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.

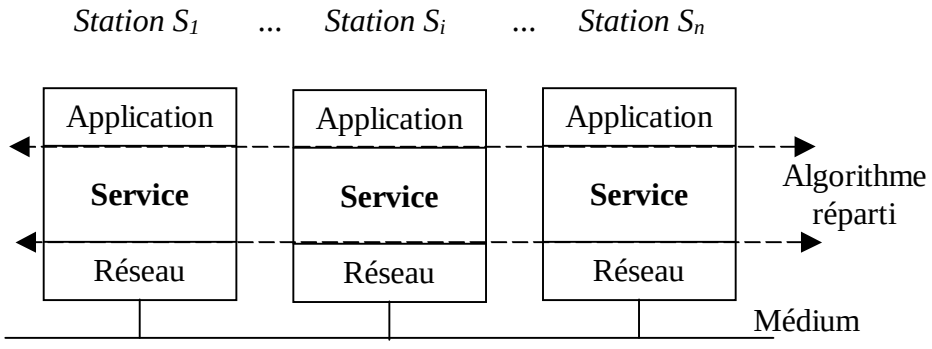


Figure 1.1 : L'environnement réparti

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.

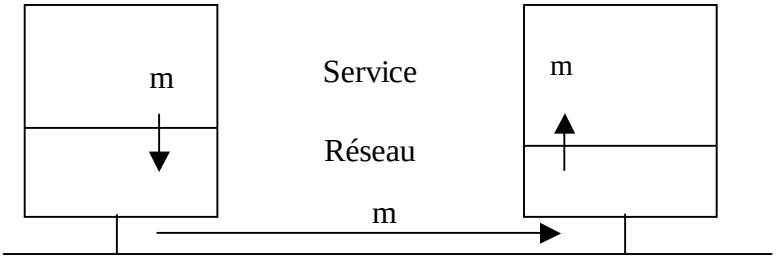


Figure 1.2 : Le trajet d'un message à travers les couches

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.

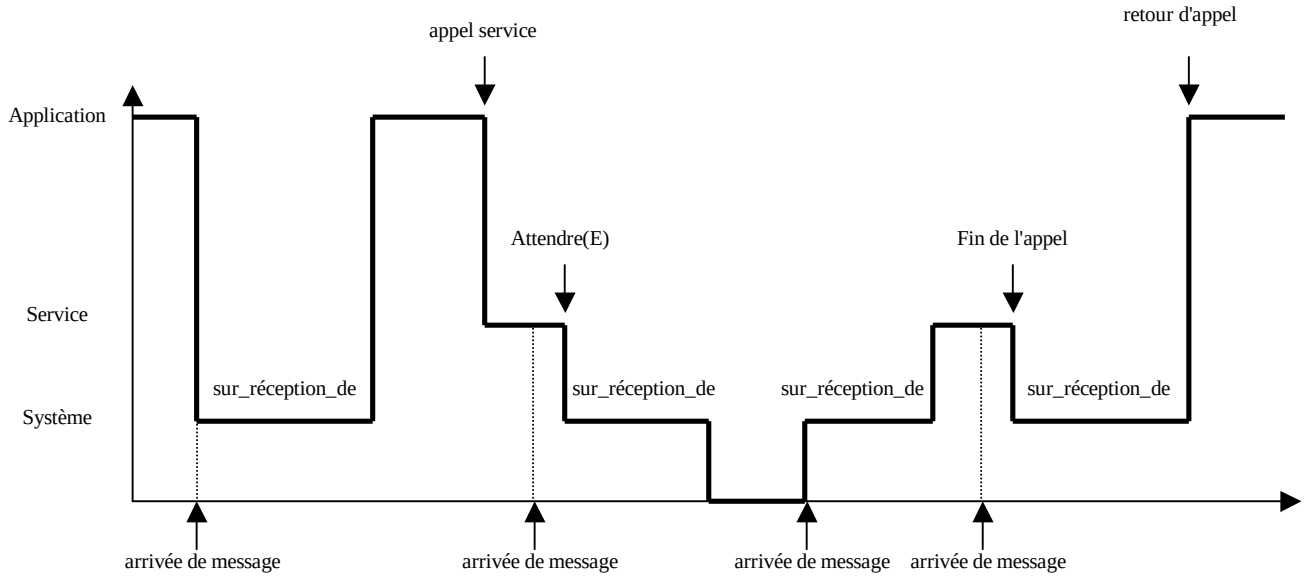


Figure 1.3 : Un déroulement d'une application sur un site

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.

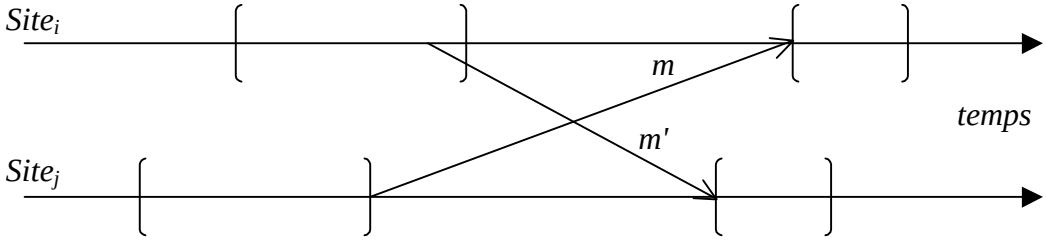


Figure 1.3 : Un exemple de chronogramme

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

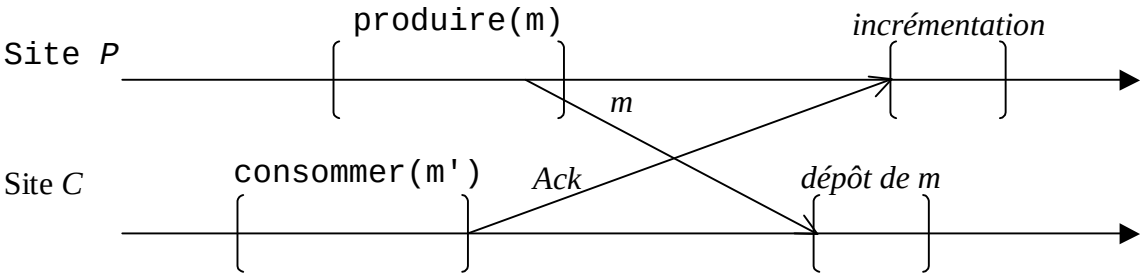


Figure 2.1 : Un extrait d'une exécution du producteur / consommateur

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).

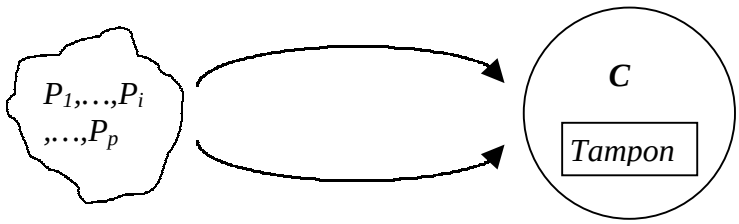


Figure 2.2 : `p` producteurs et un consommateur

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).

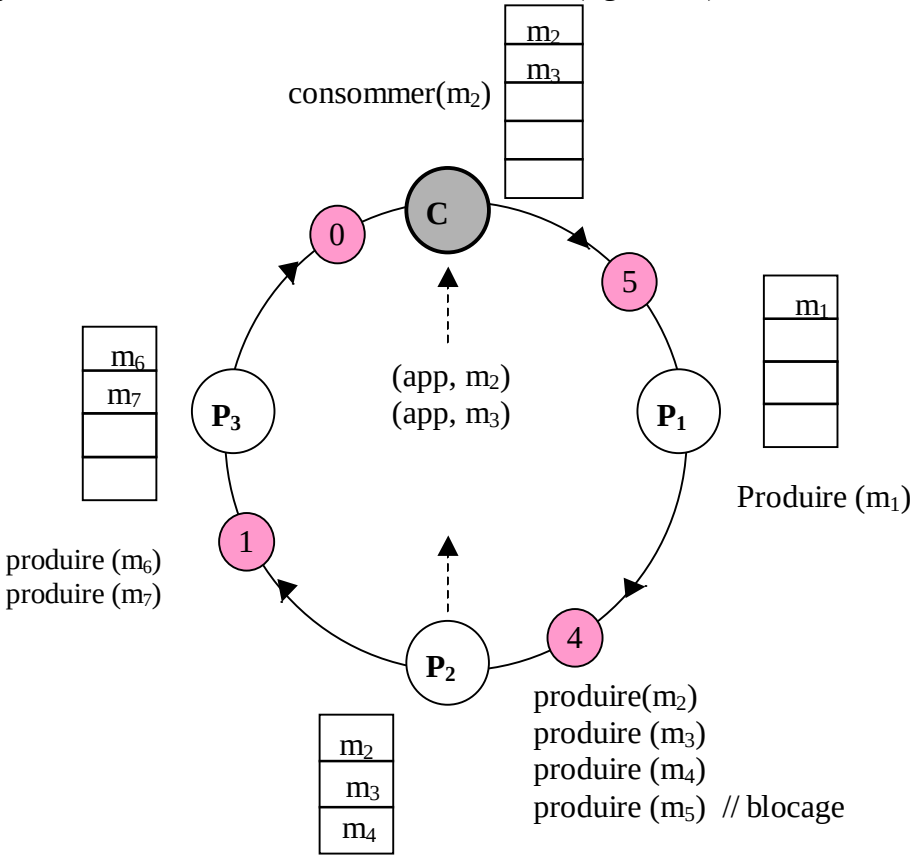


Figure 2.3 : Le premier tour du jeton sur l'anneau

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).

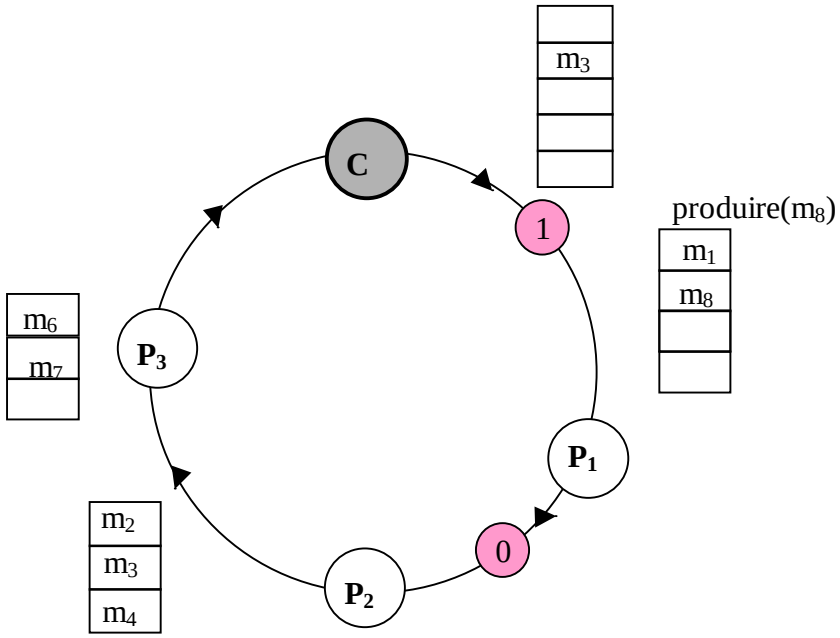


Figure 2.4 : Le deuxième tour du jeton sur l'anneau

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.

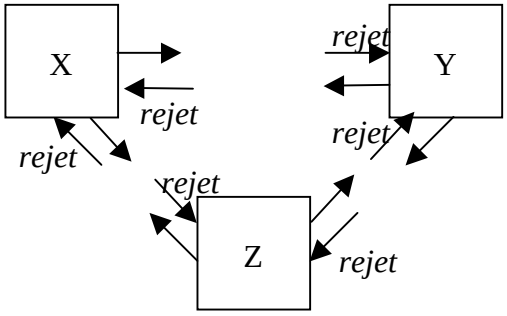


Figure 2.5 : Une situation d'interblocage

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.

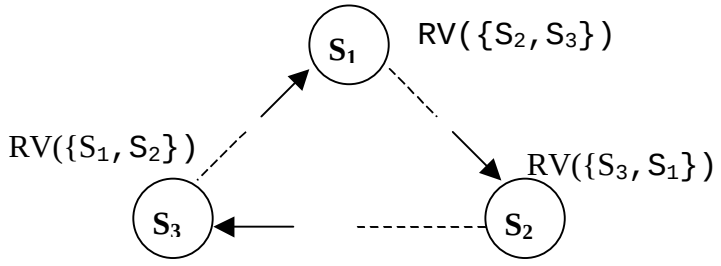


Figure 2.6 : Des demandes de rendez-vous symétriques

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.

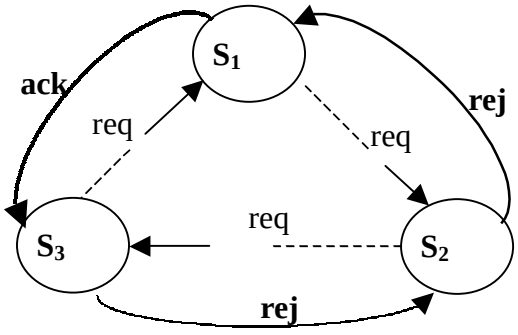


Figure 2.7 : Asymétrie de comportement

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`.

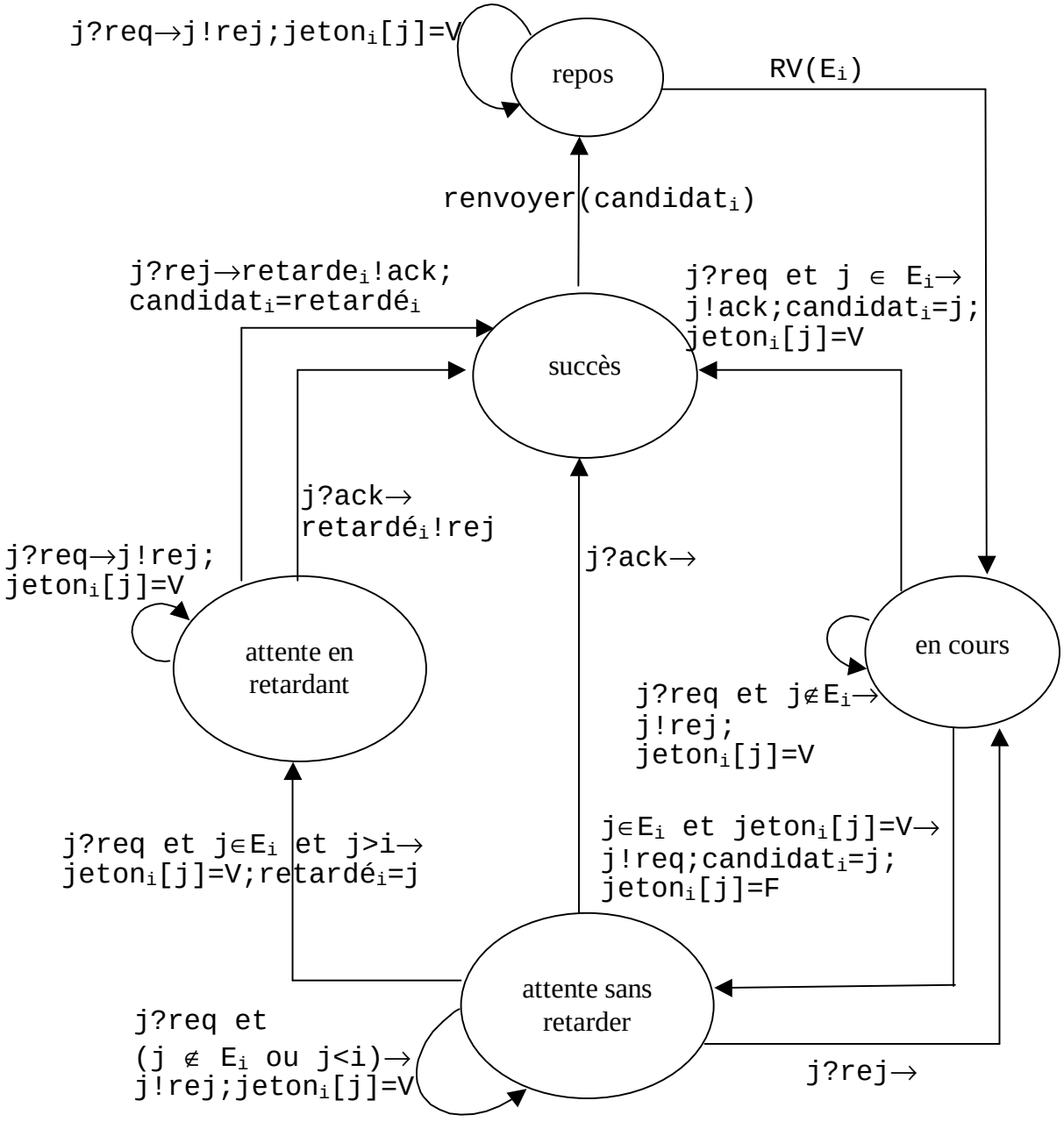


Figure 2.8 : Graphe d'états d'un site

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.

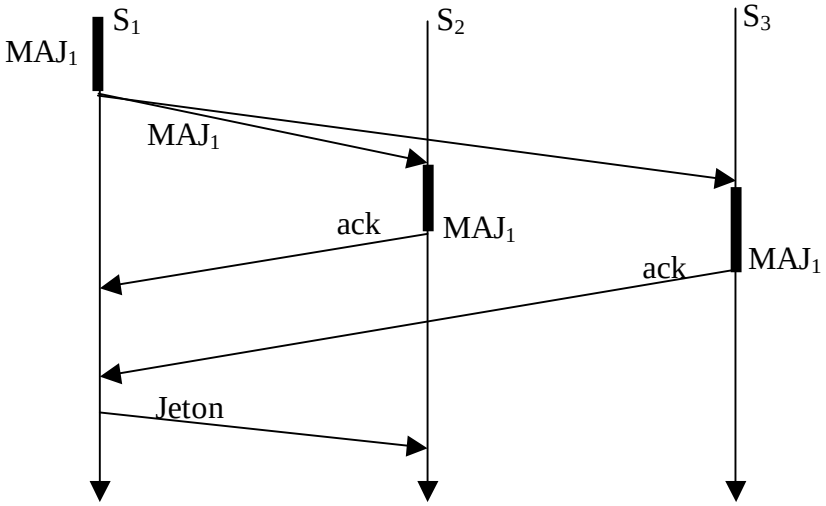


Figure 2.9 : Un mécanisme de mise à jour d'une base de données dupliquée

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.

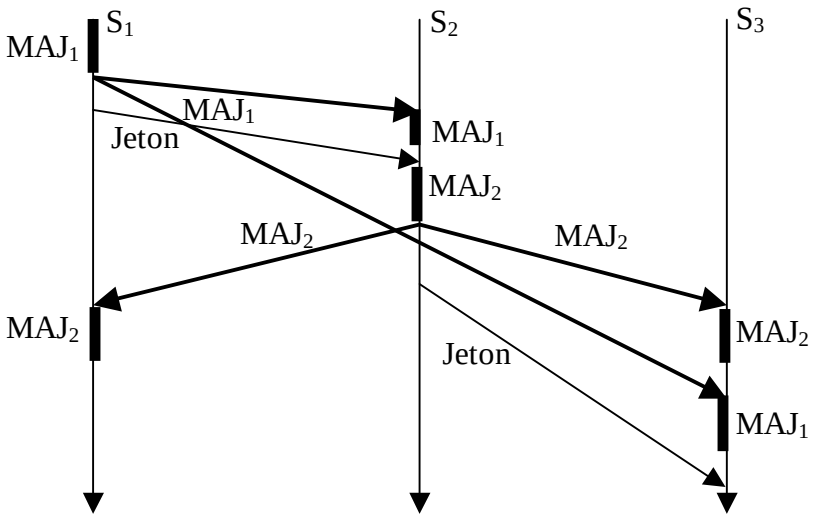


Figure 2.10 : Deux transactions conduisant à des copies incohérentes

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.

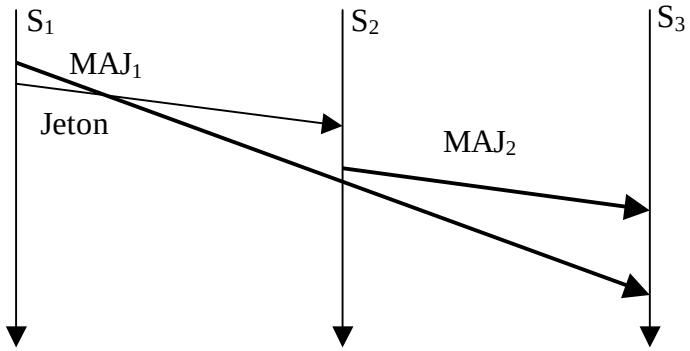


Figure 2.11 : Un comportement "pathologique" du réseau

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.

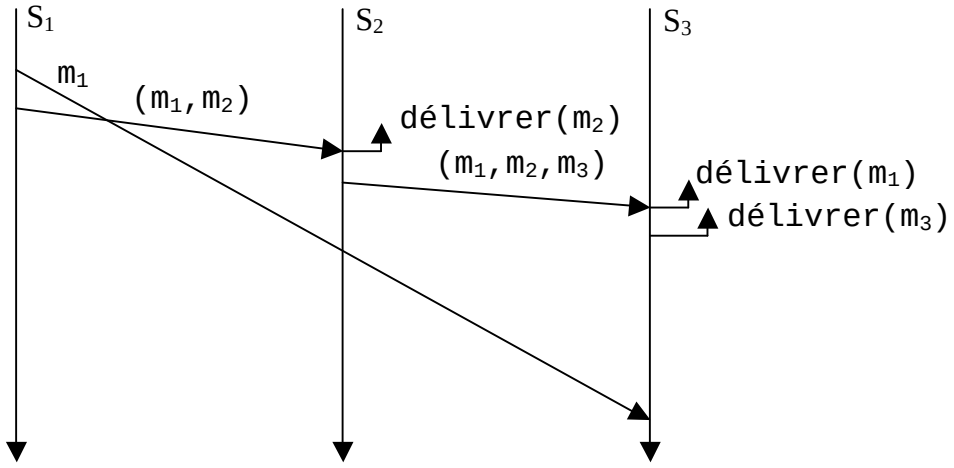


Figure 2.12 : Elimination du comportement pathologique

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.

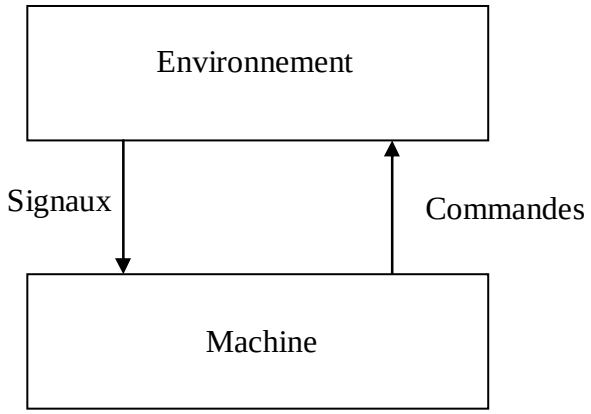


Figure 3.1 : Une application temps réel

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).

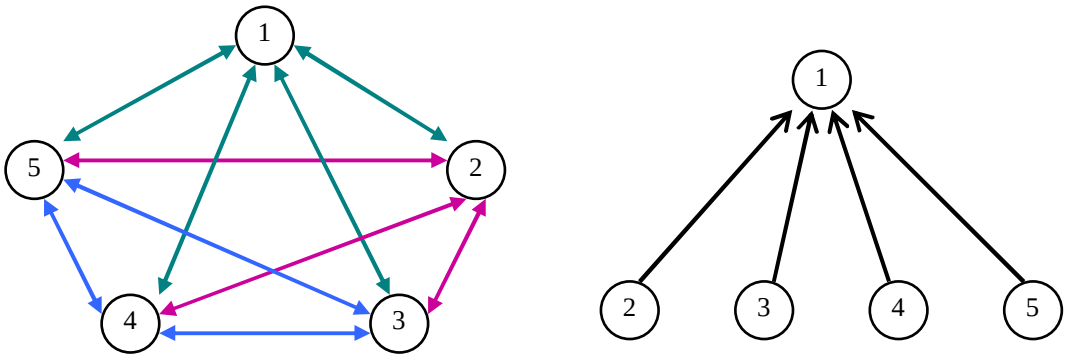


Figure 3.2 : Arbre de plus courts chemins d'une clique

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 $\rightarrow$.

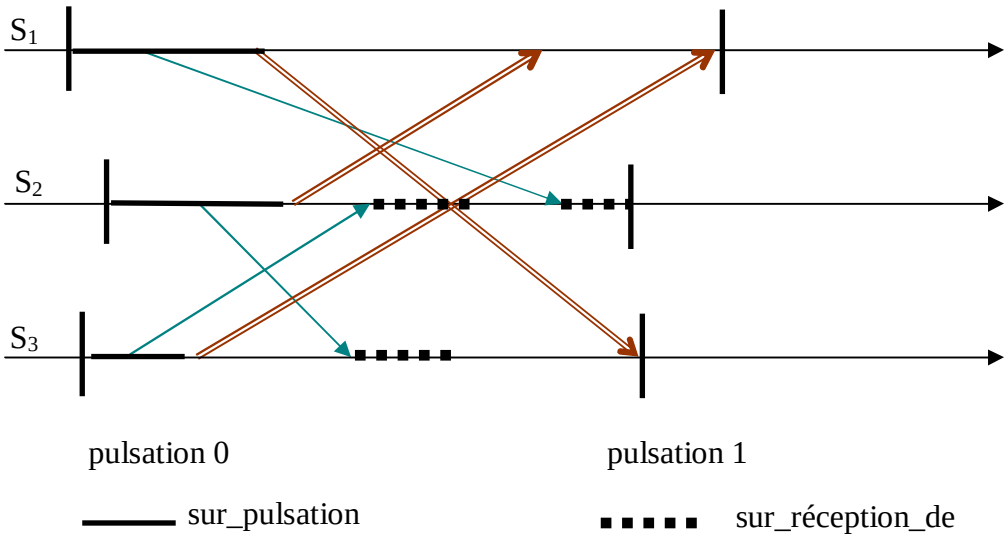


Figure 3.3 : Quand battre la pulsation?

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.

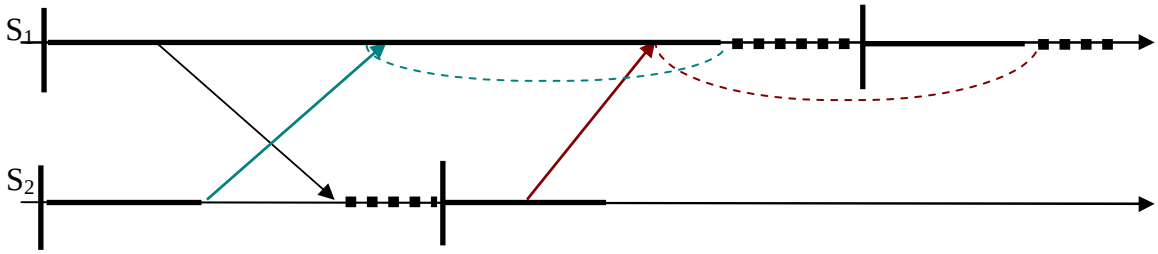


Figure 3.4 : Combien de messages conserver ?

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.

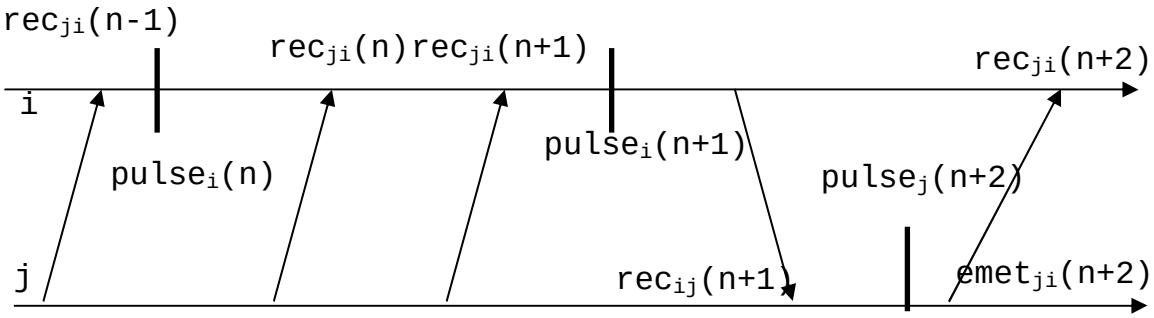


Figure 3.5 : Au plus deux messages à conserver par pulsation

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

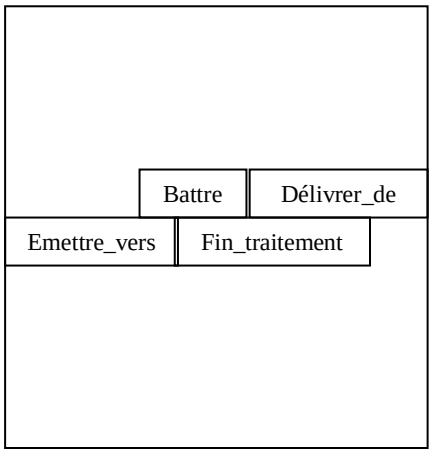


Figure 3.6 : Interface application / service

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 ρ. Autrement dit :

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

La valeur de ρ est donnée par le constructeur dans la spécification du matériel. Généralement ρ 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 δ_{min} et supérieurement par δ_{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.

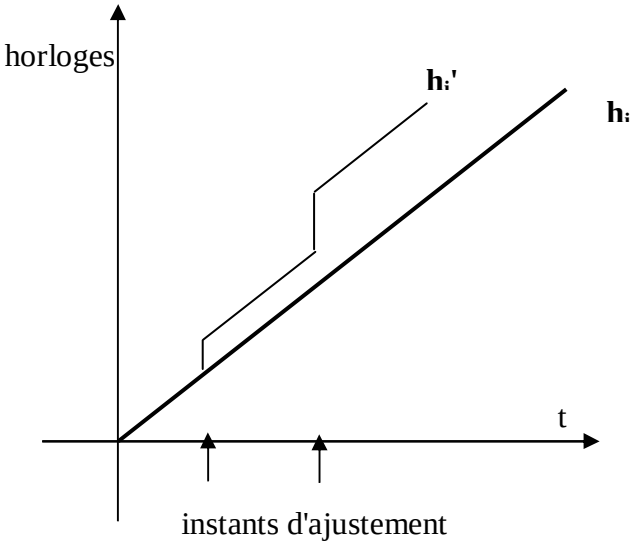


Figure 3.7 : Effet de l'ajustement (h_i' horloge ajustée)

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 :

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

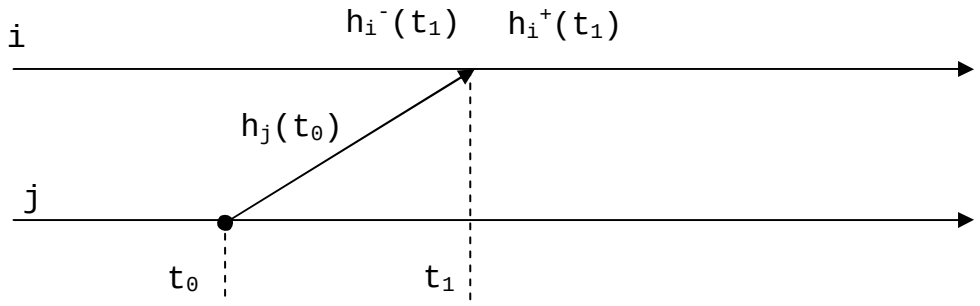


Figure 3.8 : Ajustement d'une horloge

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

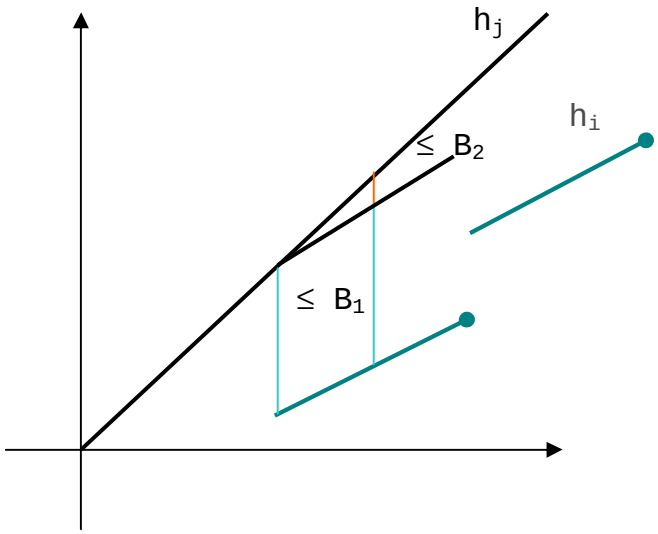


Figure 3.9 : Ecart entre deux horloges

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 :

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

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

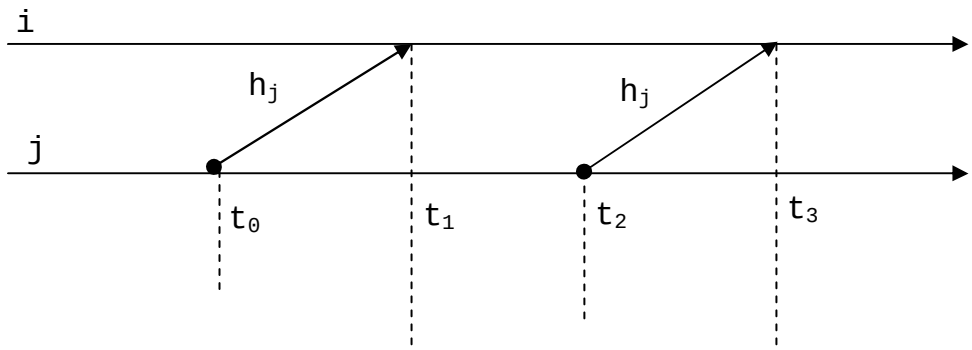


Figure 3.10 : Deux diffusions d'horloge

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.

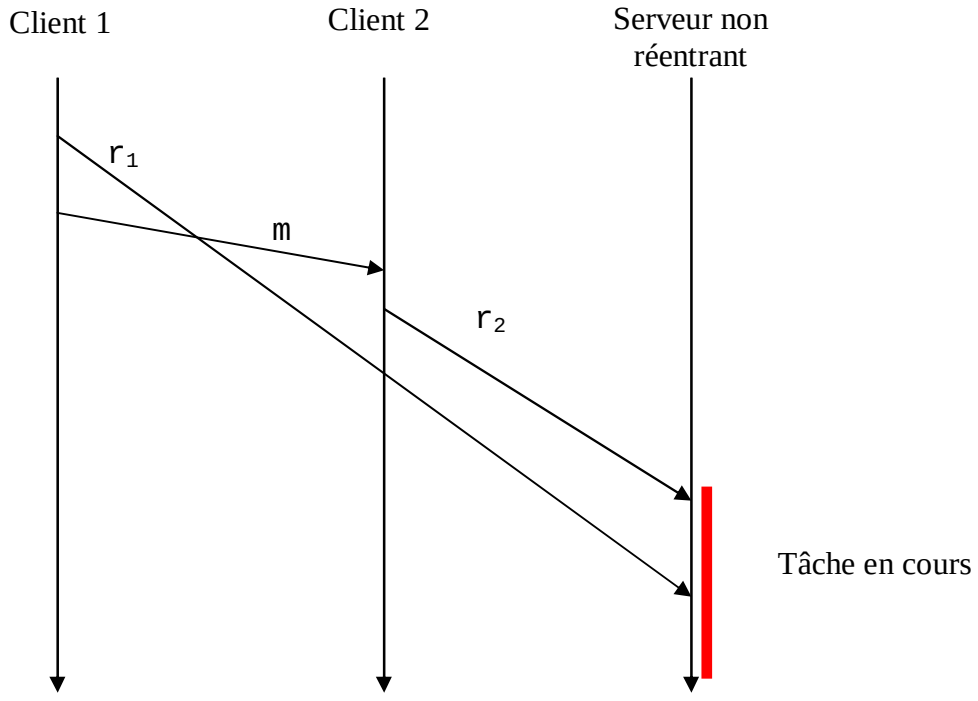


Figure 3.11 Un serveur non réentrant

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

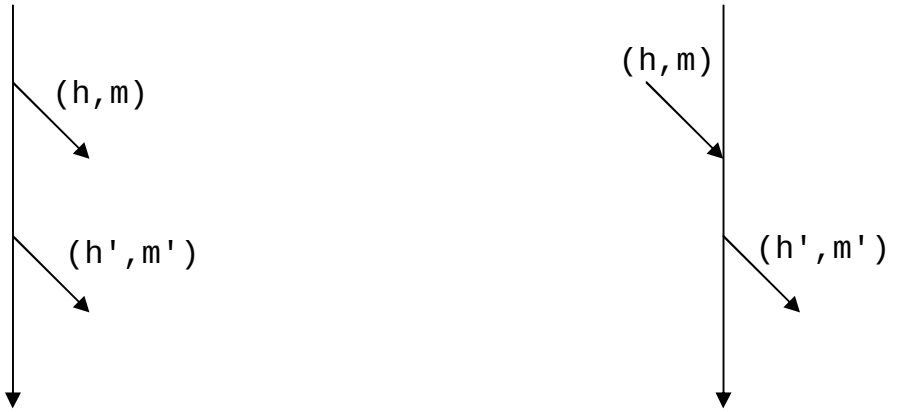


Figure 3.12 Les deux cas de l'ordre immédiat

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.

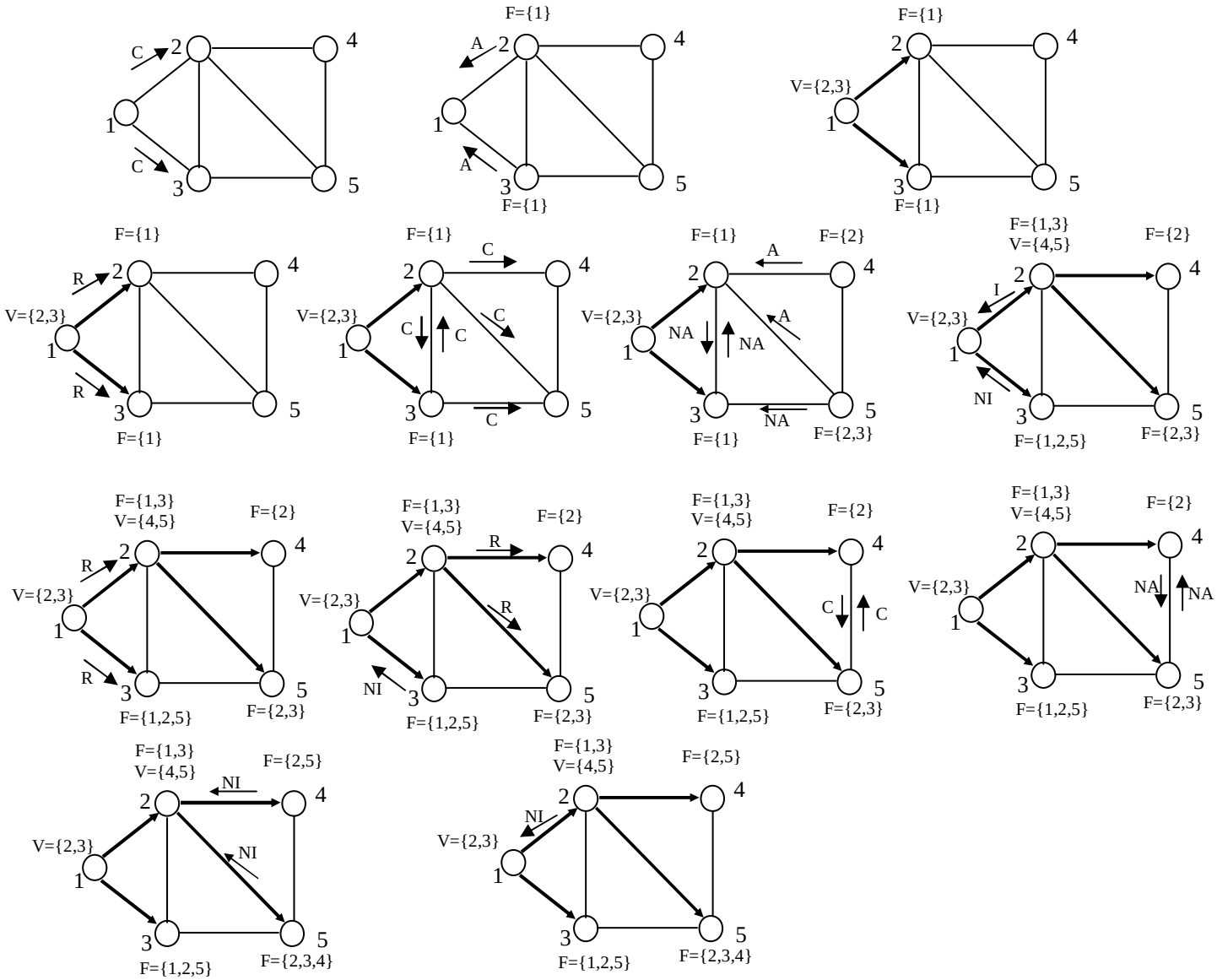
Durant la première ronde, l'initiateur envoie à tous ses voisins un `(cons,1)` qui provoque l'envoi d'un `(acq,1)` par ses voisins. A de la réception du dernier `(acq,1)`, l'initiateur démarre la deuxième ronde. Remarquons que ceci correspond au fait que le tableau $\texttt{fils}_i$ ne contient plus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

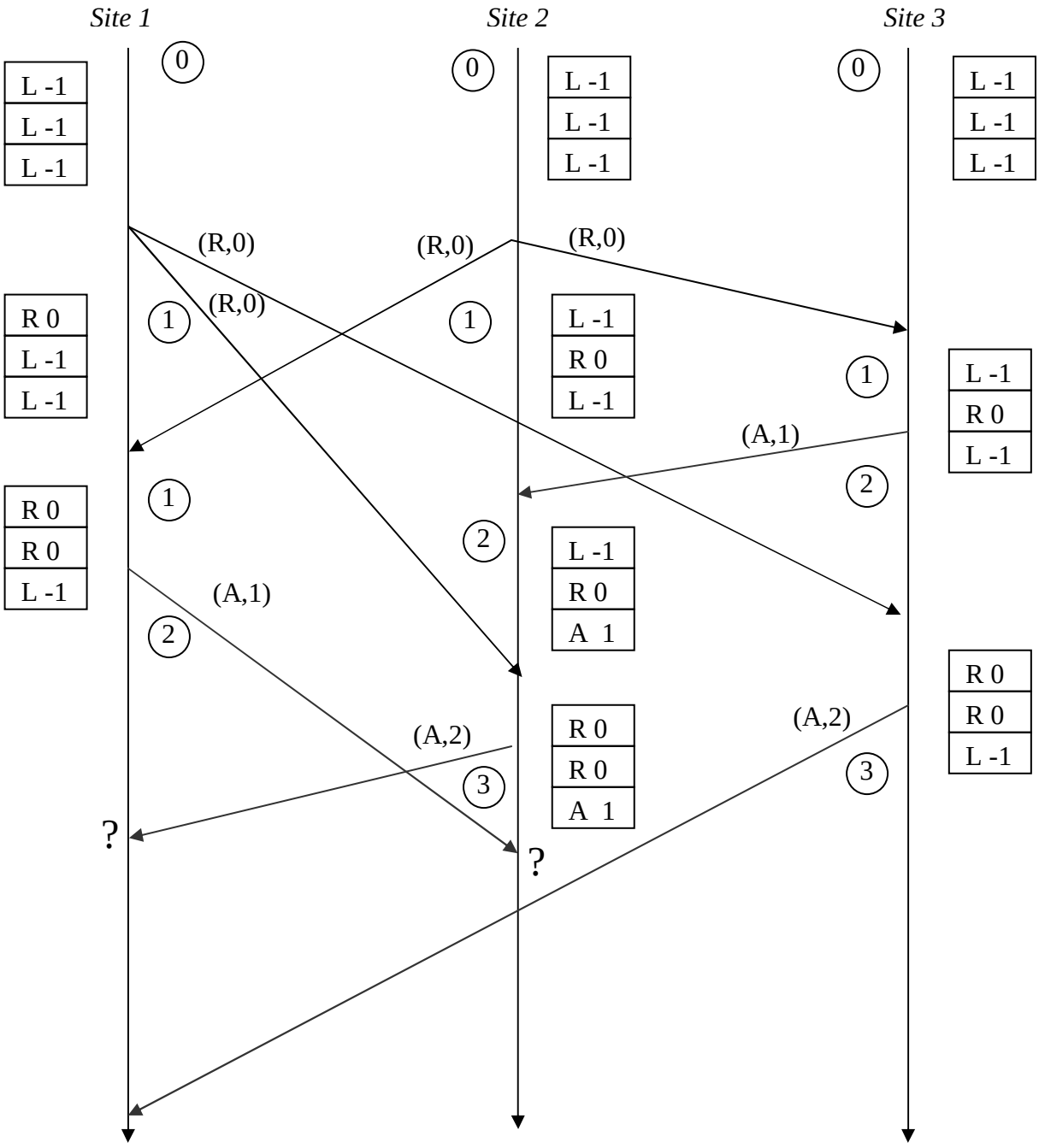


Figure 4.1 : Un scénario d'entrée en section critique (début)

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.

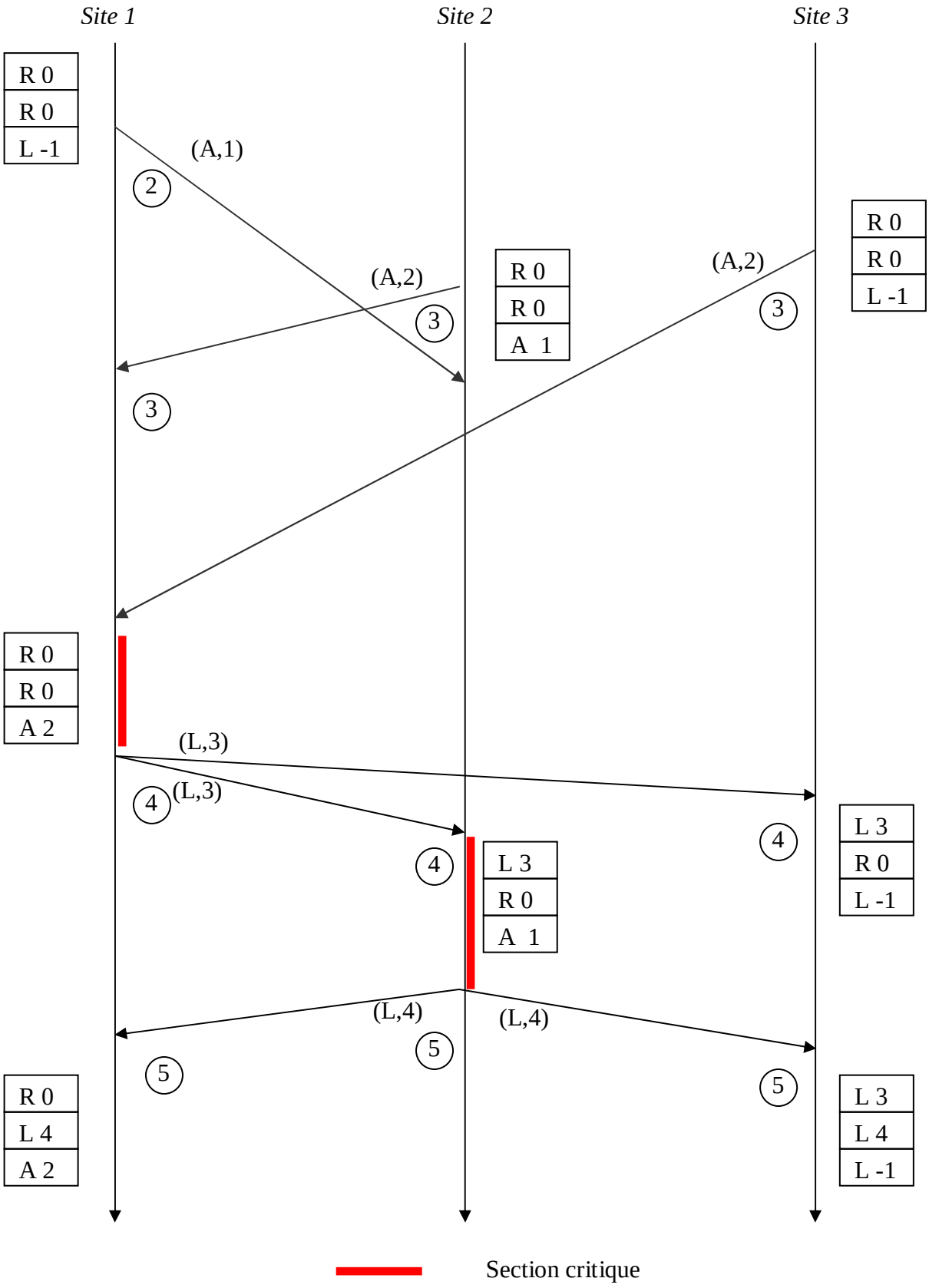


Figure 4.2 : Un scénario d'entrée en section critique (suite)

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.

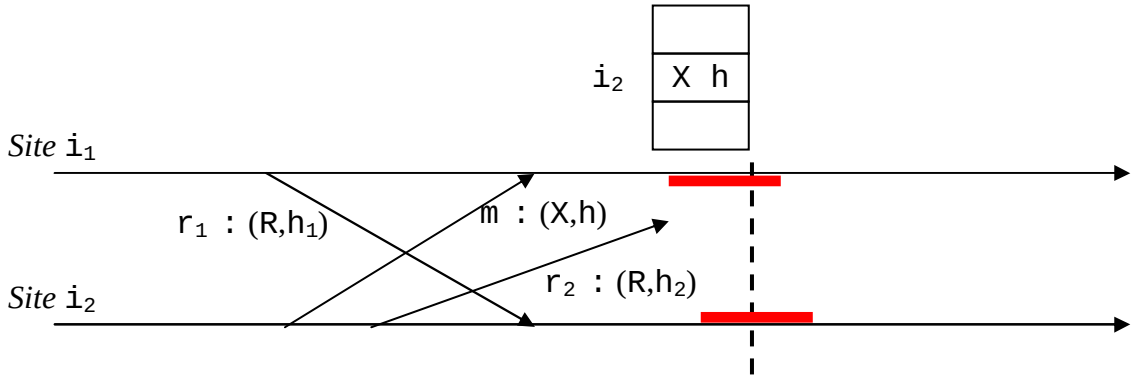


Figure 4.3 : Deux sites simultanément en section critique

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.

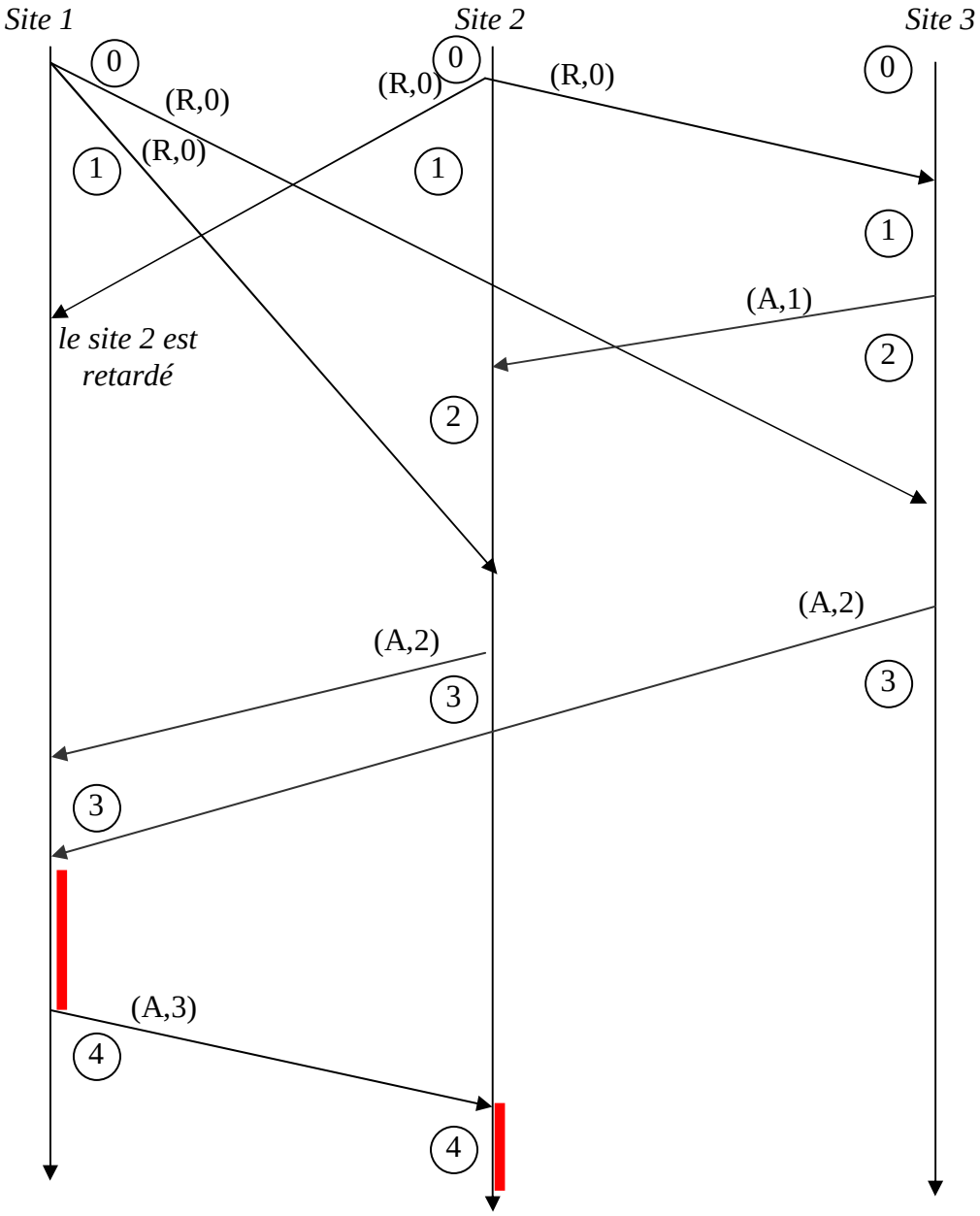


Figure 4.4 : Un scénario d'entrée en section critique

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.

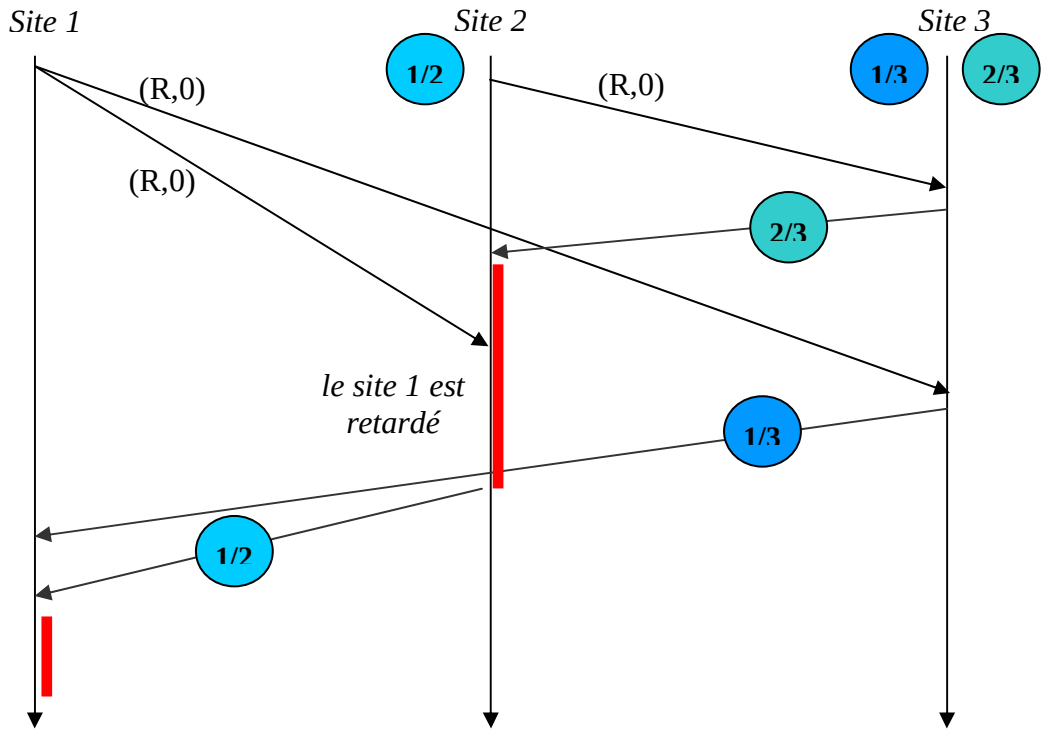


Figure 4.5 : Une requête retardée par une requête plus jeune

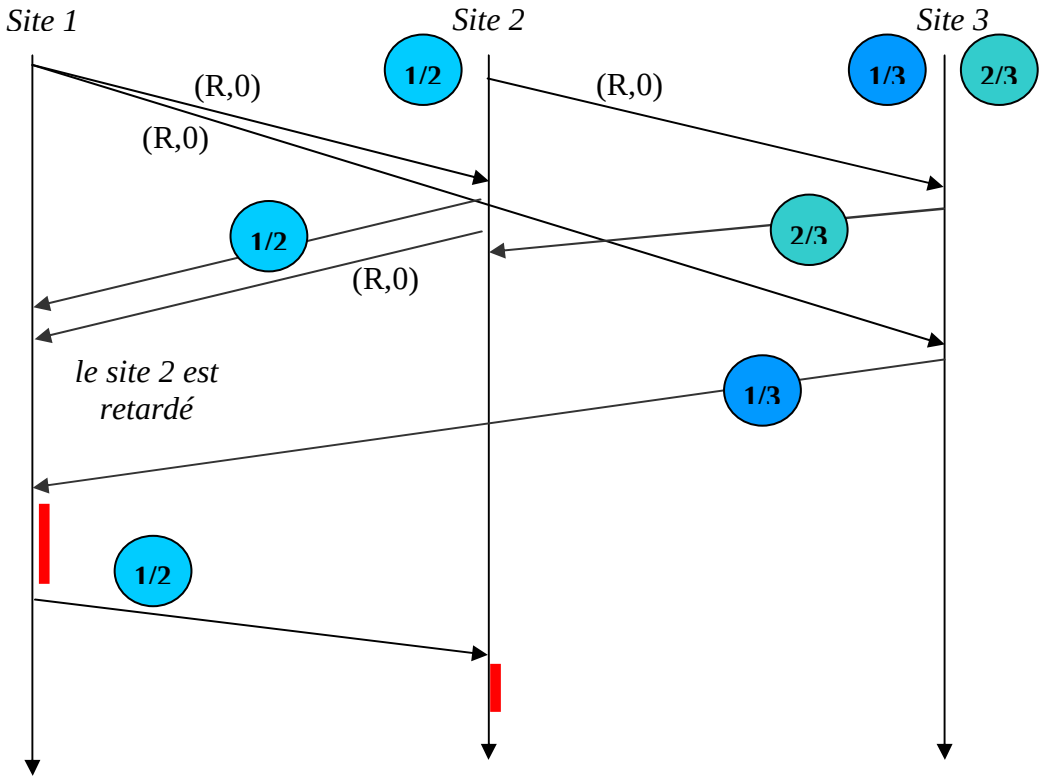


Figure 4.6 : Un site qui donne un jeton et le réclame aussitôt

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**.

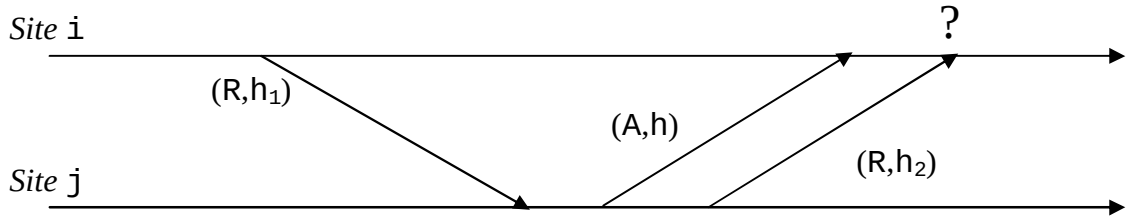


Figure 4.7 : Un jeton récupéré et réclamé une nouvelle fois

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.