On enregistre l'heure de l'écriture, on informe les autres sites de la modification et on met à jour son horloge logique.

```
write(x,v)
Début
     val_i[x]= v;
     Pour tout y tel que prop_i[y]!=i faire
          val_i[y]= der_i[y];
     Fin pour
     hor_i[i]= h_i;
     diffuser(maj,h_i,x,v);
     h_i++;
Fin
```

A la réception d'une modification, on mémorise la valeur et on vérifie qu'elle peut être renvoyée en cas de lecture. Puis on met à jour le tableau des heures d'écritures.

```
sur_réception_de(j,(maj,h,x,v))
Début
     der_i[x]= v;
     Si ((h,j)<(hor_i[i],i)) Alors
          val_i[x]=v;
     Finsi
     hor_i[j]= h+1;
     h_i = max(h_i,h+1);
     envoyer_à(j,(acq,h_i);
Fin
```

L'acquittement sert uniquement à mettre à jour le tableau des heures des opérations.

```
sur_réception_de(j,(acq,h))
Début
     hor_i[j]= h;
Fin
```

### *8.3 Correction de l'algorithme*

Nous établissons la correction de l'algorithme en construisant la relation d'ordre total et en démontrant qu'elle vérifie les propriétés de la consistance séquentielle. Soit `op` une opération quelconque, on note `der(op)`, la dernière écriture exécutée sur le même site qui précède `op`.

A chaque écriture `w`, on associe l'heure logique de son exécution `h(w)` et l'identité du site qui l'exécute `i(w)`. On définit un ordre total entre écritures analogue à celui du paragraphe 6.2 par :

$$w_1 < w_2 \Leftrightarrow (h(w_1), i(w_1)) < (h(w_2), i(w_2))$$

On étend cet ordre aux lectures en plaçant toutes les lectures `r` telles que `der(r)=w`, immédiatement après `w` dans leur ordre d'occurence sur `i(w)`. Vérifions que les contraintes sont bien respectées par cet ordre.

Il étend la précédence locale par construction. Puisque l'horloge logique croît après chaque écriture et ne décroît jamais, les écritures sont correctement placées. Les lectures sont intercalées en respectant la précédence locale.

Examinons la valeur renvoyée par une lecture $r_i(x):v$. `v` est la valeur contenue dans $val_i[x]$ à la fin de l'appel. Soit la dernière écriture sur `x`, $w_j(x,v')$ qui précède (pour $<$) cette lecture et $w_l(x,v'')$ celle qui la suit. Appelons enfin $w_k = der(r_i)$.

S'il s'agit d'une lecture locale, la consistance est trivialement assurée puisque la précédence locale est préservée. Dans le cas d'une lecture distante, $w_j$ est différent de $w_k$. Puisque $w_j < w_k < w_l$ :

- la valeur fournie par $w_l$ ne peut être reçue avant $w_k$ et si elle est reçue avant la lecture, elle n'est pas encore acceptable au moment de la lecture,
- le résultat de la lecture ne peut être renvoyé avant la réception de la propagation de $w_j$ sur $i(w_k)$. Si la réception de cette propagation a lieu après $w_k$, elle est immédiatement acceptable sinon elle le devient après $w_k$.

Enfin comme nous l'avons expliqué précédemment, il ne peut y avoir d'attente pour une lecture au delà de la réception de tous les acquittements de l'écriture locale qui l'a précédée.

Le cas d'une lecture sur un objet non encore écrit se traite de manière similaire (et même plus simplement).

Remarque : Un site qui n'écrit plus ne "verra" jamais les valeurs des objets distants actualisées. Il est donc conseillé d'armer un "time-out" après chaque écriture et s'il arrive à expiration d'effectuer une écriture factice. Le réglage du délai correspond à la qualité temps-réel que l'on vise.

# 9 Exercices

***Sujet 1***

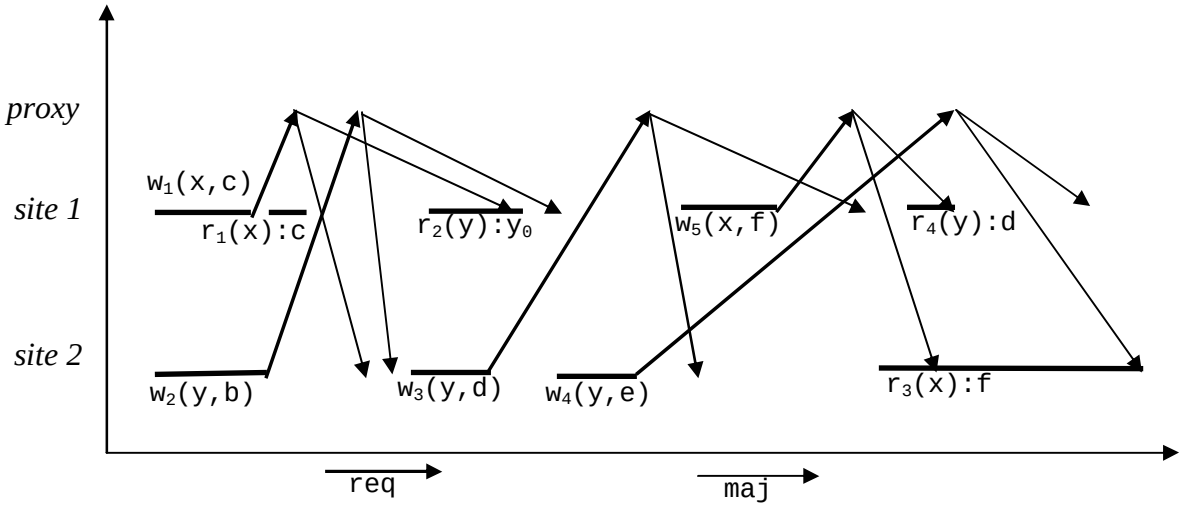
On se propose de définir un nouvel algorithme garantissant la consistance séquentielle n'utilisant pas les horloges logiques. Cet algorithme est une adaptation de l'algorithme de la propagation des écritures.

Pour ce faire, on dédie un site appelé `proxy` qui ne participe pas à l'application. Lorsqu'un site désire diffuser la mise à jour d'une de ses variables, il envoie une requête de mise à jour au `proxy` qui la diffuse à tous les sites y compris l'initiateur de la mise à jour. L'ordre des réceptions sur le `proxy` (et par suite sur chaque site) détermine l'ordre global des écritures.

Les écritures sont non bloquantes et incrémentent un compteur $\texttt{nbreq}_i$ initialisé à zéro qui conditionne le déroulement des lectures.

Une lecture distante attend que le compteur soit nul et renvoie la valeur "significative" de l'objet ($\texttt{val}_i$). Une lecture locale renvoie immédiatement cette valeur.

A la réception d'une valeur "distante", on met à jour la dernière valeur reçue ($\texttt{der}_i$). Dans le cas d'une valeur locale, le compteur est décrémenté et s'il devient nul toutes les dernières valeurs reçues ($\texttt{der}_i$) deviennent significatives ($\texttt{val}_i$).




La figure ci-dessus décrit une exécution possible de cet algorithme incluant le flux des messages et les valeurs renvoyées par les lectures.

Les variables d'un site `i` sont :

- $\texttt{val}_i\texttt{[Objets]}$ : tableau de valeurs indicé par les objets. Ces valeurs serviront comme résultat d'un appel à `read()`. Chaque cellule est initialisée à une valeur par défaut commune à tous les sites mais qui est spécifique à chaque objet.
- $\texttt{der}_i\texttt{[Objets]}$ : tableau de valeurs indicé par les objets. Ces valeurs sont les dernières valeurs connues de l'objet correspondant. Ce tableau est initialisé de la même façon que le tableau précédent. Il ne sert en fait qu'aux objets "distants".
- $\texttt{nbreq}_i$ : nombre de requêtes en cours de `i` initialisé à `0`.
- $\texttt{prop}_i\texttt{[Objets]}$ : constante qui contient l'identité du propriétaire de chaque objet.

**Question 1** Ecrire les primitives suivantes :
Pour un site de l'application

- **`read(x)`**
- **`write(x,v)`**
- **`sur_réception_de(proxy,(maj,x,v))`**

Pour le proxy

- **`sur_réception_de(j,(req,x,v))`**

**Question 2** Redessinez la figure en indiquant tous les changements de valeur des variables.

**Question 3** Quelles sont les modifications à faire sur l'algorithme si les objets n'ont plus de propriétaire, autrement dit si les écritures peuvent être effectuées par un site quelconque ? Indication : il y en a peu.

## 10 Références

[Att91] H. Attiya, J. Welch "Sequential consistency versus linearizability" Proceedings du troisième ACM symposium on parallel algorithms and architectures (juillet 1991) pp 304-315.

[Her90] M. Herlihy, J. Wing "Linearizability : a correctness conditions for concurrent objects" ACM transactions on programming languages and systems. Vol 12,3 (1990) pp 463-492.

[Li89] K. Li, P. Hudak "Memory coherence in shared virtual memory systems" ACM transactions on Computer Systems Vol. 7,4 (1989) pp 321-359.

[Miz92] M. Mizuno, M. Raynal, G. Singh, M. Neilsen "Communication efficient distributed shared memories" Rapport de recherche IRISA n° 691. Décembre 1992.

[Ray93] M. Raynal, M. Mizuno "How to find his way in the jungle of consistency criteria for distributed object memories" Rapport de recherche INRIA n° 1962. Juillet 1993.

# CHAPITRE VIII

# AUTO-STABILISATION

*version du 1er décembre 2003*

## 1 Problématique

Afin de cerner le concept d'algorithme auto-stabilisant, nous allons étudier le comportement d'un composant actif de réseaux, le commutateur.

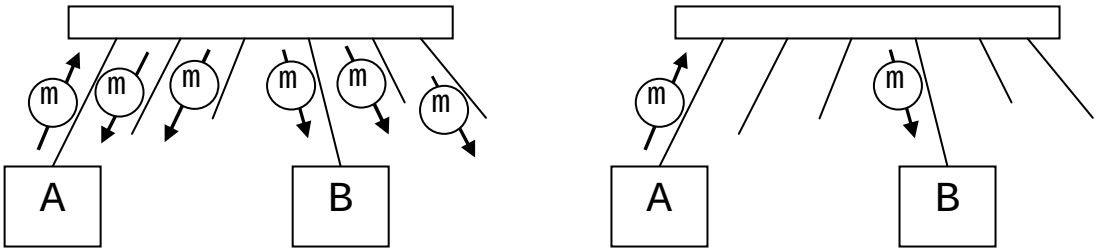



Figure 8.1 Fonctionnement d'un concentrateur et d'un commutateur

La figure 8.1 présente deux composants actifs de réseau : le concentrateur et le commutateur. Tous les deux sont dotés de ports (éventuellement) connectés soit à des machines soit à d'autres composants de réseaux. Lorsque le concentrateur (représenté à gauche) reçoit une trame `m` à destination de la machine `B` - la machine est ici désignée par son adresse MAC -, il retransmet la trame sur tous ses autres ports. Cette façon de procéder est simple mais inefficace car la bande passante des ports est consommée inutilement et de plus dans le cas du protocole Ethernet, le risque de collision est augmenté.

De son côté, le commutateur (représenté à droite) ne retransmet la trame que sur le port à partir duquel il "pense" pouvoir atteindre la machine `B`. Mais de quelle façon connait-il ce port ? La réponse réside dans le mécanisme d'apprentissage dont est pourvu le commutateur. Le commutateur est doté d'une table associative (notée `Table` dans la suite) dans laquelle chaque cellule comporte deux champs : une adresse de machine et un numéro de port. Cette table est initialement vide. Nous décrivons ci-dessous le principe d'acheminement d'une trame.

Lorsqu'une trame arrive sur un port, le commutateur met à jour la table en ajoutant l'association entre l'adresse de l'émetteur et le port de réception (et en supprimant éventuellement une association erronée). Puis il recherche une association pour le destinataire et envoie la trame sur le port trouvé en cas de succès. En cas d'échec, il diffuse la trame sur tous les ports (excepté le port de réception). Nous avons indiqué ci-dessous une version algorithmique de ce mécanisme.

```
Sur_réception_de(port,(@emet,@dest,m))
Début
     portbis = recherche(Table,@emet);
     Si portbis != port  Alors
          Si portbis != NULL Alors
               Supprime(Table,@emet);
          Finsi
          Ajoute(Table,(@emet,port));
     Finsi
     portbis = recherche(Table,@dest);
     Si portbis != NULL  Alors
          envoyer_à(portbis,(@emet,@dest,m));
     Sinon
          Pour tout portbis!=port faire
               envoyer_à(portbis,(@emet,@dest,m));
          Finpour
     Finsi
Fin
```

Ainsi très rapidement, le commutateur localise toutes les machines sur le réseau et les trames ne sont plus diffusées. Qu'arrive-t-il si la table ne reflète plus la topologie du réseau ? Ceci peut survenir à la suite d'une perturbation temporaire du commutateur ou encore plus simplement en raison d'une modification des connexions par un administrateur réseau.

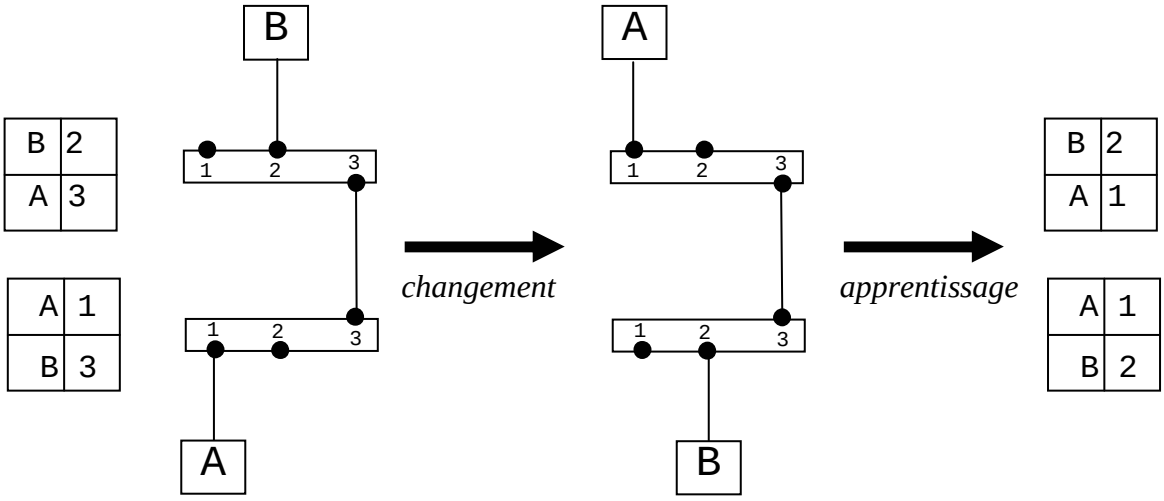



Figure 8.2 Une modification de la topologie et ses conséquences

Sur la figure 8.2, le réseau est constitué de deux commutateurs interconnectés. Les deux machines A et B ont été déplacées et reconnectées. Avec le mécanisme d'apprentissage, chaque commutateur corrige rapidement le port de sortie associé à la machine qu'on lui a nouvellement connectée. Cependant, le commutateur supérieur pense que B est connectée sur son port 2 et le commutateur inférieur pense que A est connectée sur son port 1. Aussi si B et A veulent communiquer, chacun des commutateurs envoie les trames vers une voie "sans issue". Le mécanisme d'apprentissage est alors impuissant car aucune trame ne circule sur les ports n° 3.

Nous sommes maintenant en mesure d'énoncer la propriété fondamentale d'un algorithme auto-stabilisant. Appelons "configuration", un état global de l'algorithme et de son environnement.

Sans changement ultérieur de l'environnement et partant d'une configuration quelconque, un algorithme auto-stabilisant atteint en un nombre fini d'actions une configuration cohérente, c'est à dire une configuration où les fonctionnalités de l'algorithme seront garanties dans le futur de son exécution.

Remarquons que certaines informations sont permanentes et ne doivent pas être incluses dans la configuration. Dans notre exemple, la numérotation des ports et les adresses des machines sont des informations permanentes. De même, nous supposons que le code de l'algorithme n'est jamais corrompu (par contre le compteur ordinal peut pointer n'importe où dans le code).

Il est possible de transformer le mécanisme de notre commutateur pour éviter la situation décrite plus haute. Pour cela attachons une durée de validité à chaque entrée de la table; cette durée est réinitialisée à chaque fois qu'on reçoit une trame de la machine sur le port associé. Lorsque l'entrée n'est plus valide, elle est supprimée de la table. Avec ce nouveau mécanisme, la gestion du commutateur est-elle auto-stabilisante ? Deux réponses sont possibles, selon ce qu'on considère être la fonctionnalité du commutateur :

- **Acheminer des messages** Alors le commutateur est auto-stabilisant, puisqu'une fois les adresses erronées éliminées, il ne peut y avoir que des adresses correctes dans sa table. Donc soit un message est correctement routé, soit il est diffusé. Dans les deux cas, les messages arrivent à destination.
- **Maintenir une table de routage** Alors le commutateur n'est pas auto-stabilisant car une machine qui ne transmet pas de message pendant un temps supérieur au délai de validité n'est plus référencée dans la table.

On distingue usuellement deux types de tâche pour les algorithmes auto-stabilisants :

- Les tâches statiques : dans une configuration correcte, l'algorithme ne modifie plus les données à maintenir. Le maintien d'une table routage est une tâche statique.
- Les tâches dynamiques : à partir d'une configuration correcte, un service est assuré de manière attendue. La gestion de l'exclusion mutuelle entre sites est une tâche dynamique.

Dans la deuxième section, nous présentons la construction d'un arbre de plus courts chemins qui peut être utilisé par un algorithme de routage. Puis nous montrons comment émuler une mémoire virtuelle répartie avec variables propriétaires. Enfin, en nous appuyant sur l'algorithme précédent, on montrera comment assurer l'exclusion mutuelle entre des sites organisés en anneau. Nous recommandons au lecteur intéressé l'excellent ouvrage de Shlomi Dolev [Dol00].

# 2 Routage auto-stabilisant [Dol89]

## *2.1 Principe et réalisation de l'algorithme*

Nous désirons construire une table de routage au dessus du graphe de communication. On remarque d'abord que l'on peut se restreindre à une destination particulière, la généralisation étant immédiate. Nous appellerons $i_0$ le site à atteindre. L'algorithme a pour objectif de construire un arbre de plus courts chemins vers la destination.

Pour ce faire, chaque site $i$ maintient deux variables significatives :

- un tableau $dist_i$ indicé par $i$ et ses voisins qui indique la distance supposée à $i_0$
- $père_i$ le voisin à emprunter pour y parvenir.

L'ensemble des voisins est représenté par $voisins_i$ (une information permanente). Le principe de l'algorithme est extrêmement simple. Chaque site exécute une boucle infinie et à chaque tour de boucle :

1. Il calcule en fonction des distances supposées de ses voisins, sa distance minimale à $i_0$ et simultanément son meilleur père.
2. Il envoie à ses voisins sa distance.
3. Il arme un time-out (information permanente) avant d'entamer le tour suivant.

A la réception d'une distance, le site met à jour la cellule appropriée de son tableau.

Variables du site $i$

- $voisins_i$ : constante contenant l'ensemble des voisins $i$ dans le graphe de communication.
- $time\_out_i$ : constante contenant la valeur d'un délai de suspension.
- $B_i$ : constante contenant un majorant strict du diamètre du réseau.
- $dist_i[voisins_i \cup \{i\}]$ : tableau des distances à $i_0$.
- $père_i$ : identité du prochain noeud sur la route.
- $temp_i$ : variable temporaire.

Algorithme du site `i`

Le service est ici un processus utilitaire qui tourne en tâche de fond.

```
router()
Début
     Tant que VRAI faire
          Si (i==i0) Alors
               disti[i]=0;
               pèrei=i0; // par convention
          Sinon
               disti[i]=Bi;
               Pour tout tempi ∈ voisinsi faire
                    Si disti[tempi]<disti[i]-1 Alors
                         perei=tempi;
                         disti[i]=disti[tempi]+1;
                    Finsi
               Finpour
          Finsi
          Pour tout tempi ∈ voisinsi faire
               envoyer_à(tempi,disti[i]);
          Finpour
          Armer(time_outi)
          Attendre();
     Fin tant que
Fin

sur_réception_de(j,d)
Début
     disti[j]=d;
Fin
```

La figure 8.3 décrit une exécution possible de cet algorithme (le site `1` est la destination). Nous faisons les hypothèses suivantes sur la configuration initiale :

- Le compteur ordinal du site `1` est en début de boucle.
- Le compteur ordinal des autres sites est positionné avant l'envoi des messages.
- Les valeurs initiales du père et de la distance supposée sont telles qu'indiquées sur le premier schéma.
- Le réseau ne contient pas de messages.

De plus, nous supposons que dans la suite de l'exécution, un message envoyé par un site arrive à destination avant l'expiration du time-out sur le site destinaire (ce qui revient à une exécution synchrone, voir le chapitre 3). Les voisins sont examinés par ordre croissant dans la boucle du choix du père.

On remarque les sites se stabilisent de proche en proche en partant de la destination. Nous allons formaliser cet argument pour établir la correction de l'algorithme au prochain paragraphe.

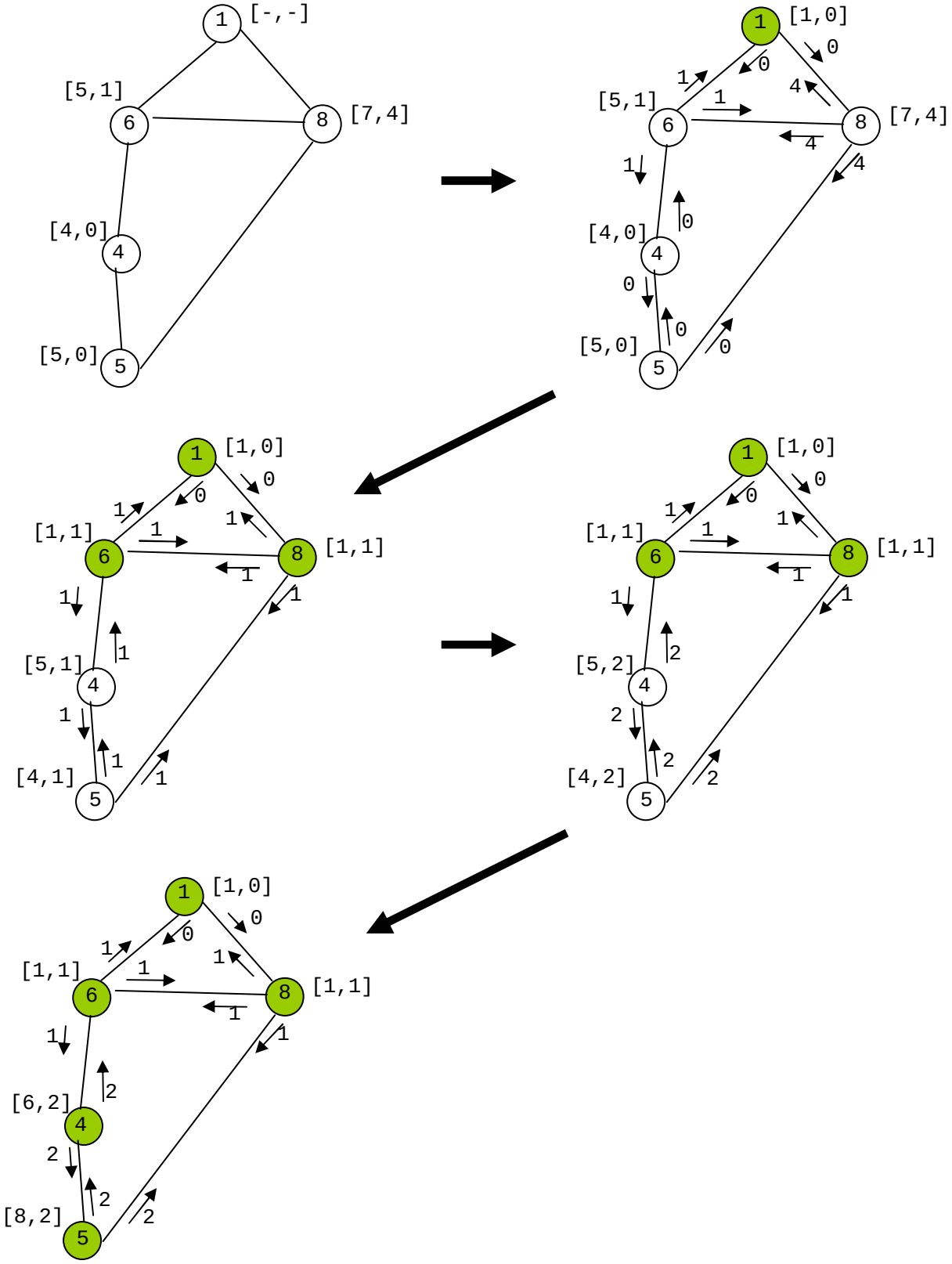



Figure 8.3 Une exécution du routage auto-stabilisant

### *2.2 Preuve de l'algorithme*

Nous introduisons d'abord quelques concepts associés à une exécution.

Une exécution est divisée en rondes. Chaque ronde débute à la fin de la ronde précédente et se termine lorsque :

- tous les processus ont exécuté au moins un tour complet de leur boucle ce qu'on appelle la première demi-ronde
- et les messages associés à ce tour de boucle ont été reçus.

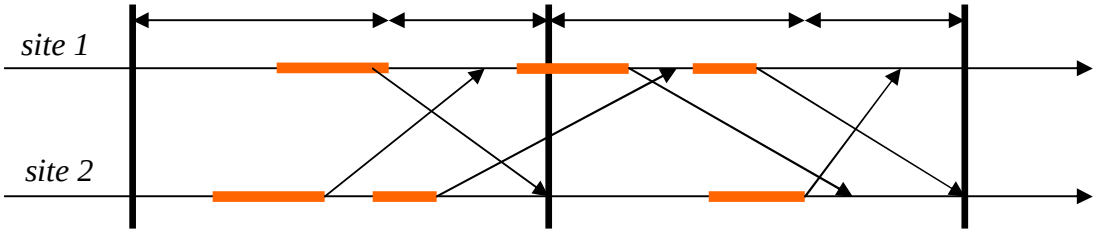



Figure 8.4 Deux rondes du routage auto-stabilisant

Sur la figure 8.4, nous avons représenté deux rondes avec leur découpage en demi-rondes. Les traits oranges correspondent à un tour de boucle du processus `router()`. Dans l'exécution présentée, les time-out sont comptés irrégulièrement (car la correction de l'algorithme ne repose pas sur une valeur particulière de temporisation). Notons d'abord qu'au cours d'une ronde, un site peut exécuter plusieurs tours de boucle et d'autre part que nous ne comptons pas dans une ronde, un tour de boucle débuté au cours de la ronde précédente.

Etant donnée une configuration, on appelle "une valeur fausse", une distance dans une variable ou dans un message qui ne correspond pas à la distance réelle. La `ppvf` correspond à la plus petite valeur fausse de la configuration. Par convention, s'il n'existe pas de valeur fausse on pose $ppvf = +\infty$.

Nous établissons maintenant la propriété clef de cet algorithme.

Propriété

1. Après `k+1` rondes de l'algorithme, on a $ppvf > k$
2. Après `k` rondes et une demi-ronde, l'ensemble des noeuds à une distance inférieure ou égale à `k` est organisée en un arbre de plus courts chemins (défini par $père_i$ et $dist_i[i]$).

Preuve (par récurrence)
Notons $ppvf^{(k)}$, la `ppvf` à la fin de la $k^{ème}$ ronde et $ppvf^{(0)}$ la `ppvf` initiale.
Evidemment $ppvf^{(0)} \geq 0$.
Montrons que $ppvf^{(k+1)} \geq ppvf^{(k)}+1$

En effet, au cours de la $k+1^{ème}$ ronde, la distance d'un noeud lorsqu'elle est recalculée est soit exacte, soit supérieure ou égale à $ppvf^{(k)}+1$. D'autre part à la fin de cette ronde ne circulent plus que des distances recalculées au cours de cette ronde puisque chaque canal a reçu une valeur recalculée (par définition d'une ronde). Ceci achève la preuve du premier point.

Après la première demi-ronde, la destination a ses variables, distance et père, correctement positionnées.

Supposons la propriété 2. établie à la k+1$^{ème}$ ronde et examinons la k+2$^{ème}$ ronde. Au cours de cette ronde, la `ppvf` est strictement supérieure à `k`. Donc les sites à distance inférieure ou égale à `k` conservent leurs variables, père et distance, inchangées. D'autre part, les messages qui définissent la deuxième demi-ronde de la ronde `k+1` ont été reçus par les sites à distance `k+1`. A la fin de la première demi-ronde de la ronde `k+2`, ces sites évalueront leur distance et choisiront leur père de manière correcte en raison du minorant du point 1. sur la `ppvf`.

Autrement dit, si `D` est le diamètre du graphe, au bout d'au plus `D+1/2` rondes, l'arbre est établi.

# 3 Gestion de mémoire virtuelle répartie [Dol97]

## *3.1 Principe de l'algorithme*

Dans l'algorithme précédent, les hypothétiques messages qui circuleraient sur le réseau n'empêche pas celui-ci de se stabiliser. Cependant la conception de la plupart des algorithmes auto-stabilisants se fait en supposant que les processus partagent une mémoire virtuelle répartie et plus précisément une gestion de variables propriétaires (voir le chapitre 6). Comme nous le verrons dans l'algorithme de la prochaine section, cela facilite grandement la conception des algorithmes.

Mais cette supposition n'a d'intérêt que si la gestion de la mémoire est elle-même auto-stabilisante et c'est donc l'objectif de la présente section. Avant toute chose, il faut savoir que dans un environnement réseau purement asynchrone, il n'existe pas de solution auto-stabilisante à ce problème [Gou91]. Heureusement moyennant des hypothèses faibles, ce résultat d'impossibilité n'est plus vérifié. Dans notre cas, nous supposons que chaque canal bidirectionnel contient un nombre de messages strictement inférieur à une borne B.

La solution s'obtient par modifications successives du protocole de lecture à la demande. Dans le protocole initial, le lecteur envoie une requête au propriétaire et attend la valeur en retour. En raison du caractère quelconque de la configuration initiale, il se peut que le lecteur attende une réponse que le propriétaire n'enverra jamais. Aussi la première modification consiste à faire circuler un jeton qui "porte" à la fois la requête et la réponse. Le lecteur attend un premier passage du jeton qui correspond à l'émission de sa requête et un deuxième passage pour recevoir la réponse.

Ceci ne résout pas encore le problème puisque le jeton peut être absent de la configuration initiale. Aussi à l'aide d'un time-out, si le propriétaire ne voit passer pas de jeton, il le réémet avec bien sûr dans ce cas la possibilité de plusieurs "jetons" circulant simultanément. Ce remède introduit un nouveau problème. Comment le lecteur va-t-il reconnaitre que le message qui arrive est le jeton ?

Pour cela, les messages sont numérotés par un compteur qui s'incrémente modulo B (le choix de ce modulo s'expliquera lors de la preuve). Chacun des sites a son propre compteur. Lorsque le lecteur voit arriver un message avec une nouvelle valeur, il reconnait le jeton. De même lorsque le propriétaire voit arriver un message avec sa propre valeur, il reconnait le jeton et incrémente son compteur avant de renvoyer la valeur. Le propriétaire supprime les messages différents du jeton.

Nous décrivons maintenant l'algorithme, puis nous illustrons son fonctionnement nominal avant d'établir la preuve de la stabilisation. L'algorithme se généralise facilement au cas de plusieurs lecteurs et de plusieurs variables. De plus tel qu'il est écrit, il supporte des canaux à pertes du moment qu'un message réémis indéfiniment finit par être reçu.

### *3.2 Description et exemple d'exécution*

Variables du site distant `i`

- $état_i$ : état du service. Cette variable prend une valeur parmi (`repos`, `prem_att`, `sec_att`).
- $h_i$ : compteur du lecteur
- $val_i$ : valeur à renvoyer à la lecture

Algorithme du site distant `i`
La lecture consiste à se mettre en (première) attente et attendre de repasser au repos pour renvoyer la valeur.

```
read(x)
Début
     état_i=prem_att;
     Attendre(état_i==repos);
     renvoyer(val_i);
Fin
```

A la réception d'une message, on teste d'abord s'il s'agit du jeton. Dans ce cas, on met à jour son compteur, puis si une lecture est en première attente, elle passe en seconde attente et si elle est en seconde attente, elle passe au repos en récupérant la valeur courante de l'objet. Dans tous les cas le message est renvoyé.

```
sur_réception_de(j,(v,h))
Début
     Si (h_i!=h) Alors
          h_i=h;
          Si (état_i == prem_att) Alors
               état_i=sec_att;
          Sinon si (état_i == sec_att) Alors
               val_i=v;
               état_i=repos;
          Finsi
     Finsi
     envoyer_à(j,(acq,h));
Fin
```

Variables du site propriétaire `j`

- $time\_out_j$ : constante contenant la valeur d'un délai de suspension.
- $h_j$ : compteur du propriétaire
- $x_j$ : valeur de l'objet `x`

```
Algorithme du site propriétaire j
```

L'écriture est purement locale.

```
write(x,v)
Début
     xj=v;
Fin
```

Ce processus utilitaire a pour but d'envoyer la valeur au site distant si un time-out expire.

```
régénérer()
Début
     Tant que VRAI faire
          envoyer_à(i,(xj,hj));
          Armer(time_outj);
          Attendre();
     Fin tant que
Fin
```

A la réception d'un message qui porte le compteur (i.e. reconnu comme un jeton), on incrémente celui-ci et on le renvoie. Puis on envoie la valeur et le compteur et on réarme le time-out pour éviter des envois superflus.

```
sur_réception_de(i,(acq,h))
Début
     Si (hj==h) Alors
               hj = hj+1 %B;
               envoyer_à(i,(xj,hj));
               Armer(time_outj);
     Finsi
Fin
```

Nous appelons dans la suite, une configuration nominale initiale, une configuration telle que les deux compteurs sont égaux, qu'il y a au moins un message dans le canal bidirectionnel et que tous les messages ont une valeur de compteur identique à celle des sites. Le message le plus "proche" du site propriétaire est appelé le jeton. La configuration (a) de la figure 8.5 décrit une telle situation. Avant de continuer, le lecteur est invité à vérifier que sur toutes les configurations de la figure 8.5, une réémission de message (suite à une expiration du time-out) ne change pas la situation décrite.

(b) Le jeton est arrivé sur le propriétaire qui incrémente son compteur ($\oplus$ désigne l'addition modulo B) et renvoie le jeton. Les messages présents sur le réseau s'ils arrivent sur le lecteur ne sont pas reconnus comme étant le jeton.

(c) Le jeton se dirige vers le lecteur suivi d'éventuelles copies (dues à l'expiration du time-out). Les autres messages sont absorbés par le propriétaire.

(d) Le jeton est le message le plus proche du lecteur.

(e) Le jeton est arrivé sur le lecteur qui met à jour son compteur.

(f) Les copies récentes du jeton ne sont pas reconnues par le lecteur; les vieilles copies sont absorbées et nous retrouvons la situation (a) avec les compteurs "incrémentés" de `1`.

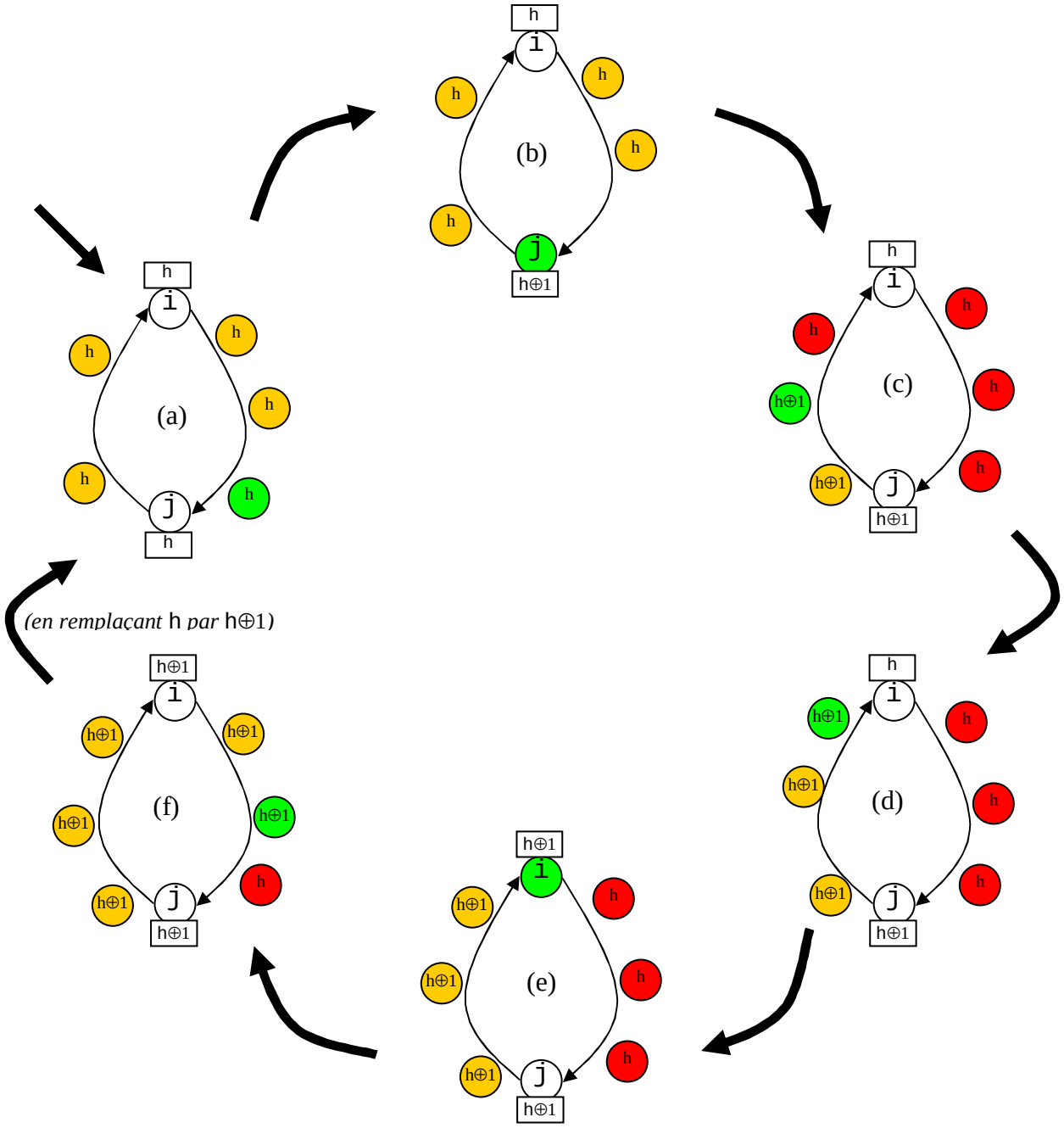



Figure 8.5 Circulation du jeton (une fois la phase de stabilisation passée)

### *3.3 Correction de l'algorithme*

Comme l'avons vu sur l'exemple précédent, une fois atteinte une configuration nominale initiale, l'algorithme est semblable au protocole de lecture à demande. Il ne nous reste qu'à démontrer que partant d'une configuration quelconque, on atteint une configuration nominale initiale.

Nous procédons par étapes.

1. Le compteur du propriétaire ne reste jamais bloqué. Supposons qu'il n'en soit pas ainsi. Un message estampillé avec cette valeur sera émis à l'expiration du time-out puisque le propriétaire ne reconnait jamais le jeton. Ce message passe le lecteur et revient au propriétaire qui le reconnait comme jeton et incrémente son compteur. D'où la contradiction.
2. Il y a moins de B messages dans la configuration initiale. Donc une valeur de compteur est absente. D'après le premier point, on atteint une configuration où le compteur du propriétaire a cette valeur (ce peut être la configuration initiale). Dans cette configuration, un message est envoyé avec la valeur de ce compteur et il est le seul à avoir cette propriété.
3. Tous les messages qui le précèdent sont supprimés par le propriétaire et ceux qui le suivent portent la même valeur de compteur. Lorsqu'il arrive sur le lecteur, celui-ci prend cette valeur. Enfin lorsqu'il est le message le plus proche du propriétaire, nous retrouvons la configuration (a) de la figure 8.5.

# 4 Exclusion mutuelle [Dij73]

## *4.1 Présentation*

Le lecteur attentif aura remarqué que le protocole précédent assure implicitement une exclusion mutuelle entre le propriétaire et le lecteur. On désire généraliser ce principe pour obtenir l'exclusion mutuelle entre un ensemble de sites. C'est l'objectif de l'algorithme de Dijkstra qui a de plus la particularité d'être l'agorithme d'où a émergé le concept d'auto-stabilisation.

Tout d'abord nous organisons les `N` sites en un anneau "unidirectionnel" avec un site distingué $i_0$ (l'équivalent du propriétaire). Chaque site est pourvu d'un compteur. Nous faisons l'hypothèse simplificatrice que chaque site peut lire la valeur du compteur du site précédent sur l'anneau. Dans les deux derniers paragraphes, nous lèverons cette hypothèse.

La possibilité d'entrer en section critique est matérialisée de la façon suivante :

- Le site distingué peut entrer en section critique, lorsque son compteur est égal au compteur précédent.
- Un autre site peut entrer en section critique lorsque son compteur est différent du compteur précédent.

La sortie de section critique se définit ainsi :

- Le site distingué incrémente son compteur modulo `N+1` (la valeur du modulo se comprendra lors de la preuve).
- Un autre site recopie la valeur du compteur précédent dans son compteur.

Evidemment, si l'application n'est pas intéressée par la section critique, le service effectue la sortie de section critique immédiatement. Avant de spécifier l'algorithme, nous donnons un exemple de fonctionnement nominal et une phase de stabilisation.

De manière analogue à l'algorithme précédent, une configuration nominale initiale est une configuration dans laquelle tous les compteurs sont égaux.

Ainsi les schémas (a) et (f) de la figure 8.6 décrivent des configurations nominales initiales. Les schémas intermédiaires montrent que chaque site sur l'anneau peut successivement entrer en section critique. De plus, aucun autre site ne peut entrer simultanément en section critique. Les propriétés de sûreté et de vicacité de l'exclusion mutuelle sont ainsi vérifiées si on démarre d'une configuration initiale nominale. Sur cette figure et la suivante, les sites qui ont la possibilité d'entrer en section critique sont colorés en vert et le site distingué est cerclé d'un trait gras.

Il nous restera à montrer que de toute configuration on atteint une configuration initiale nominale. Pour comprendre ce qui garantit cette "convergence", nous déroulons un exemple de la phase de stabilisation sur la figure 8.7.

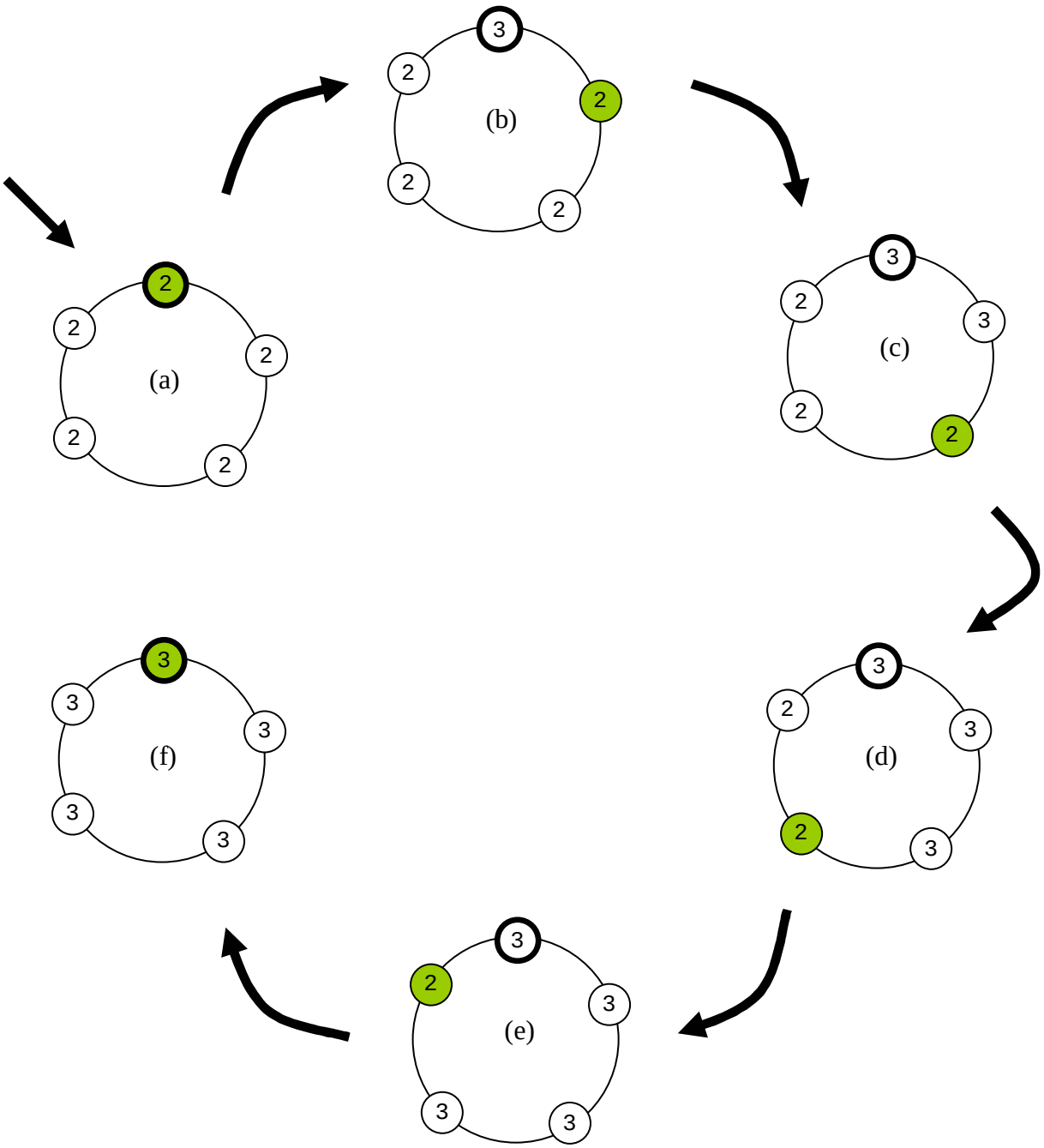



Figure 8.6 Fonctionnement nominal

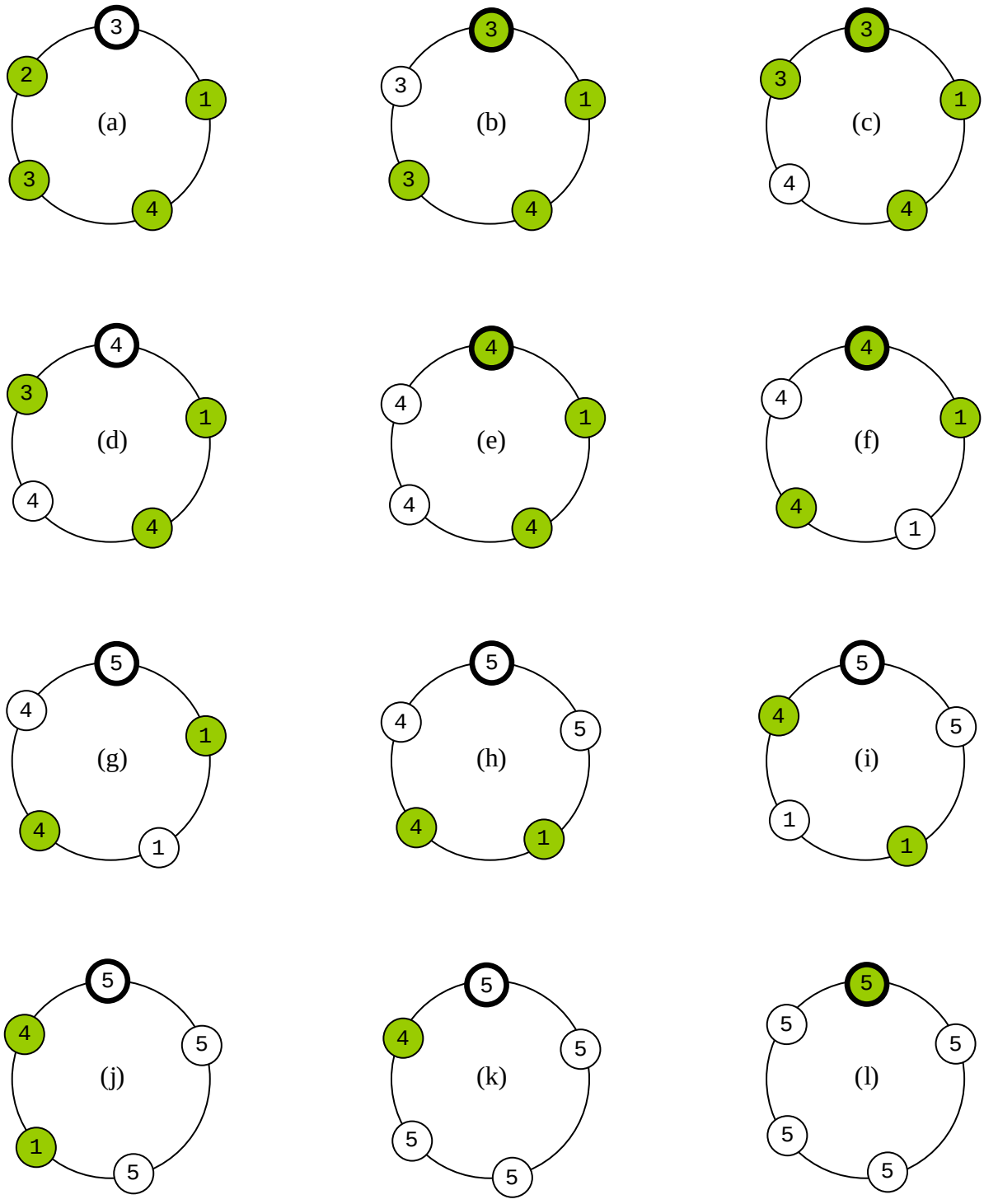



Figure 8.7 Phase de stabilisation

Au cours de la stabilisation, le site distingué incrémente son compteur en (d) et en (g). La valeur prise en (g) est absente sur les autres sites. Aussi la prochaine fois, que le site distingué incrémente son compteur (après la configuration (l)), les autres sites se seront transmis cette valeur et on aura atteint une configuration nominale initiale. La preuve formelle suit la spécification de l'algorithme.

### *4.2 Spécification de l'algorithme*

Lorsque l'application d'un site veut exécuter une fonction en section critique, elle appelle la primitive `SC(f)` où `f` est la fonction à exécuter. Le service est doté d'un processus `circuler()` qui tourne en arrière plan.

Variables du site `i`

- $c_i$ : compteur du site i.
- $cprec_i$ : compteur du site précédent i sur l'anneau. Il n'est accédé qu'en lecture par le site `i`.
- $f_i$ : fonction à exécuter en section critique
- $time\_out_i$ : constante contenant la valeur d'un délai de suspension.

Algorithme du site `i`

Le processus qui fait circuler le droit d'entrer en section critique, teste d'abord la possibilité d'entrée en section critique. Si tel est le cas, il appelle la fonction prévue et remet la fonction à `NULL` après exécution. Il fait ensuite circuler le droit. Nous faisons ici l'hypothèse que le mécanisme d'appel de fonction applicative ne peut aborter le processus de service et que ce mécanisme teste si l'adresse est nulle.

```
circuler()
Début
     Tant que VRAI faire
          Armer(time_out_i);
          Attendre();
          Si   (i==i_0)&&(c_i==cprec_i)||
               (i!=i_0)&&(c_i!=cprec_i) Alors
               f_i();
               f_i=NULL;
               Si (i==i_0) Alors
                    c_i = c_i+1 %(N+1);
               Sinon
                    c_i = cprec_i;
               Finsi
          Finsi
     Fin tant que
Fin
```

Pour exécuter une fonction `f` en section critique un processus positionne la variable de fonction de manière appropriée et attend l'indication d'exécution.

```
SC(f)
Début
     f_i=f;
     Attendre(f_i==NULL);
Fin
```

### *4.3 Correction de l'algorithme*

Nous remarquons d'abord que la variable $f_i$ sera nécessairement soit positionnée à `NULL` soit à la fonction à exécuter après l'obtention du premier droit. Seul le premier appel à $f_i()$ peut donc être incorrect. Il suffit alors de démontrer que l'on atteint nécessairement une configuration initiale nominale et se placer après l'obtention du premier droit sur chaque site.

Nous procédons en plusieurs étapes.

1. Le compteur du site distingué ne reste jamais bloqué. Supposons qu'il n'en soit pas ainsi. Alors le site suivant ne peut plus faire qu'une mise à jour de sa variable. Le suivant de ce site ne peut faire que deux mises à jour, etc. On obtient donc une configuration où aucun site ne peut faire de mise à jour. En raisonnant successivement sur les sites non distingués, on conclut que toutes les valeurs sont identiques. Mais dans ce cas, le site distingué peut faire une mise à jour, d'où la contradiction.
2. Il y a moins de `N+1` valeurs différentes dans la configuration initiale. Donc une valeur de compteur `val` est absente. D'après le premier point, on atteint une configuration $conf_1$ où le compteur du site distingué a cette valeur (ce peut être la configuration initiale).
3. Examinons maintenant la configuration $conf_2$ qui suit $conf_1$ dans laquelle le site distingué peut à nouveau modifier son compteur. Puisque `val` est absente des compteurs des sites non distingués, un raisonnement par récurrence (en "remontant" l'anneau à partir du site distingué) montre que chacun d'entre eux doit positionner au moins une fois son compteur à `val`. Un autre raisonnement par récurrence (cette fois-ci dans le sens de l'anneau) montre que ces sites ne peuvent ensuite le modifier tant que le site distingué n'a pas incrémenté son compteur. $conf_2$ est donc une configuration nominale initiale.

Une preuve plus élaborée montre que le compteur peut être incrémenté modulo `N-1`.

### *4.4 Extension de l'algorithme*

Nous allons maintenant lever l'abstraction qui consiste à supposer qu'un site peut lire le compteur de son prédécesseur. $cprec_i$ sera maintenant une copie distante de ce compteur maintenu par l'algorithme de mémoire virtuelle répartie. Cependant, nous devons prendre quelques précautions quant à la valeur du modulo. Pour l'instant nous le posons égal à `H` et nous discuterons ensuite de sa valeur précise. Nous donnons en totalité l'algorithme composé pour illustrer les précautions à prendre lors de la composition.

Variables du site `i`

Tout d'abord chaque site joue à la fois le rôle du propriétaire pour son compteur et celui du lecteur pour le compteur précédent. Ceci explique que la variable $h_i$ soit "dédoublée".

- $c_i$ : compteur du site i.
- $cprec_i$ : copie distante du compteur du site précédent i sur l'anneau.
- $f_i$ : fonction à exécuter en section critique
- $h_i$ : compteur associé à $c_i$.
- $état_i$ : état du service de lecture. Cette variable prend une valeur parmi (`repos`, `prem_att`, `sec_att`).
- $hprec_i$ : compteur associé à $cprec_i$
- $val_i$ : valeur à renvoyer à la lecture
- $time\_out_i$ : constante contenant la valeur d'un délai de suspension.
- $suivant_i$ : constante contenant l'identité du suivant sur l'anneau.

Algorithme du site `i`

La première partie de l'algorithme est presque identique à l'algorithme précédent à la différence près que $cprec_i$ donne lieu à une lecture distante.

```
circuler()
Début
     Tant que VRAI faire
          // le read introduit une temporisation implicite
          cprec_i=read();
          Si   (i==i_0)&&(c_i==cprec_i)||
               (i!=i_0)&&(c_i!=cprec_i) Alors
               f_i();
               f_i=NULL;
               Si (i==i_0) Alors
                    c_i = c_i+1 % H;
               Sinon
                    c_i = cprec_i;
               Finsi
          Finsi
     Fin tant que
Fin

SC(f)
Début
     f_i=f;
     Attendre(f_i==NULL);
Fin
```

La deuxième partie consiste à faire jouer les deux rôles de la lecture distante à un même site selon la variable.

La fonction `read()` est inchangée.

```
read()
Début
     état_i=prem_att;
     Attendre(état_i==repos);
     renvoyer(val_i);
Fin
```

Ce processus utilitaire doit être ordonnancé de manière (faiblement) équitable avec le processus `circuler()`. Autrement dit, aucun des deux ne doit être privé indéfiniment d'exécution.

```
régénérer()
Début
     Tant que VRAI faire
          envoyer_à(suivant_i,(c_i,h_i));
          Armer(time_out_i);
          Attendre();
     Fin tant que
Fin
```

Cette primitive correspond au lecteur distant. Aussi elle manipule $hprec_i$.

```
sur_réception_de(j,(v,h))
Début
     Si (hprec_i!=h) Alors
          hprec_i=h;
          Si (état_i == prem_att) Alors
               état_i=sec_att;
          Sinon si (état_i == sec_att) Alors
               val_i=v;
               état_i=repos;
          Finsi
     Finsi
     envoyer_à(j,(acq,h));
Fin
```

Cette primitive correspond au propriétaire. Aussi elle manipule $h_i$.

```
sur_réception_de(j,(acq,h))
Début
     Si (h_i==h) Alors
               h_i = h_i+1 %B;
               envoyer_à(suivant_i,(c_i,h_i));
               Armer(time_out_i);
     Finsi
Fin
```

### *4.5 Correction de l'algorithme étendu*

Nous remarquons d'abord que les différents protocoles de lecture distante (un par site) sont indifférents à l'exécution de la partie liée à la section critique. Nous avons déjà établi que ce protocole est auto-stabilisant. Nous nous plaçons donc en un instant qui suit la stabilisation de ces algorithmes.

Dans ce cas, nous pouvons identifier l'instruction "$\texttt{cprec}_i\texttt{=read();}$" à l'instruction "$\texttt{cprec}_i\texttt{=c}_j$" d'une exécution séquentielle de l'algorithme pour `j` prédecesseur de `i` sur l'anneau.

Posons `H=2.N+1` et démontrons la stabilisation.

Nous procédons en plusieurs étapes.

1. Le compteur du site distingué ne reste jamais bloqué. Supposons qu'il n'en soit pas ainsi. Alors le site suivant ne peut plus faire que deux mises à jour de sa variable : une avant la copie du compteur précédent, une après. De même, le suivant de ce site ne peut faire que quatre mises à jour, etc. On obtient donc une configuration où aucun site ne peut faire de mise à jour et de là une autre configuration où toutes les copies distantes sont égales aux variables. En raisonnant successivement sur les sites non distingués, on conclut que toutes les valeurs sont identiques. Mais dans ce cas, le site distingué peut faire une mise à jour, d'où la contradiction.
2. Il y a moins de `2.N+1` valeurs différentes dans la configuration initiale. Donc une valeur de compteur `val` est absente. D'après le premier point, on atteint une configuration $\texttt{conf}_1$ où le compteur du site distingué a cette valeur (ce peut être la configuration initiale).
3. Examinons maintenant la configuration $\texttt{conf}_2$ qui suit $\texttt{conf}_1$ dans laquelle le site distingué peut à nouveau modifier son compteur. Puisque `val` est absente des compteurs des sites non distingués et de toutes les copies, un raisonnement par récurrence (en "remontant" l'anneau à partir du site distingué) montre que chacun d'entre eux doit positionner au moins une fois son compteur à `val`. Un autre raisonnement par récurrence (cette fois-ci dans le sens de l'anneau) montre que ces sites ne peuvent ensuite le modifier tant que le site distingué n'a pas incrémenté son compteur. $\texttt{conf}_2$ est donc une configuration nominale initiale.

Une preuve plus élaborée montre que le compteur peut être incrémenté modulo `2.N-1`.

Remarque Ce type de composition est pratiqué de manière très générale pour obtenir des algorithmes auto-stabilisants traitant des tâches complexes.

# 5 Exercices

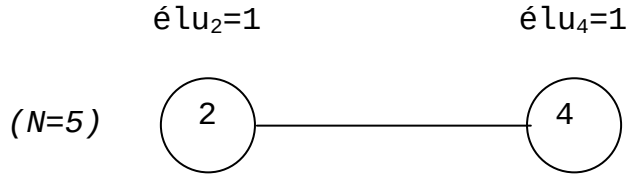
***Sujet 1***

On se propose de définir un algorithme d'élection auto-stabilisante sur un graphe de communication quelconque. Le site élu sera le site de plus petite identité. Nous précisons d'abord les hypothèses sur l'environnement :

- Chaque site `i` a en mémoire permanente sa propre identité et l'ensemble des identités de ses voisins stocké dans la constante $\texttt{Voisins}_i$.
- Chaque site peut lire les variables de ses voisins.
- Les identités des sites du réseau forment un sous-ensemble de l'intervalle `0..N`.
- Chaque site ne connaît que l'identité de ses voisins.
- Le site maintient une variable $\texttt{élu}_i$ à valeurs dans `0..N` qui contiendra l'identité du site élu.

Une première tentative consiste pour le site `i` à mettre à jour sa variable $\texttt{élu}_i$ périodiquement avec le minimum de `i` et des variables $\texttt{élu}_j$ pour tout $j \in \texttt{Voisins}_i$.

**Question 2** Expliquer en vous appuyant sur la figure ci-dessous qui représente un état initial possible pourquoi cet algorithme ne fonctionne pas.



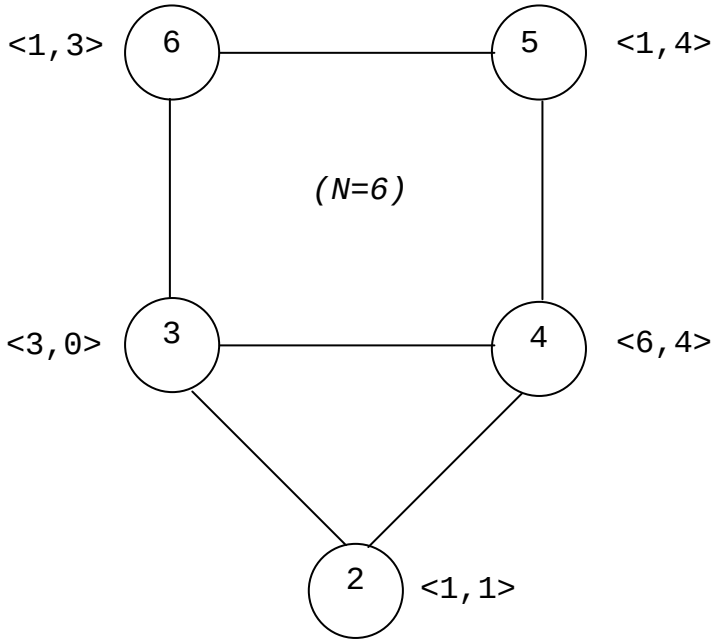

Afin de remédier au problème découvert, on décide d'ajouter une nouvelle variable $\texttt{distance}_i$ qui représente la distance minimale supposée à laquelle se trouve l'élu. $\texttt{distance}_i$ prend ses valeurs dans `0..N`. On modifie l'algorithme précédent avec les règles suivantes :

- La variable $\texttt{élu}_i$ est mise à jour périodiquement avec le minimum de `i` et des variables $\texttt{élu}_j$ pour tout $j \in \texttt{Voisins}_i$ tel que $\texttt{distance}_j < N$.
- Puis si $\texttt{élu}_i = i$ alors $\texttt{distance}_i$ est mise à jour avec `0` sinon $\texttt{distance}_i$ est mise à jour avec le minimum des $\texttt{distance}_j + 1$ tel que $\texttt{élu}_j = \texttt{élu}_i$.

**Question 3** Ecrire l'algorithme du site `i`.

**Question 4** Dérouler l'algorithme à partir de l'état initial décrit à la figure ci-dessous en exécutant l'algorithme par ronde. Au cours d'une ronde, les sites `2`, `3`, `4`, `5` et `6` exécutent à tour de rôle leur mise à jour. On dessinera une figure par état obtenu à l'issue d'une ronde jusqu'à stabilisation de l'algorithme.

Dans le couple <x,y>
x désigne $élu_i$
et y désigne $distance_i$




**Question 5** Etablir la correction de l'algorithme en décomposant toute exécution en rondes telles qu'au cours d'une ronde, tout site a effectué au moins une mise à jour de ses variables. Donner une borne, fonction de N, sur le nombre maximal de rondes qu'il faut pour que toutes les variables $élu_i$ contiennent la plus petite identité du réseau.

## 6 Références

[Dij73] E.W. Dijkstra "Self-stabilization in spite of distributed control" EWD391 Springer-Verlag (1973) pp 41-46.

[Dol89] S. Dolev, A. Israeli, S. Moran "Self-stabilization of dynamic systems" Proceedings of the MCC Workshop on Self-stabilizing systems, MCC Technical Report N° STP-379-89 (1989)

[Dol97] S. Dolev, A. Israeli, S. Moran "Resource bounds for self-stabilizing message driven protocols" SIAM Journal of Computing 26, pp 273-290 (1997)

[Dol00] S. Dolev "Self-Stabilization" MIT press (2000)

[Gou91] M.G. Gouda, N. Multari "Stabilizing communication protocols" IEEE Transactions on computers, 40, pp 448-458 (1991)