Précis de répartition

Département
Informatique et Mathématiques appliquées
E.N.S.E.E.I.H.T.

Gérard Padiou

17 juin 2016

Préambule

Le document suivant n'est qu'un **résumé très condensé** de ce que l'on appelle la *répartition* dans le domaine de l'informatique. De surcroît, ne seront envisagés que deux aspects : d'une part, l'algorithmique répartie qui a donné naissance à tout un courant de recherche d'algorithmes nouveaux pour programmer des applications réparties, et d'autre part, quelques services de base qui existent maintenant dans la plupart des systèmes d'exploitation ou intergiciels prenant en compte la répartition.

Pour les lecteurs intéressés, la lecture des livres suivants est recommandée :

- pour l'aspect algorithmique :
  - Un premier essai de bonne facture :
    Michel Raynal, *Algorithmes distribués et protocoles*, éditions Eyrolles, 1985.
  - Une suite intéressante :
    Michel Raynal, *Synchronisation et état global dans les systèmes répartis* et *Gestion des données réparties*, éditions Eyrolles, 1992.
  - La bible des algorithmes répartis par une sommité du domaine, Nancy A. Lynch. Le formalisme utilisé pour la description des algorithmes est cependant difficile d'accès :
    Nancy A. Lynch, *Distributed algorithms*, Morgan Kaufmann Publichers, Inc., 1996.
- pour l'aspect systèmes répartis :
  - Une bonne introduction sur les aspects « système d'exploitation » :
    Pradeep Kumar Sinha, *Distributed Operating Systems*, IEEE Computer society press, 1996.
  - Un livre de base très complet :
    Sape Mullender, *Distributed Systems*, ACM Press Frontier Series, Addison Wesley, 1993.
  - Un livre très complet, orienté « système », bien écrit et facile d'accès :
    George Coulouris, Jean Dollimore et Tim Kindberg, *Distributed Systems : concepts and design*, Third Edition, Pearson Education Limited, 2001.
  - Une bible sur la tolérance aux fautes dans les systèmes répartis :
    Kenneth P. Birman, *Building secure and reliable network applications*, Manning Publications Co., 1996.
  - Et pour les lecteurs intéressés par la simulation répartie :
    Richard M. Fujimoto, *Parallel and distributed simulation systems*, Wiley series on Parallel and distributed computing, John Wiley & sons, Inc., 2000.

Remerciements

Ces quelques pages ne seraient pas ce qu'elles sont aujourd'hui sans les nombreux relecteurs ou relectrices qui ont détecté les erreurs, oublis, points obscurs tout au long des mises à jour annuelles.

Il y a bien sûr parmi eux des générations d'élèves du département, mais aussi, les collègues qui ont participé à l'enseignement des systèmes d'exploitation, des intergiciels et des systèmes parallèles de façon récurrente ou temporaire. En tout premier lieu, je tiens à remercier Philippe Quéinnec, Philippe Mauran et Michel Charpentier pour leurs critiques toujours constructives.

Table des matières

Chapitre 1

Définition et problématique

1.1 Introduction

La répartition des applications informatiques fait l'objet d'un fort développement depuis une trentaine d'années. Cet essor s'est accentué avec les progrès technologiques des réseaux de télécommunication. En effet, les applications réparties doivent leur naissance en tout premier lieu au mariage de l'informatique et des télécommunications.

Un point anecdotique concerne la terminologie. Doit-on parler de systèmes répartis ou de systèmes distribués. Les constructeurs d'ordinateurs ou de réseaux utilisent plutôt le terme « distribué». Les spécialistes du logiciel préfèrent le terme « réparti ». Appartenant à cette confrérie, j'utiliserai donc le mot répartition et ses dérivés. De toute façon, l'objet de ce petit précis est justement de donner une sémantique la plus précise possible à ces termes.

1.2 Les épines...

La problématique de la répartition est née avec l'idée de faire communiquer des ordinateurs via un réseau de communication. Par simple échange de messages, il est en effet possible de développer des applications qui peuvent disposer de l'ensemble des ressources de l'architecture informatique composée des nœuds « ordinateurs » et du réseau. [1]

La puissance et l'intérêt d'une telle architecture répartie est évidemment étroitement liée à la capacité de communication offerte par le réseau. De ce point de vue, les progrès technologiques réalisés en la matière ont largement contribué au développement rapides des applications réparties.

Reste le problème du développement, de la programmation des applications réparties. En effet, la conception d'applications réparties a soulevé de nombreux problèmes. Aujourd'hui, des principes et concepts, mais aussi des langages, des systèmes d'exploitation, des environnements sont disponibles. Ils sont le résultat d'une trentaine d'années de maturation à travers des centaines de projets de recherche.

Tous les problèmes ne sont pourtant pas encore résolus et les applications futures soulèveront sans doute de nouveaux défis avec en particulier le plein essor de l'informatique dans le nuage ou « *infonuagique* » alias « *Cloud computing* ».

Une question essentielle est de percevoir pourquoi un tel effort de recherche a été nécessaire. Autrement dit, pourquoi et en quoi, la programmation d'applications réparties se distingue d'une application « centralisée » ?

1. Il faut bien remarquer que cette définition exclut une architecture composée d'un ordinateur et de terminaux de saisie/affichage, même si ceux-ci sont distants.

L'origine des difficultés tient à deux hypothèses perdues et à deux propriétés affaiblies.

Pas d'état global
Pas d'horloge globale
Fiabilité relative
Sécurité relative

Nous allons préciser ces quatre points essentiels.

Pas d'état global

Dans une architecture répartie, un « nœud » physique (un ordinateur) ou logique (un simple processus) ne possède pas une connaissance immédiate exacte de l'état d'un autre nœud. En effet, cette connaissance passe par l'échange d'un message qui introduit obligatoirement un délai. Un nœud connaît son état local et n'a qu'une connaissance du passé (certes parfois très proche) des autres.

Cette difficulté conduira, en particulier, à la recherche d'algorithmes qui permettent de construire des clichés globaux qui représenteront un état passé global cohérent d'un ensemble de nœuds.

Pas d'horloge globale

Chaque nœud possède sa propre horloge pour dater les événements qui lui sont locaux. Par conséquent, si les horloges indépendantes de chaque nœud ne sont pas parfaitement synchronisées, l'ordre des événements répartis dans l'application n'est pas déductible à partir des datations locales. Cette difficulté conduira à définir des mécanismes de datation logique qui permettent de corriger ce problème.

Ces deux propriétés augmentent fondamentalement la difficulté de compréhension et d'analyse des applications réparties. **Mais deux autres propriétés** doivent être prises en compte, en l'occurrence, la fiabilité et la sécurité.

Fiablilité ... relative

Une architecture répartie est composée de nœuds actifs dont le fonctionnement correct est relativement indépendant. Si deux nœuds sont simplement reliés par un réseau de communication, l'arrêt de l'un est généralement indépendant de l'arrêt de l'autre. [2] Contrairement aux systèmes centralisés, un système réparti peut donc tolérer la défaillance de certains de ses éléments et continuer à assurer certains services à ses usagers.

Cependant, l'arrêt d'un site parmi un nombre élevé, contrairement à une unité isolée, n'est pas un événement exceptionnel. Par conséquent, il faudra que le concepteur prenne en compte ce risque non-négligeable de défaillance. Une architecture répartie est une bonne base pour développer une application tolérante aux fautes, mais le traitement des défaillances doit y être soigneusement étudié.

Sécurité ... relative

Une architecture répartie est plus difficile à protéger contre les malversations qu'une architecture centralisée. Ceci pour deux raisons :

- les points d'accès aux ressources du système global sont multiples et souvent « hors les murs ». Autrement dit, l'utilisateur doit être authentifié de façon sûre.
- le réseau de communication est un point d'accès potentiel pour toute tentative d'intrusion.

2. Excepté quelques cas très particuliers : coupure électrique globale concernant des ordinateurs en réseau local.

Plus généralement, les nœuds connectés peuvent apparaître et disparaître dynamiquement et la configuration globale évoluer dynamiquement au cours du temps. Ceci peut conduire à ne pas connaître exactement la totalité de l'architecture répartie. Cette tendance est de plus en plus forte avec les possibilités de connexion via des ordinateurs portables ou des téléphones mobiles.

Pas nouveau mais récurrent : le non-déterminisme

Enfin, il ne faut pas oublier qu'une application répartie est aussi une application parallèle asynchrone. Les nœuds s'exécutent en parallèle et chaque nœud peut lui-même comporter plusieurs activités parallèles (processus, threads). Il existe ainsi un non-déterminisme, une non-reproductibilité et une explosion combinatoire des états qui constituent les difficultés classiques de la programmation parallèle asynchrone.

1.3 Les parfums ...

Après ce qui précède, peut-on encore trouver un intérêt à la répartition ? Malgré les handicaps précédents, la réponse est sans aucun doute : oui !

Il suffit d'en montrer les principaux avantages.

Partage et mise à disposition

La répartition est avant tout un moyen de mise à disposition et donc de **partage de ressources et de services** à distance.

Cette idée de partage et de disponibilité constitue même la sémantique que l'on peut donner au mot répartition. Cette propriété est le fondement même de l'essor des systèmes répartis et de leur expression présente sous la forme du *Cloud computing*.

Exemples

- Partage de ressources physiques : une ressource telle qu'une imprimante peut être rendue accessible à tous les usagers d'une architecture répartie. Elle peut même constituer un nœud du réseau.
- Partage de ressources logiques : des fichiers localisés sur un disque d'un nœud peuvent être rendus visibles voire modifiables par des usagers connectés à un nœud distant. Un service de gestion de fichiers « répartis » est un des services de base des systèmes d'exploitation répartis.
- Partage d'informations : un serveur de pages Web est accessible à partir de son identifiant [3] » `http://`... sans que l'on sache ni sa localisation physique, ni le système d'exploitation support, ni la machine serveur. Il est implicitement à la disposition de tous ceux qui peuvent connaître son identifiant.

Répartition géographique

La répartition est un moyen essentiel pour mettre à disposition des usagers les moyens informatiques locaux dont ils ont besoin tout en gardant la possibilité d'accéder aux ressources et services distants de leurs collègues.

Exemples

- Un système de réservation (d'hôtels, d'avions, de trains, ou de tout à la fois) est un exemple de système qui peut avoir plusieurs centres dans différents lieux, par pays, par compagnie, par agences.
- Un système bancaire peut comporter un ensemble d'agences régionales qui communiquent avec le siège central.
- Le contrôle aérien peut nécessiter un découpage en secteurs dont il faudra coordonner les interactions.

3. URL : Uniform Resource Locator

Autres avantages complémentaires

Moins essentiels mais non négligeables, on n'oubliera pas les points complémentaires suivants :

- **Puissance de calcul** La connexion de machines en réseau permet d'obtenir à moindre coût une puissance de calcul importante. Bien entendu, l'exploitation optimale de ces performances potentielles nécessite de paralléliser les algorithmes de calcul et de disposer d'un environnement d'exécution répartie adéquat. En la matière, quelques produits de ce type sont aujourd'hui largement diffusés (PVM, MPI par exemple).

- **Disponibilité** La disponibilité d'un service développé sur une architecture répartie peut être rendue plus grande que celle d'un service centralisé. La relative indépendance des défaillances qui peuvent survenir dans les nœuds, permet de continuer à offrir un service même dégradé. En particulier, la réplication d'un même service par l'installation de plusieurs serveurs équivalents est une solution classique mais de mise en œuvre délicate notamment si les serveurs sont à état rémanents (via le partage d'une base de données par exemple).

- **Flexibilité** Une architecture répartie est par nature modulaire. Il est donc plus facile d'ajouter ou d'enlever un nœud connecté au réseau, le système global continuant à être disponible. Cette caractéristique tend à jouer un rôle important avec la mobilité de plus en plus grande des usagers.

Ce sont maintenant des ordinateurs portables mais surtout des téléphones mobiles ou des tablettes qui se connectent par un réseau de télécommunication pour accéder à un système réparti aux frontières parfois mal connues (par exemple le réseau Internet).

Les points d'accès sont donc mobiles. On parle d'informatique nomade. La connexion est temporaire et doit être même minimisée à cause des coûts de communication non négligeables. Les problèmes posés concernent donc essentiellement la localisation, l'identification et l'authentification des usagers « mobiles » et l'optimisation des temps de connexion (par l'utilisation d'agents mobiles par exemple).

Cette flexibilité se situe aussi au niveau logique. Un nouveau service peut être installé sur un nœud sans nécessiter une reconfiguration des autres nœuds.

Autrement dit, un système réparti peut en principe mieux s'adapter d'une part, à une croissance des besoins informatiques d'une entreprise en permettant une adaptation dynamique et continue et d'autre part, au nombre variable et à la mobilité de ses usagers.

Ces caractéristiques et possibilités sont essentielles et ont donné naissance à « *l'infonuagique* ». Maintenant, des machines, des systèmes, des logiciels et des services peuvent être mis à disposition d'usagers quand ils en ont besoin et en s'adaptant à leur demande. Ces usagers distants ignorent où se trouvent ces ressources physiques (architectures informatiques) ou logiques (systèmes de fichiers, services de stockage, services Web, etc) qu'ils utilisent.

Les avantages prédédents ont fait le succès des systèmes répartis. Ils permettent aussi de donner une définition plus précise du mot répartition.

> La répartition est la mise à disposition
> d'un ensemble de ressources et services
> accessibles via un réseau de télécommunication
> pour tous les usagers
> qui possèdent un droit d'accès en un point quelconque.

1.4 Comprendre, conceptualiser, expérimenter ...

La recherche de solutions, face aux difficultés intrinsèques posées par la répartition, s'est déployée dans différents domaines et selon différents points de vue complémentaires : analytique, conceptuel, expérimental. Du domaine le plus abstrait au plus technique, les recherches sur la notion de calcul, de systèmes, d'applications répartis ont portés sur les points suivants.

1.4.1 La théorie

Pour bien comprendre la problématique de la répartition, il est important de pouvoir modéliser le plus formellement possible sa sémantique. Il s'agit donc d'élaborer des modèles de calcul traduisant de façon abstraite et formelle (mathématique) les propriétés d'un calcul réparti.

Une première école a pour origine les travaux de Milner[19] et Hoare[14]. Elle s'appuie sur la définition d'algèbres de processus communicants. Un calcul réparti est un terme algébrique qui décrit un ensemble de processus qui sont autant de termes composants. La communication entre processus apparaît sous la forme d'un rendez-vous, c'est-à-dire une interaction élémentaire atomique sous la forme de l'échange synchrone d'un message.

Une deuxième école s'appuie sur le concept d'acteur. Le protocole de communication entre acteurs est asynchrone, chaque acteur possédant une boite à lettres pour recevoir les messages des autres acteurs. Le typage des messages permet d'associer un comportement « réactif » spécifique et modifiable dynamiquement à la réception de chaque message. Ce modèle d'acteurs communiquant par messages a lui aussi été formalisé.

Enfin, les systèmes de transitions et la logique temporelle sont aussi utilisables et utilisés pour spécifier et modéliser des traitements répartis. Dans ce cas, la communication est souvent modélisée simplement sous la forme de séquences de messages.

1.4.2 L'algorithmique

L'écriture d'algorithmes adaptés à une architecture répartie soulève des problèmes algorithmiques spécifiques[24]. Ceux-ci ont pour origine les deux hypothèses « perdues » bien utiles au programmeur. Un courant de recherche s'est donc développé pour étudier et proposer des algorithmes répartis apportant en particulier des solutions à des problèmes **génériques**.

Une importante classe d'algorithmes concerne la définition de protocoles de communication point à point ou à diffusion. En effet, pour pouvoir coopérer via des échanges de messages, les partenaires doivent se mettre d'accord sur le séquencement et le type de ces messages. La définition de nouveaux protocoles de communication est donc une activité importante.

Sous une simplicité apparente, la spécification de protocoles est une activité délicate qui a nécessité la recherche de formalismes de description (automates communicants, réseaux de Petri, langages LOTOS ou SDL,...) et d'outils d'aide à la validation. Certains standards existent comme par exemple, l'appel procédural à distance. Mais beaucoup de protocoles sont adaptés à une classe spécifique d'applications (par exemple, le temps réel).

On trouve ensuite des problèmes plus généraux que l'on peut classer en deux catégories : d'une part, des problèmes connus du monde parallèle (exclusion mutuelle, interblocage, atomicité, réplication par exemple) mais qui ont dû être reconsidérés et d'autre part, des problèmes nouveaux apparus avec la répartition des traitements et des données.

On peut citer par exemple le problème de la terminaison d'une application répartie[10] (problème simple en centralisé, mais délicat en réparti) ou celui de l'interblocage[8], le calcul d'états globaux (appelés clichés)[25], la détection d'états stables ou inévitables, la réalisation d'un consensus[13] [4] , ...

4. Ce problème consiste à obtenir qu'une même décision soit adoptée par tous les participants (ici processus)

1.4.3 Les langages de programmation

À première vue, la répartition n'implique pas de grande révolution dans les langages de programmation. En effet, dans un effort minimaliste, il suffit d'utiliser une interface de programmation (API) permettant d'échanger des messages, soit deux primitives *émettre* et *recevoir* un message. L'interface à base de « prises » (en anglais « sockets ») offerte par le système `Unix` n'en propose pas beaucoup plus encore aujourd'hui.

On peut cependant être plus ambitieux en proposant par exemple des structures de contrôle facilitant le raisonnement et la programmation des échanges de messages. De nombreuses propositions ont été faites. Il est difficile d'en donner une liste exhaustive.

Les approches gardant une communication explicite par messages, ont introduit la notion de non-déterminisme en réception voire en émission. Le non-déterminisme en réception est en effet important car l'on doit souvent exprimer l'attente d'un message parmi plusieurs possibles d'origine et de type différents. Il est alors intéressant de posséder une structure de contrôle permettant d'associer à chaque type de message une action spécifique. On trouve ce type de construction dans des langages comme OCCAM et Ada[15].

Enfin et surtout, une extension de l'appel de procédure a été proposée, l'**appel procédural à distance**. Le programmeur garde son modèle de programmation de type procédural, mais, en fait, les procédures appelées par une processus donné peuvent être localisées sur des sites distants. Ce modèle de programmation réparti est celui qui est aujourd'hui le plus utilisé et connu sous le nom de modèle **client-serveur**.

Au niveau langage, il faut introduire un langage de définition d'interface (IDL [5]) qui permet de décrire l'interface des procédures qui doivent pouvoir être appelées à distance. Les problèmes d'implantation sont, autant que faire se peut, masqués au programmeur par une génération automatique des routines de traitement des appels côté client (appelant) et serveur (appelé) à partir des descriptions en langage IDL.

1.4.4 Les systèmes d'exploitation

Les systèmes d'exploitation sont évidemment les premiers concernés par la répartition. Il leur revient d'assurer l'interface avec le medium de communication.

Les concepteurs de systèmes ont envisagé deux approches possibles :

- reprendre le problème à la base et concevoir de nouveaux noyaux d'exécution répartie à partir d'une architecture matérielle. L'avantage d'une telle approche est de réduire la taille du noyau d'exécution, qualifié de micro-noyau, en limitant celui-ci aux fonctionnalités de base : gestion des ressources locales (mémoire, périphériques), gestion du parallélisme et de la communication. Les services classiques d'un système d'exploitation (tel que la gestion des fichiers) deviennent simplement des services hors du noyau d'exécution proprement-dit. La modularité du système obtenu est de ce fait beaucoup plus élevée que celle d'un système d'exploitation centralisé. L'inconvénient d'une telle approche est qu'il faut redémarrer à zéro. On peut citer une réussite française en la matière avec le système CHORUS qui est un micro-noyau d'exécution répartie orienté Temps Réel.
- étendre les systèmes d'exploitation centralisés en ajoutant au minimum une interface de communication et si possible quelques services répartis, par exemple, un système de gestion de fichiers répartis. L'avantage de cette approche est la continuité et la réutilisation. C'est ainsi par exemple que le système `Unix` a été progressivement étendu avec une interface de communication par sockets, par RPC [6][21] avec ses différentes sémantiques, puis par un service de fichiers répartis (NFS [7]).
  L'inconvénient de cette approche est le caractère moins modulaire du système obtenu. Les services offerts, comme la gestion des fichiers, sont inclus dans le noyau et donc pratiquement figés.

En réalité, les deux approches se sont mutuellement fertilisées dans la mesure où les concepts introduits dans les micro-noyaux répartis ont amené à modifier la conception des systèmes classiques en essayant de leur donner une plus grande modularité.

5. IDL : Interface Definition Language
6. Remote Procedure Call
7. Network File System

1.4.5 Les environnements d'exécution répartie (middleware)

Un problème de base des systèmes répartis est de maîtriser l'hétérogénéité aussi bien matérielle que logicielle (systèmes d'exploitation, langages de programmation par exemple). L'objectif est de pouvoir faire communiquer et coopérer des composants a priori hétérogènes.

Pour résoudre ce problème, le modèle adopté repose sur d'une part, le schéma de communication client/serveur et d'autre part, sur la notion de « bus logiciel ». Un bus logiciel doit permettre d'accéder à des services spécifiés par leur interface. Ces interfaces sont enregistrées dans des services d'annuaires qui permettent de trouver le ou les nœuds serveurs (qui auront été au préalable répertoriés).

Une norme de fait existe actuellement avec l'environnement CORBA (Common Object Request Broker) défini par l'OMG (Object Management Group Architecture). Une spécification d'un bus logiciel à objet a été définie et implantée par de nombreux développeurs. Cette couche logicielle se place donc entre le système d'exploitation et les applications.

Une seconde approche est l'approche interprétée diffusée largement sous la forme de l'environnement Java. On définit un environnement d'exécution exécutable par une machine abstraite qui sera implantée par un interpréteur. Il suffit d'avoir une version de cet interpréteur exécutable sous le système d'exploitation que l'on utilise pour pouvoir développer des applications réparties selon le modèle client/serveur. L'objectif est clair : **développer une seule fois et exécuter n'importe où**[8]. Le modèle d'exécution offrant l'appel procédural à distance, le développement d'applications réparties est immédiate et l'hétérogénéité des systèmes d'exploitation et machines supports est masquée.

1.5 Principe de conception

La démarche adoptée par les concepteurs de systèmes répartis repose sur un paradoxe un peu décevant. Pour caricaturer, on pourrait résumer leur principe fondamental par :

> Pour concevoir un bon système réparti,
> concevez un système qui a « l'air centralisé »
> (qui s'utilise comme un système centralisé)

Autrement dit, l'objectif est de faire en sorte que le système réparti ait une interface la plus proche possible de celle d'un système centralisé. Pourquoi une telle approche ? La raison en est simple : le programmation sous les hypothèses centralisées (horloge globale et mémoire globale) est plus simple que la programmation sous les hypothèses réparties. Par conséquent, il vaut mieux « masquer » le plus possible les propriétés délicates de la répartition.

En fait, il est impossible de masquer totalement la répartition. D'une part, parce qu'il faut parfois savoir où est physiquement la ressource que l'on utilise (par exemple, une imprimante), d'autre part et surtout parce que la fiabilité du système global n'est pas suffisante.

À titre d'exemple, considérons le mécanisme d'appel procédural à distance (RPC). Ce mécanisme permet d'utiliser le concept bien connu de procédure en l'étendant au contexte réparti. La procédure appelée peut être exécutée sur un site différent de celui de la procédure appelante.

Cette extension permet de programmer simplement des traitements répartis mettant en jeu plusieurs nœuds en conservant le style de programmation procédurale. Cependant, la fiabilité d'un appel réellement distant est bien inférieure à celle d'un appel procédural local (centralisé). La mise en œuvre du mécanisme de RPC nécessite des échanges de messages, met en jeu deux sites distincts, pose des problèmes de conversion de format des données, de fiabilité dans l'échange des messages, ...

8. selon le slogan de SUN, créateur de l'environnement Java : Write Once, Run Anywhere

Le résultat est qu'il faut que le programmeur prévoit, par exemple, qu'un appel peut échouer parce que le site appelé est arrêté. Pire, la détection de cette événement peut être erronée : l'appelant considère que l'appel a échoué, mais en réalité, le site appelé a bien exécuté la procédure.

Dès lors, quelles sont les propriétés qui peuvent masquer en partie ou totalement la répartition des données et des traitements ? On appelle de telles propriétés, des propriétés de transparence. Elles ont été hiérarchisées un peu comme les « couches » de protocole dans le domaine des réseaux de communication. Nous parlerons plutôt de niveau, il y en aura huit au lieu des sept couches du modèle ISO et nous les exposerons du plus bas au plus haut niveau de la hiérarchie.

1.5.1 Transparence d'accès

Cette propriété, au plus bas niveau de la hiérarchie, consiste à assurer que l'interface d'accès à une ressource (fichier par exemple), sera identique que la ressource soit locale ou distante.

Exemples

- Considérons un opération d'ouverture d'un fichier sous `Unix` :

  ```
  fd = open("projet/test.data",...) ;
  ```

  Que le fichier spéficié soit local ou distant, le sous-système de gestion de fichiers répartis NFS permet d'utiliser la même primitive `open`. On a bien la transparence d'accès.

- Considérons une commande d'impression sous `Unix` :

  ```
  lp -d lj  test.data
  ```

  L'imprimante peut être connectée localement ou distante, la commande utilisée restera `lp` ;

- Considérons une opération de connexion à distance :

  ```
  rlogin   blaise.irit.fr   -l arthur
  ```

  La transparence d'accès n'est pas vérifiée puisqu'on utilise une commande différente de la commande locale.

Ce type de transparence élémentaire est maintenant bien maîtrisé et implanté sous la forme du modèle client/serveur. Le mécanisme utilisé est l'appel de procédure à distance. Il en existe de nombreuses implantations dans des environnements différents d'exécution. On retiendra le contexte des intergiciels tels que CORBA ou celui du Web avec le protocole SOAP [9] utilisant XML par exemple.

1.5.2 Transparence de localisation

Cette propriété consiste à assurer que la désignation de la ressource est indépendante de la localisation de cette ressource. Autrement dit, les usagers peuvent ignorer la localisation réelle. Si la transparence d'accès est de plus assurée, on peut alors utiliser une ressource en ignorant si elle est locale ou distante.

Cette propriété peut être affinée en assurant que tous les usagers de la ressource pourront la désigner par le même nom global indépendamment de leur propre localisation. La désignation est alors qualifiée d'uniforme.

9. Simple Object Acces Protocol.

Exemples

- Si l'on reprend l'opération d'ouverture d'un fichier,

  ```
  fd = open("projet/test.data",...) ;
  ```

  NFS assure la transparence de localisation. [a]

- Pour la commande d'impression suivante :

  ```
  lp -d lj  test.data
  ```

  l'imprimante porte un nom symbolique qui ne contient pas d'information sur sa localisation (pour ce type de ressource, il est bien entendu utile de savoir finalement où elle se trouve) ;

- Pour l'opération de connexion à distance :

  ```
  rlogin   blaise.irit.fr   -l arthur
  ```

  la transparence de localisation n'est évidemment pas possible.

a. Attention, NFS n'assure pas forcément une désignation uniforme. Le même fichier sera éventuellement accessible sous des noms différents selon la localisation de l'appelant.

La transparence de localisation est une propriété fondamentale. Le mécanisme sous-jacent à son implantation est celui de service d'annuaire. Des serveurs de noms assurent la conversion d'un nom symbolique à un nom interne permettant de localiser finalement la ressource sollicitée.

Les services d'annuaire sont essentiels pour la couche de communication permettant par exemple, le passage d'un nom de site symbolique `www.enseeiht.fr` à une adresse IP.

De façon similaire, un appel de méthode à distance de type RMI [10] de Java désigne l'objet cible sous une forme symbolique, par exemple `Objet_X` et nécessite d'obtenir une référence interne localisant à la fois la machine distante support et le processus serveur gérant l'objet concerné sur cette machine.

Il est à noter que dans la plupart des cas, le serveur d'annuaire est lui-même un objet accessible à distance qui offre donc des méthodes d'enregistrement (`bind`) et de consultation (`lookup`) appelées à distance.

Problème : comment obtient-on une référence permettant d'accéder à cet objet puisqu'on ne peut pas s'adresser au serveur lui-même (récursivité) ? La solution repose sur une désignation plus « directe » du serveur, par exemple en fournissant le nom de machine et le numéro de port d'écoute du processus serveur.

1.5.3 Transparence du partage

Cette propriété consiste à assurer que les accès concurrents à la ressource seront contrôlés de telle façon que l'intégrité de la ressource soit assurée. À titre d'exemple, pour un fichier, la transparence du partage implique de garantir les règles de synchronisation dites des lecteurs/rédacteurs. Pour une imprimante, il s'agit d'éviter les mélanges des requêtes d'impression en assurant un accès exclusif.

Les systèmes d'exploitation répartis se contentent en général d'assurer le minimum vital. Si le partage d'une imprimante est garanti sans problème (principe du spooling), le partage des fichiers est plus problématique. À titre d'exemple, le système NFS essaie de minimiser les risques d'incohérence, mais sans garantie absolue (cf. fichiers répartis). Une telle approche tient au coût de supervision du partage pour des fichiers qui, dans leur grande majorité, ne seront que des fichiers temporaires non partagés.

Ce sont donc des systèmes plus dédiés à un contexte d'application particulier qui garantissent cette transparence. Le contexte le plus répandu est celui des bases de données réparties et des sous-systèmes transactionnels associés. Les enregistrements d'une base de données sont bien des ressources fortement partagées par nature. Il est donc important d'offrir, dans un contexte réparti, le même type de service que celui apporté par les systèmes transactionnels centralisés. Le problème de base est celui de l'atomicité des traitements (transactions) sur la ou les bases de données.

10. Remote Method Invocation

En conclusion, la transparence du partage est une propriété difficile dans un contexte réparti. Elle passe par des mécanismes de synchronisation pessimistes (verrouillage par exemple) ou optimistes (détection des conflits et annulation de certaines opérations) qui sont coûteux et peu adaptés à un contexte temps réel. Indépendamment de la répartition, c'est encore un problème clé du parallélisme et un défi pour l'avenir compte tenu de l'évolution des machines vers des architectures comportant un très haut degré de parallélisme.

1.5.4 Transparence de la réplication

Cette propriété consiste à assurer que l'accès à une ressource soit identique quel que soit la forme d'implantation de cette ressource, en particulier répliquée. C'est la même idée que la transparence d'accès mais étendue à la réplication.

Ce genre de transparence est évidemment dédié à des systèmes très spécifiques. La réplication a deux objectifs complémentaires :

- améliorer les performances en offrant la possibilité d'accéder à une copie éventuellement locale d'un objet répliqué ;
- obtenir une plus grande robustesse vis-à-vis des fautes. Les opérations sur l'objet répliqué peuvent rester possibles même si une, voire plusieurs copies, ne sont plus opérationnelles. Les systèmes tolérants aux fautes font ainsi appel à la réplication.

On peut distinguer deux types de réplication : la réplication des traitants (serveurs) et/ou la réplication des données.

Par exemple, les ressources répliquées peuvent être des serveurs assurant une même fonctionnalité : la répartition du traitement des requêtes qui leur sont soumises évite des points de contention et améliore les performances, par exemple, dans le cas de serveurs Web.

Cependant, si ces serveurs accèdent à des données communes, il se pose un problème de contention et de robustesse si ces données ne sont pas répliquées. Il est alors tentant de résoudre ce problème par la réplication des données. Néanmoins, il se pose alors un problème de maintien de la cohérence des copies si celles-ci sont mises-à-jour. Diverses solutions et formes de cohérence ont été définies et implantées. Nous verrons que plusieurs mécanismes sont nécessaires notamment des mécanismes de datation.

1.5.5 Transparence des fautes

De façon plus générale, nous avons vu qu'un système réparti devait obligatoirement considérer les défaillances de certains de ses nœuds comme des événements probables. Des fautes lors des échanges de messages peuvent aussi survenir. Il est donc important d'assurer une bonne tolérance aux fautes de l'ensemble des services d'un système réparti. À titre d'exemple, pour un système de gestion de fichiers, la défaillance d'un volume monté à distance mais momentanément non utilisé, ne doit pas bloquer le fonctionnement global du service.

Un problème fondamental est alors d'assurer que les traitements gardent malgré les fautes possibles un caractère atomique. L'atomicité d'un traitement sera assurée si d'une part, le traitement s'exécute complètement ou pas du tout (sans effet) et d'autre part, aucun état intermédiaire n'est perçu par d'autres traitements parallèles.

À titre d'exemple, l'appel procédural à distance admet plusieurs sémantiques mais, seule, la plus contraignante[11] garantit l'atomicité de l'exécution de la procédure.

Cette propriété d'atomicité, bien connue des bases de données, est associée à la notion de transaction devant assurer un traitement cohérent des données en présence de parallélisme et éventuellement de répartition. Les propriétés d'une transaction sont résumées dans l'acronyme *ACID* qui fait référence à quatre propriétés devant être garanties : *Atomicity, Consistency, Isolation, Durability.*

La répartition a conduit à revisiter ces problèmes de transactions dont l'implantation s'avère complexe et touche à des problèmes clés de la répartition comme celui du consensus.

11. La sémantique dîte « exactement une fois » (exactly once).

Les trois derniers niveaux ... ces trois derniers niveaux concernent des propriétés très utiles pour optimiser la répartition des traitements et des données dans des systèmes répartis à large échelle. Nous sommes donc en plein cœur des problèmes à résoudre d'un point de vue purement technique lorsque l'on considère le *Cloud computing*[12].

1.5.6 Transparence de la migration

Comme toujours en informatique, deux types d'entités peuvent migrer : les traitements et les données. Dans un contexte réparti, on peut donc envisager soit de « déplacer » dynamiquement des données là où elles sont utilisées, soit de déplacer les traitements au plus près des données qu'ils utilisent.

Les deux idées ont été testées et étudiées. Le déplacement des données induit souvent une forme de réplication : c'est une copie qui est fournie à l'utilisateur. Le déplacement d'un traitement, autrement dit, sa répartition dynamique sur plusieurs sites successifs d'exécution a aussi été expérimentée. On parle d'*agents mobiles* et de migration *proactive* : c'est dans le code même exécuté par l'agent (processus) que les instructions de migration vers un autre site sont présentes.

La transparence de la migration consiste, quant à elle, à assurer qu'une ressource pourra migrer d'un nœud à un autre sans que les usagers qui l'utilise s'en aperçoivent. Ce genre de problème a été en particulier étudié sur la ressource processus. En effet, ceci permet de déplacer un service dynamiquement vers un nœud moins chargé et donc de mettre en œuvre des stratégies de régulation de charge sur une architecture répartie. La difficulté tient au fait que le processus déplacé doit retrouver sur la machine cible un environnement identique à celui qu'il a quitté.

L'implantation de ce type de transparence passe par des méthodes de liaison dynamique. Deux approches sont possibles :

- soit le déplacement de la ressource s'accompagne de la mise à jour de toutes les « tables de références » qui contenaient une référence à cette ressource ;
- soit le déplacement provoque une exception chez les usagers lors d'une tentative d'accès. Cette exception doit être traitée par consultation d'un serveur d'annuaire connaissant la nouvelle localisation de la ressource afin de mettre à jour, à la demande, les tables de références.

1.5.7 Transparence de charge

Dans une architecture répartie, la régulation de la charge de chaque nœud peut permettre une meilleure exploitation du système global et une meilleure satisfaction des utilisateurs.

La mise en œuvre est cependant délicate car une régulation de la charge globale implique une bonne connaissance de l'état global du système. Or, cette connaissance est justement difficile à obtenir comme nous l'avons vu. Au mieux peut-on espérer avoir une observation d'un passé récent. C'est pourquoi, des algorithmes répartis doivent être conçus spécifiquement.

La notion de **machine virtuelle** apporte une assistance importante dans ce domaine. Elle permet de créer des nœuds localisés sur une même configuration physique en contrôlant de façon fine l'allocation des ressources de cette configuration entre les différentes instances de machines virtuelles.

Cette notion n'est pas nouvelle. Elle a été mise en œuvre dans les années 1965-70 par IBM et le *Cambridge Scientific Center* pour aboutir au système CP/CMS (CP/CMS : Control Program/Cambridge Monitor System). Pour la première fois, un noyau hyperviseur permettait de faire tourner plusieurs machines virtuelles, elles-mêmes supports d'exécution de plusieurs noyaux de systèmes d'exploitation éventuellement différents.

La puissance et le parallélisme interne des architectures matérielles actuelles ont remis en avant cette notion de machine virtuelle. Ainsi une architecture matérielle peut « héberger » un nombre dynamique de machines virtuelles selon la charge de chacune d'entre elles.

12. Nous ne nous intéressons pas ici au modèle économique associé.

1.5.8 Transparence d'échelle

Nous avons vu qu'une architecture répartie est plus modulaire et adaptable dynamiquement qu'une configuration centralisée. L'ajout de nœuds ne nécessite pas l'arrêt du système. Cependant, le passage d'un système comportant une dizaine de sites à un système d'une centaine de sites n'est pas forcément transparent pour les usagers. C'est ce phénomène de changement d'échelle qu'il faut pouvoir contrôler et rendre le plus transparent possible.

La transparence d'échelle ne peut être acquise que par une adaptation très dynamique et automatique aux besoins des applications. Le *Cloud computing* est confronté à ce problème puisqu'il doit mettre à disposition les ressources nécessaires à ses usagers dans un contexte de répartition à large échelle. Les ressources logicielles et matérielles disponibles dans les grands centres de calcul doivent être réparties au mieux entre les usagers. Cet objectif sera d'autant mieux atteint que des mécanismes performants d'allocation dynamique des ressources seront développés et mis en place.

1.6 Exemple de répartition : Variations sur un thème de location

Pour bien comprendre les problèmes algorithmiques posés par la répartition, nous utiliserons un exemple que nous enrichirons progressivement. Au cours de ces variations, nous nous concentrons sur l'aspect contrôle, synchronisation du système et de ses usagers.

1.6.1 Exposition du problème

Considérons un garage de véhicules de location qui possède les propriétés suivantes :
- la capacité du garage est limitée à N places ;
- il existe deux guichets d'accès au garage : l'un que l'on peut qualifier de guichet de départ pour les prises de véhicules, l'autre que l'on peut qualifier de guichet de retour pour les restitutions. Ces deux guichets sont distants géographiquement pour faciliter la circulation des véhicules (par exemple l'un au nord du garage et l'autre au sud).
- Les véhicules présents dans le garage peuvent être loués et sortir par l'issue correspondante et réciproquement, des véhicules loués peuvent être restitués.
- Les véhicules loués ne sont pas forcement restitués à ce garage et inversement, les véhicules restitués ne sont pas obligatoirement issus de ce garage.
  On notera *Restituer* et *Louer* les deux opérations qui constituent l'interface du garage. Le guichet de retour contrôle l'exécution de l'opération *Restituer*. Celui de départ contrôle l'exécution de l'opération *Louer*.

1.6.2 Le contrôle des mouvements de véhicules

Spécification

Les locations et restitutions des véhicules doivent donc être contrôlées de façon à garantir qu'il n'y a pas trop de véhicules dans le garage ni qu'on loue plus de véhicules qu'il n'en existe. Cette propriété de sûreté se traduit simplement sous la forme d'un invariant en introduisant deux compteurs monotones croissants : l'un comptant les véhicules restitués (R) et l'autre les véhicules loués (D). Leur différence doit être exactement le nombre de places occupés (le nombre de véhicules présents) dans le garage.

$$\texttt{invariant} \quad 0 \leq (R - D) \leq N \tag{1.1}$$

La figure (1.1) illustre les spécifications de ce système de location.

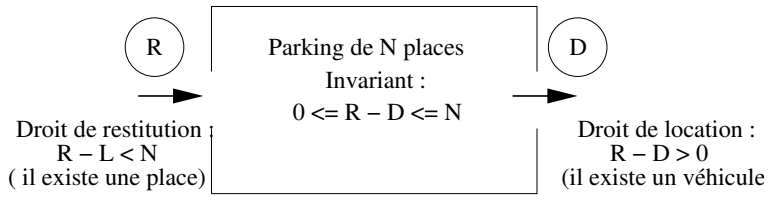


FIGURE 1.1 – Spécification du problème

Remarques

1. Il faut bien noter que les compteurs R et D n'ont pas forcément une valeur nulle initialement. Il faut simplement que leur différence $(R - D)$ reflète bien le nombre de véhicules présents dans le garage conformément à l'invariant.
2. Ce problème constitue un système dit réactif composé d'une partie «environnement» représentée par les véhicules et d'une partie «contrôle» assurant la gestion correcte de la ressource garage.
3. L'invariant (1.1) doit être vérifié par le système global composé des véhicules et des deux issues guichets.

Indépendamment de tout problème de répartition, l'entrée ou le départ d'un véhicule est donc soumis à précondition.

$$\begin{array}{lcl} \text{Pré}(Restituer) & : & (R-D) < N \\ \text{Pré}(Louer) & : & 0 < (R-D) \end{array}$$

Compte tenu des hypothèses, la connaissance du compteur des entrées R est localisée au guichet de restitution. Celle du compteur des départs D est localisée au guichet de départ. Or, on peut constater que le guichet de retour, comme celui de départ, doit avoir une connaissance «globale» des deux compteurs pour pouvoir décider si une opération est acceptable en évaluant la précondition correspondante.

Autrement dit, nous sommes en face d'un problème réparti. Comment chaque guichet peut-il avoir une connaissance suffisante du compteur géré par l'autre pour pouvoir assurer un contrôle correct sachant qu'ils ne peuvent que s'échanger des messages ? Nous allons envisager plusieurs possibilités de répartition du contrôle.

Répartition par accès distant (maintien d'un chef d'orchestre)

Une version simple consiste à «centraliser» les variables d'états R et D de façon à pouvoir évaluer les deux prédicats. Il faut donc introduire un troisième nœud «central» qui implante un service `Contrôle` accessible à distance offrant les deux opérations :

```
interface Contrôle() {
    IncrR() /* incrémenter le compteur R */
    IncrD() /* incrémenter le compteur D */
}
```

Sur le nœud central, il faudra donc démarrer un serveur acceptant l'interface précédente. Ce serveur devra assurer le maintien de l'invariant (1.1) par une synchronisation correcte des deux opérations. Le guichet de restitution exécute une opération locale de restitution en commençant par appeler la procédure `IncrR` à distance. La terminaison de cet appel autorise le guichet à continuer son opération locale de restitution d'un véhicule. Il en est de même pour le guichet de départ avec l'opération `IncrD`.

L'inconvénient de cette solution est qu'elle nécessite un appel procédural à distance pour chaque opération de restitution ou de location. De plus, la disponibilité du service repose sur le bon fonctionnement des trois nœuds.

Répartition par image (deux solistes mais ils s'observent)

Supposons que, par un protocole adéquat à définir, chaque guichet puisse avoir une «image» du compteur distant (source) de l'autre. La sémantique d'un compteur image peut être définie informellement par les propriétés de sûreté et de vivacité suivantes :

- le compteur image a initialement la même valeur que le compteur «source» ;
- le compteur image ne peut prendre que des valeurs prises par le compteur source et ceci dans le même ordre chronologique ;
- le compteur image peut ne pas prendre toutes les valeurs successives du compteur source ;
- si le compteur source prend la valeur k, le compteur image finira par prendre une valeur égale ou supérieure à k.

On notera cette relation dite d'observation entre une source et une image[13] par le symbole $\prec$. Pour le garage, on peut donc supposer que les relations d'observations suivantes vont être mises en œuvre :

$$`R \prec R \ \wedge \ `D \prec D$$

Compte tenu des propriétés énoncées sur les observations, l'image d'un compteur croissant est toujours inférieure ou égale à la valeur de sa source (Petit théorème facile à démontrer). On a donc :

$$\texttt{invariant} \ \ `R \leq R \ \wedge \ `D \leq D \tag{1.2}$$

On a par conséquent le théorème suivant :

$$R \geq D \ \wedge \ `D \leq D \ \vdash \ (R - `D < N \ \Rightarrow \ R - D < N)$$

Autrement dit, le guichet de restitution, s'il possède localement l'image $`D$, peut décider à coup sûr en évaluant le prédicat **local** mais approché $R - `D < N$. Il en est de même pour le guichet de location avec :

$$R \geq D \ \wedge \ `R \leq R \ \vdash \ (`R - D > 0 \ \Rightarrow \ R - D > 0)$$

Une solution au problème consiste donc ici à «localiser» l'évaluation des préconditions réparties grâce à l'utilisation d'images. Il suffira d'implanter le mécanisme d'observation. Ceci peut être fait de multiple façons. Par exemple, si l'on suppose que les deux guichets sont reliés par un canal de communication bidirectionnel respectant l'ordre chronologique des envois de messages, le mécanisme d'observation peut être implanté par :

- l'envoi périodique ou à chaque modification de la valeur de la source vers le nœud de l'image ;
- l'appel procédural à distance, issu de la source, de l'opération $image.incr(1)$ par la source répliquant chaque opération locale $source.incr(1)$;
- l'appel procédural à distance, issu de l'image, de l'opération *source.lire* pour obtenir la valeur de la source, appel qui peut être périodique ou à la demande lorsque le prédicat à évaluer est faux par exemple ;
- etc, etc , ...

Cet exemple montre l'absence d'état global nécessitant ici d'utiliser une connaissance approchée impliquant la condition globale attendue.

Les avantages que l'on peut tirer de la répartition du contrôle dans cette solution sont variés :

- minimisation possible des échanges de messages. Le protocole d'observation peut s'ajuster aux besoins applicatifs ;
- fiabilité du système : il y a un composant en moins ;
- disponibilité : même si un des deux guichets connaît une défaillance momentanée, l'autre peut assurer encore son rôle dans la mesure où la précondition reste valide avec la dernière valeur d'image obtenue.

Sur ce dernier point, le redémarrage du guichet défaillant pose cependant le problème de la restauration d'une valeur correcte du compteur associé. Il faut assurer que la valeur courante des compteurs de chaque guichet est robuste, c'est-à-dire qu'elle survit à un arrêt.

13. Cette notion d'image est similaire à la notion de cache en cohérence faible

Répartition par circulation (une partition nomade)

L'idée de base de cette solution est que l'on peut «simuler» des variables globales en faisant circuler équitablement entre les nœuds un message qui contient la valeur courante de ces variables. On parle aussi de jeton circulant valué. Lorsqu'un nœud possède ce jeton (≡ a reçu le message unique qui circule), il peut considérer qu'il possède le droit d'utilisation ainsi que la valeur courante de la variable globale.

Dans notre problème, il suffit de faire circuler entre les deux guichets un message « jeton» unique valué par la différence des deux compteurs $VP = R - D$. Cette différence représente, ne l'oublions pas, le nombre de véhicules présents dans le garage. Le guichet de restitution incrémente VP et le guichet de départ le décrémente en garantissant l'invariant : `invariant` $0 \leq VP \leq N$.

Les opérations de restitution et de location sont alors conditionnées non seulement par l'invariant précédent, mais aussi par la présence du jeton. L'objet jeton doit passer par le site pour que celui-ci soit accédé et éventuellement modifié. Ceci peut être décrit par un objet de classe Passage ayant comme interface :

```
class Passage {
   int obtenirJeton() ;
   void libérerJeton(int val) ;
   }
```

À titre d'exemple, l'algorithme d'une opération de restitution de véhicule utilisant le jeton valué sera de la forme suivante :

```
int vpl = unPassage.obtenirJeton() ; unPassage.libérerJeton(vpl++) ;
```

À titre d'exercice, se pencher sur l'opération de location. Quel est son algorithme ?

Remarques

- Ne pas oublier qu'il faut assurer la traversée du nœud par le jeton lorsqu'il n'existe aucune requête en attente localement.
- La circulation du jeton se fait en général sur un anneau logique passant par tous les nœuds. Un problème non négligeable de cette approche est donc le bruit de fond provoqué par la circulation permanente du message jeton même si aucune opération n'est en attente sur un nœud.
  Un deuxième problème se pose avec le risque de perte du jeton en cas de défaillance dans les communications (perte du message jeton) ou de défaillance d'un nœud (arrêt du nœud).

1.6.3 Enrichissement du système (Plusieurs orchestres)

Supposons maintenant qu'il existe plusieurs compagnies de location (Hertz,Ada, Europcar,...) et qu'elles aient décidé de partager le garage sans pour autant mettre en commun leurs systèmes d'information. Chaque compagnie possède donc un guichet de retour et un de départ et ne loue que ses propres véhicules. Si l'on suppose qu'il existe P compagnies, on a donc P compteurs D_i et P compteurs R_i répartis dans chaque guichet.

Répartition par observation et circulation

Pour un guichet de départ i, la précondition à évaluer est :

$$\text{Pré}(Louer) \quad : \quad (R_i - D_i) > 0$$

Cette condition indique que la compagnie i possède encore au moins un véhicule à louer dans le garage. Une solution simple pour une évaluation locale au guichet de départ, est d'utiliser des observations. Le guichet de départ doit posséder une image $`R_i$ du compteur R_i. On aura la propriété :

$$(`R_i - D_i > 0) \;\Rightarrow\; (R_i - D_i > 0)$$

Pour un guichet de retour, le problème est un peu plus difficile. Il faut en effet tester la précondition :

$$\text{Pré}(\mathit{Restituer}) \quad : \quad (\sum_{i=1}^{i=P} R_i - \sum_{i=1}^{i=P} D_i) < N$$

Pour les compteurs D_i, chaque guichet de retour peut se contenter d'une approximation. On peut donc placer sur un guichet de retour i_0, les images de tous les autres compteurs $D_j,\ j \neq i_0$ [14].

Pour un guichet i_0, on aura donc besoin des relations d'observation $\langle \wedge\ j = 1, P\ :\ `D_j^{i_0} \prec D_j \rangle$ et, par suite, l'implication suivante sera satisfaite :

$$(\sum_{i=1}^{i=P} R_i - \sum_{j=1}^{j=P} `D_j^{i_0}) < N \quad \Rightarrow \quad (\sum_{i=1}^{i=P} R_i - \sum_{i=1}^{i=P} D_i) < N$$

Reste le problème de la somme des compteurs R_i. Il faut pouvoir disposer de la valeur exacte ou d'une approximation par excès. Une solution possible est celle du jeton. Une variable globale abstraite S_R peut être représentée par un jeton circulant entre les guichets de retour placés sur un anneau logique. Pour préciser la localisation de l'objet (de la variable) jeton mobile, on introduit le prédicat $X@\mathit{node}(i)$. Ce prédicat est vrai si la variable X est localisée sur le site $\mathit{node}(i)$. Le test devient alors possible sous la forme :

$$S_R@\mathit{node}(i_0)\ \wedge\ (S_R - \sum_{j=1}^{j=P} `D_j^{i_0}) < N$$

Répartition par réplication

Une autre solution consiste à répliquer la variable somme S_R sur tous les guichets de retour. Il se pose alors le problème de la cohérence globale de ces copies. Il est nécessaire de respecter un protocole bien précis pour assurer une mise à jour dite «atomique» des différentes copies [15]. À titre d'exemple des problèmes qui peuvent survenir, il suffit de considérer le cas de deux guichets de retour G_{R1} et G_{R2}. Supposons que l'on ait donc deux copies S_{R1} et S_{R2}. On peut envisager le maintien de la cohérence des deux copies en répliquant tout simplement chaque opération logique sur les deux copies. Ainsi, une opération de restitution ayant pour origine G_{R1} devra se traduire par une opération locale $S_{R1}.\mathit{incr}(1)$ et une opération distante $S_{R2}.\mathit{incr}(1)$. Symétriquement pour une opération de restitution ayant pour origine G_{R2}, il faudra exécuter une opération locale $S_{R2}.\mathit{incr}(1)$ et une opération distante $S_{R1}.\mathit{incr}(1)$.Supposons le déroulement chronologique suivant :

Restitution ayant pour origine le guichet	G_{R1}	G_{R2}
Opération locale	$S_{R1}.\mathit{incr}(1)$	
"		$S_{R2}.\mathit{incr}(1)$
Opération distante		$S_{R1}.\mathit{incr}(1)$
"	$S_{R2}.\mathit{incr}(1)$	

Si la valeur des deux copies était cohérente avant ces deux opérations de restitution mais que la précondition $(S_{R_i} - \sum_{j=1}^{j=P} `D_j^{i_0}) < N$ est devenue fausse aussi bien pour R_1 que pour R_2 après les deux incrémentations locales, les deux opérations distantes deviennent impossibles. En fait, les deux opérations de restitution ont été partiellement éxécutées. Pourtant, il était possible d'exécuter complètement une SEULE des deux restitutions. Il ne fallait simplement pas entrelacer les opérations d'incrémentation. Des protocoles de diffusion atomiques ont été proposés pour résoudre ce genre de situation.

14. Pour simplifier l'écriture des formules, on suppose que sur le site i_0, l'image $`D_{i_0}$ est un alias pour désigner la source D_{i_0} puisque image et source sont sur le même site

15. L'atomicité est une propriété qui n'est pas spécifique aux systèmes réparties. Les systèmes transactionnels (centralisés) ont largement étudié ce problème. Cependant, la répartition pose des problèmes de mise en œuvre plus complexes.

1.6.4 Conclusion

Nous avons vu à travers cet exemple, à la fois les difficultés et la richesse des solutions réparties. Encore, n'avons nous pas considéré le système d'information proprement dit. En effet, chaque guichet devra sans doute gérer un journal des opérations qu'il a traité par exemple dans une journée. Ces journaux enregistrés sur chaque guichet seront éventuellement transmis à un site central chaque nuit durant la fermeture du garage. Inversement, un site central(isateur) pourra interroger régulièrement les guichets.

On peut aussi envisager une régulation entre garages s'il en existe plusieurs implantés dans différentes villes. Il faudra alors définir un protocole d'échange permettant d'éviter à la fois les «ruptures de stock» et les engorgements des garages.

Dans tous les cas, certains problèmes récurrents devront être résolus : choix de la répartition des données, synchronisation globale, atomicité des opérations, réplication, diffusion,. . ..

1.6.5 Exercices

1. Un jeton se perd. Imaginer des solutions pour détecter cette perte et engendrer un nouveau jeton (Attention, il doit être unique !).
2. La circulation permanente du jeton consomme de la bande passante réseau inutilement. Imaginer un protocole permettant d'arrêter le jeton sur un site et de le remettre en mouvement lorsqu'une requête apparaît sur un site quelconque.
3. Réfléchir au problème posé par l'arrêt d'un guichet de retour, de départ notamment vis-à-vis des observations.
4. Concevoir une régulation possible entre plusieurs garages de location.
5. Imaginer un protocole qui éviterait le problème des entrelacements erronés d'opérations répliquées.

Première partie

Algorithmique répartie

Chapitre 2

Les principes

2.1 Modélisation de la répartition

Pour mieux comprendre la répartition, il est nécessaire d'avoir une modélisation la plus précise possible d'un système réparti. Un certain nombre d'abstractions ont été proposées. Elles conduisent à des modèles de calcul réparti différents.

On peut distinguer deux grande classes de modèles :

- d'une part, les modèles fondés sur la notion de processus communicants. Ces modèles d'exécution répartie conservent la notion fondamentale de processus des systèmes parallèles.[1] Par contre, divers modes de communication entre processus sont possibles et ils modifient très profondément le modèle final. A titre d'exemple, la communication peut être synchrone ou asynchrone, point à point ou par diffusion, etc.
- d'autre part, les modèles fondés sur la notion d'objet. Cette approche s'est développée avec le succès de la technologie objet dans les langages de programmation. Toutefois, l'utilisation du concept d'objet qui en est faite est très différente. L'accent est mis sur la modularité apportée par le concept, l'aspect réutilisation étant beaucoup plus secondaire.

Dans ce chapitre, nous nous concentrons sur le modèle des processus communiquant par messages. C'est en effet le modèle de base qui a bien évidemment été le plus étudié et exploité. C'est en particulier celui utilisé pour la description des algorithmes répartis. C'est pourquoi nous décrirons plus spécifiquement un modèle standard de processus communicants utilisé dans ce contexte d'étude.

2.1.1 Processus communiquant par messages

Les processus communiquant par messages constituent le modèle à la fois le plus ancien et le plus étudié des modèles de calcul réparti. Un calcul réparti est tout d'abord un calcul parallèle. C'est pourquoi la notion de processus séquentiel a été réutilisée pour exprimer cette fois-ci le grain de répartition. Cependant, l'absence de mémoire globale aux sites d'une architecture distribuée a conduit à introduire des primitives de communication permettant aux processus d'échanger des messages. De multiples protocoles d'échange de messages étant envisageables, ce modèle possède de nombreuses variantes et a été l'objet aussi bien d'études théoriques approfondies que d'implantations diverses.

2.1.2 Formalisation du modèle

D'un point de vue théorique, ce modèle a été formalisé sous forme d'algèbres de processus [1]. Deux approches algébriques ont été particulièrement développées : le modèle CSP (*Communicating Sequential Processes*) de Hoare et le modèle CCS (*Calculus of Communicating Systems*) de Milner. Les termes de

1. Un processus est une unité d'allocation de ressources locales, donc de répartition. Il fournit d'une part des resssources mémoires, fichiers, programme et d'autre part, de la puissance de traitement sous forme de processus légers ou threads.

l'algèbre permettent de décrire le comportement des processus. Les échanges de messages sont des actions élémentaires simultanées synchrones (on parle de rendez-vous) entre deux processus ou plus. Cette approche algébrique modélise un calcul réparti de façon suffisamment formelle pour valider certaines propriétés de sûreté telles que l'interblocage. Cependant, la complexité du raisonnement croît très rapidement avec le nombre des interactions par messages et rend difficile la validation de telles applications. Paradoxalement, bien que le formalisme de description algébrique soit très abstrait, deux obstacles se combinent et réduisent les possibilités offertes par cette approche : d'une part, les concepts modélisés sont eux très élémentaires et d'autre part, la communication est considérée comme synchrone et idéalement atomique, masquant ainsi bien des difficultés rencontrées dans les systèmes réels.

2.1.3 Mises en œuvre du modèle

Au niveau système d'exploitation, l'implantation de ce modèle est évidemment la plus répandue puisqu'indispensable d'un point de vue pratique. Depuis le système ACCENT jusqu'aux protocoles d'Internet, en passant par PVM/MPI, de nombreux systèmes d'exploitation ont intégré des primitives de communication permettant l'échange de messages entre processus distants. De façon pratique, chaque machine est le support d'exécution de plusieurs processus, ce qui amène à distinguer pour fixer l'origine ou la destination d'un message, d'une part, un nom de site global à l'architecture distribuée (cette désignation étant elle-même structurée en domaines, sous-domaines...) et d'autre part, un nom local associé au processus (notion de port dans IP par exemple).

L'impact de la répartition sur les systèmes d'exploitation ne s'est pas réduite à l'introduction de primitives de communication par messages. Des services accessibles à distance sont très vite apparus (service de fichiers par exemple). La notion même de processus a été sensiblement modifiée. À l'origine, l'objet « système » processus recouvrait le suivi et la gestion de l'exécution d'un programme séquentiel fixant ainsi la granularité du parallélisme. Le modèle client-serveur a amené à structurer l'activité d'un processus en plusieurs sous-activités séquentielles dédiées chacune à la gestion des échanges avec un client mais partageant les ressources du même processus. Ainsi, un processus peut comporter un degré quelconque de parallélisme sous forme de processus légers (ou *threads*). On distingue alors le grain de répartition fixé par les processus dit lourds qui gardent leur rôle d'unité d'allocation de ressources et le grain de parallélisme déterminé par les processus légers.

L'intégration de ce modèle de calcul réparti dans les langages a donné lieu à de nombreuses propositions. L'approche la plus élémentaire est d'introduire dans le langage un type prédéfini « canal » doté d'opérations de communication *émettre* et *recevoir*. Ces opérations peuvent être synchrones (rendez-vous du langage Occam) ou asynchrones. Pour faciliter la prise en compte du non-déterminisme inhérent aux échanges de messages, la notion de commande gardée, proposée par Dijkstra, a été à la base de structures de contrôle permettant l'écoute de plusieurs canaux simultanément.

Hormis les langages classiques impératifs, une approche issue des langages fonctionnels a aussi été proposée. Il s'agit en l'occurrence des langages à acteurs. Dans de tels langages, les activités sont représentées sous forme d'acteurs. Un acteur peut échanger des messages avec ses acteurs voisins (accointances). Comme dans le modèle à processus, il peut créer de nouveaux acteurs, mais il peut aussi changer dynamiquement de comportement en interprétant tout message reçu comme une continuation. Ce modèle, de par son origine fonctionnelle, inclut à la fois un fort degré de réflexivité et de dynamisme. Il a été mis en œuvre, par exemple, par la réalisation de machines virtuelles à acteurs offrant un langage intermédiaire. Des implantations réparties ont pu être ainsi développées simplement. On peut donc souligner que de nombreuses idées des langages à acteurs sont aujourd'hui exploitées dans les environnements répartis tels que Java (machine virtuelle, réflexivité, mobilité du code).

2.2 Le modèle standard de l'algorithmique répartie

Le problème de base des systèmes répartis est de trouver des principes et concepts pour maîtriser la communication par messages.

Pour concevoir puis décrire des algorithmes répartis, il est nécessaire de choisir un modèle de calcul réparti. Plusieurs modèles existent selon le niveau d'abstraction des communications que l'on choisit (communication synchrone ou non, par message ou par appel de procédure, point à point ou multipoint, fiable ou non, ...). Un modèle élémentaire assez «standard» est cependant utilisé pour décrire les algorithmes. Nous en donnons une description détaillée.

2.2.1 Structuration d'un programme réparti

Un programme réparti peut être naturellement structuré en un ensemble fixe de processus ou noeuds. La description de l'algorithme consistera donc à fournir le programme exécuté par chaque noeud (processus).

$$CR \;=\; \langle \star\; i = 1, N \;::\; P_i \rangle$$

Définition d'un processus communicant

Un processus communicant est l'unité de répartition. Il satisfait les propriétés suivantes :

Encapsulation Un processus encapsule un ensemble de variables dites locales qu'il est seul à pouvoir utiliser. Les valeurs courantes de ces variables définissent l'état courant du processus.

Comportement Un processus exécute séquentiellement une suite d'instructions. L'exécution d'une instruction peut modifier atomiquement cet état (transition), envoyer un message ou recevoir un message. Le comportement d'un processus peut donc être décrit par la suite des transitions qu'il exécute. Cependant, l'hypothèse d'atomicité permet d'adopter un point de vue événementiel : à chaque instruction peut être associé un événement qui traduit l'occurrence de l'exécution de cette instruction. Par conséquent, il est aussi possible de «tracer» et d'abstraire l'exécution d'un processus sous la forme d'une suite d'événements. En général, ce sont évidemment les événements d'envoi et de réception de message qui sont particulièrement significatifs.

L'hypothèse d'atomicité des instructions peut paraître très restrictive. En fait, pour qu'un usager perçoive des instructions atomiques, il suffit qu'une instruction localement non atomique ne soit pas interrompue par la réception ou l'émission d'un message. Ce type de comportement est en général acceptable dans les applications sans contraintes de temps réel.

Identification Chaque processus possède une identification distincte des autres. On peut distinguer deux niveaux : un nom symbolique (une URL, par exemple bach.enseeiht.fr) ou une adresse IP. C'est une hypothèse communément admise.

Connaissance locale Un processus n'a qu'une connaissance très partielle du calcul auquel il participe. Initialement, une hypothèse couramment admise est qu'il connaît son identification, ses voisins via les canaux qu'il peut utiliser et son état interne fixé par la valeur de ses variables locales. Il ne peut connaître qu'une approximation des états des autres processus.

La communication par messages

Les échanges de messages peuvent se faire via des canaux logiques de communication point à point dont les propriétés peuvent varier : (a)synchrone, canal uni/bidirectionnel, fiable ou non, respectant la chronologie d'envoi en réception (canal FIFO), de capacité limitée ou non,...

Des hypothèses très usuelles (car les plus commodes à utiliser dans les algorithmes) consistent à considérer des canaux asynchrones, bidirectionnels fiables et FIFO de capacité illimitée.

L'ensemble des canaux fixe la topologie de communication.

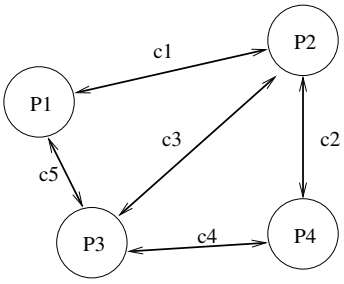


FIGURE 2.1 – Structure d'un calcul réparti

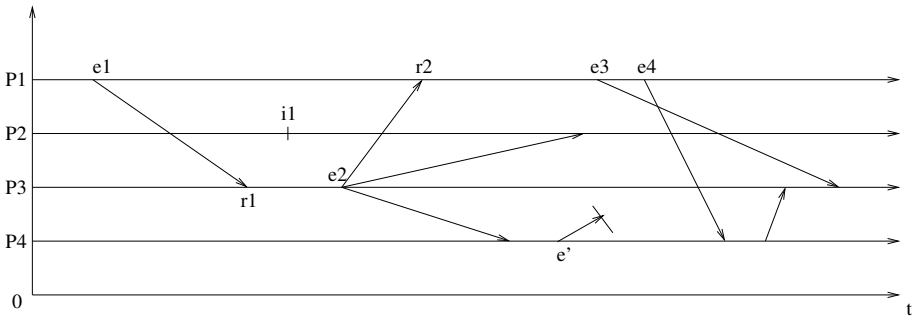


FIGURE 2.2 – Exemple de chronogramme

Remarques Bien remarquer les restrictions du modèle :
- le nombre de processus est constant et la topologie de communication est fixe ;
- Les seules interactions possibles sont les échanges de messages ;
- Aucun processus n'est isolé. Autrement dit, le graphe du réseau de communication est connexe.

Représentation graphique

Cette structure peut être représentée sous forme d'un graphe non orienté dont les sommets sont les processus et les arêtes sont les canaux. La figure (2.1) est un exemple d'un tel graphe comportant 4 processus.

2.2.2 Le chronogramme

Lorsqu'on cherche à analyser le comportement d'un calcul réparti, il est intéressant d'en avoir une image graphique. La notion de chronogramme est un outil élémentaire et indispensable. Il permet d'abstraire graphiquement une exécution répartie en ne considérant que les événements significatifs issus de chaque processus. Trois types d'événements sont distingués : les événements internes au processus, les émissions de message et les réceptions de message.

Le chronogramme de la figure 2.2 montre par exemple un événement interne $i1$, des envois de message point à point (par exemple couple $e1; r1$), une diffusion (émission $e2$), la perte d'un message (émission e'),...

2.2.3 Représentation d'un calcul réparti

Sous les hypothèses précédentes, un calcul réparti peut être abstrait en terme des ensembles d'événements produits par chaque processus au cours d'une exécution particulière. C'est le point de vue événementiel. À tout processus P_i peut donc être associé une suite finie ou non d'événements C_i totalement ordonnés dénotant des événements internes, des envois ou des réceptions de messages issus de P_i ayant eu lieu pour une exécution

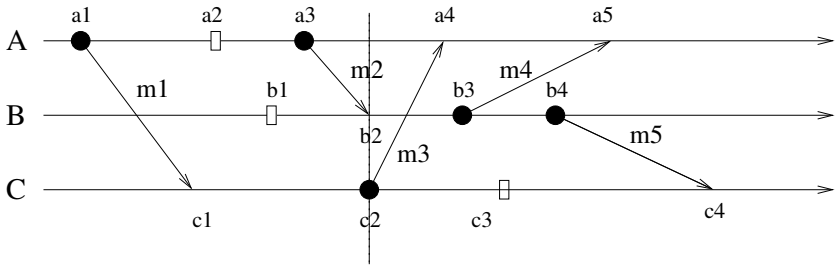


FIGURE 2.3 – Exemple de calcul réparti

donnée. Globalement, un calcul réparti est représenté par une union de toutes ces suites :

$$C = \bigcup_{1}^{N} C_i$$

Les événements issus de processus différents ne sont pas forcément ordonnés. Un ordre partiel les relie cependant, induit par les messages échangés.

Un calcul réparti va donc être caractérisé par une ordre partiel fondé sur une relation dite de causalité. À titre d'exemple, la figure (2.3) montre une suite d'événements ayant lieu sur trois sites différents durant une exécution répartie comportant des échanges de messages entre les sites.

2.2.4 La causalité

La causalité définit une relation d'ordre partiel entre les événements d'un calcul réparti. Cette relation d'ordre partiel, notée $\prec$, est la plus petite relation transitive satisfaisant les deux conditions suivantes :

- Pour tout couple d'événements (e, e') issu d'un même processus, tel que e précède e' dans la suite associée au processus, la relation $e \prec e'$ est vérifiée ;
- Lorsque deux processus échangent un message M, les événements d'envoi e et de réception r sont liés : e précède toujours r dans un temps global et e est la cause de r. En conséquence, pour tout message M, on aura la relation $e \prec r$.

Dotée de cette relation de causalité, une union $C = \bigcup_1^N C_i$ peut représenter un calcul réparti si la relation $\prec$ est acyclique. En effet, tout calcul réel implique que $(C, \prec)$ est un ordre partiel strict.

L'idée générale est que, si $e \prec e'$, alors e' est potentiellement la conséquence de e. Un intérêt de l'ordre causal tient au fait qu'il peut être implanté plus efficacement qu'un ordre total sur les événements et qu'il est néanmoins suffisant pour beaucoup d'applications.

Dans l'exemple de la figure (2.3), certains couples d'événements sont liés causalement, soit directement, par exemple $a_1 \prec c_1$, soit par transitivité de la relation causale, par exemple $a_1 \prec b_3$. Par ailleurs, certains couples d'événements ne sont pas causalement liés, par exemple a_3 et c_1. On note par $||$, l'absence de causalité entre événements. Autrement dit :

$$e \,||\, e' \quad \equiv \quad \neg((e \prec e') \vee (e' \prec e))$$

Attention, cette relation $||$ dénote une relation logique de parallélisme. Elle ne signifie pas que les deux événements se sont produits simultanément dans le temps global réel mais simplement qu'ils auraient pu sans enfreindre la causalité.

Si l'on se place dans un référenciel de temps global, les événements d'un calcul réparti sont ordonnés totalement avec possibilité d'événements simultanés (on notera ce cas par $e \,|||\, e'$ pour le distinguer du cas précédent.).

Si l'on reprend les événements de la figure (2.3), l'ordonnancement réel est par exemple :

$$a_1 \rightarrow c_1 \rightarrow a_2 \rightarrow b_1 \rightarrow a_3 \rightarrow (b_2 ||| c_2) \rightarrow a_4 \rightarrow b_3 \rightarrow c_3 \rightarrow b_4 \rightarrow a_5 \rightarrow c_4$$

Il ne s'agit cependant pas de la seule séquence d'exécution possible compatible avec les contraintes de causalité existant entre les événements de ce calcul. Par exemple, l'ordonnancement suivant est différent de la réalité mais parfaitement compatible avec la causalité :

$$a_1 \rightarrow b_1 \rightarrow c_1 \rightarrow a_2 \rightarrow a_3 \rightarrow (b_2|||c_2) \rightarrow a_4 \rightarrow b_3 \rightarrow a_5 \rightarrow c_3 \rightarrow b_4 \rightarrow c_4$$

Nous verrons que le mécanisme d'horloge logique de Lamport permet d'ordonner totalement les événements d'un calcul réparti en garantissant néanmoins le respect de la causalité. Cependant, comme nous venons de le voir, cet ordonnancement ne sera pas forcément fidèle par rapport à la réalité de l'exécution.

2.2.5 Description des algorithmes

Puisqu'un calcul réparti peut être abstrait sous la forme d'un ensemble d'événements reliés par la relation de causalité, la description des algorithmes répartis utilise souvent une approche de programmation de type événementiel. La description d'un algorithme prend la forme d'une liste de couples (occurrence d'événement, routine de traitement). Les occurrences d'événement portent soit sur la réception d'un message soit sur l'atteinte d'un état par le processus. Chaque processus apparaît comme un système de transitions, le déclenchement d'une transition étant lié à l'état courant du processus et éventuellement à la réception d'un message. Les règles syntaxiques utilisées empruntent aux langages tels que CSP ou Occam très proches de ce modèle de calcul. À titre indicatif, la description de l'algorithme symétrique exécuté par N processus prend la forme suivante :

```
process P(i :0..N-1) {
   type Etat = { <Définition des états possibles> } ;
   Etat EtatCourant= Init ;
   <Déclaration de variables locales>
   <Action d'initialisation>
   while (<Condition>) {
       upon <Condition-1> → {<Action-1>}
   ‖
       ...
   ‖
       upon <Condition-M> → {<Action-M>}
   }
}
```

avec

```
<Condition-i> = <Prédicat> | reception <message>
```

La clause `upon` permet de conditionner une action à un prédicat sur l'état courant du processus et la clause `upon reception` permet d'associer l'exécution d'une action à l'événement de réception d'un message. Dans les actions, l'émission d'un message est traduite sous la forme d'une instruction `send` précisant le type et le contenu du message ainsi que le ou les destinataires. À titre d'exemple, l'envoi d'un message de type $Calc$ et de contenu (x, y, z) vers un processus P_j prendra la forme : `send` $Calc(x, y, z)$ `to` P_j.

Pour certains algorithmes nécessitant du parallélisme interne au processus, plusieurs threads pourront être introduits, chacun étant décrit sous la forme précédente.

2.2.6 Classes d'algorithmes

Il existe un ensemble de classes génériques d'algorithmes. Ceux-ci reposent sur des hypothèses relatives à la topologie de communication et au comportement des nœuds. Après une présentation d'un modèle de calcul très utilisé comme hypothèse, en l'occurrence le calcul diffusant, nous présentons les classes d'algorithme proprement dit.

Calcul diffusant

Le parallélisme des processus et le caractère asynchrone des échanges de messages engendrent une difficulté majeure pour appréhender le déroulement d'un calcul réparti. Il est donc important de pouvoir définir quelques schémas génériques d'exécution capables de modéliser une large classe de programmes répartis. Parmi eux, un modèle de base est celui du calcul diffusant.

Un programme réparti SR composé de N processus communicants obéit au schéma d'exécution d'un calcul diffusant si ses processus respectent les règles suivantes :

- Initialement, un ou plusieurs processus émette(nt) un premier message spontanément ;
- Puis, tous les processus sans exception adoptent le même comportement cyclique générique :

```
while ( Calcul en cours ) {
        Recevoir un message M ;
        Traiter le message M ;
        Diffuser éventuellement un ou plusieurs messages à d'autre processus ;
      }
```

Tout processus passe alternativement d'un état passif (attente de message) à un état actif et inversement.

Ce modèle de comportement est assez général dans la mesure où il exclut simplement l'envoi spontané d'un message alors que le processus est dans l'état passif.

Un problème important dans ce schéma est celui de la terminaison. En effet le test de la boucle n'est pas une condition locale ; il s'agit bien de la terminaison globale du calcul. Une façon de détecter la fin du calcul est de détecter que tous les processus sont passifs (en attente de message) et que plus aucun message n'est en transit.

Les protocoles

La conception des algorithmes répartis utilise souvent le principe de couches hiérarchisées. Ce principe est en particulier mis en œuvre pour les protocoles. Chaque niveau de protocole s'appuie sur les messages du niveau inférieur. Tout niveau de protocole interprète certains champs des messages, champs qui seront ignorés par les niveaux inférieurs.

À titre d'exemple, un protocole de datation des messages d'une application s'appuie sur les messages de cette application pour assurer le service de datation globale des messages. Aucun message spécifique de datation n'est nécessaire. Les informations nécessaires au protocole de datation sont concaténées aux messages applicatifs de manière transparente.

Ce type d'approche présente l'avantage de ne pas engendrer de nouveaux messages mais simplement d'augmenter leur longueur.

Les services

Une deuxième classe d'algorithmes est dédiée à la réalisation de services pour une application. Comme pour les systèmes centralisés, les concepteurs de systèmes répartis ont essayé de dégager un ensemble de services «système» d'intérêt général.

Un service possède une interface définissant un type abstrait de donnée (une classe d'objets).

Les principales classes de services sont :

- les services de synchronisation : exclusion mutuelle, allocation de ressources, élection, sérialisation,...
- les services de fichiers, de ramasse-miettes, de transactions, ...
- les services de nommage ;

Les services peuvent être spéficiés par deux classes de propriétés.

- Une propriété de sûreté exprime le service garanti sous la forme d'un invariant. Par exemple, pour un allocateur de ressource critique, l'invariant précise qu'une ressource ne peut jamais être allouée à plus d'un processus.
- Une propriété de vivacité exprime que toute requête doit finalement être traitée. Par exemple, pour le service d'allocation, toute requête devra être finalement satisfaite. Des contraintes plus fortes d'équité entre les requêtes peuvent aussi être spécifiées.

Les algorithmes de supervision

Les algorithmes de supervision constituent une classe particulière de services relevant souvent des services offerts par un système d'exploitation.

Le contrôle et la supervision d'une application répartie pose un problème d'algorithmique ...répartie.

Par exemple, pour détecter la terminaison d'une application, tous les sites participants doivent être interrogés. Ainsi, la détection de propriétés globales, en particulier stables, nécessite de superviser l'application sans altérer le résultat final de l'exécution. Pour ces algorithmes de supervision il est donc important de distinguer d'une part, les messages de l'application et d'autre part, les messages nécessaires à la collecte des informations de supervision.

Les tâches de supervision consiste en particulier à détecter des propriétés stables. Une propriété est qualifiée de stable si, dès qu'elle devient vraie, elle le reste définitivement dans tout état ultérieur.

Les algorithmes de détection de propriétés stables constituent une classe importante des algorithmes de supervision. Si l'on note par $P.Algo$ le prédicat qui est vrai lorsque l'algorithme $Algo$ a détecté P et $P.SR$ le prédicat qui est vrai lorsque le système supervisé SR vérifie P, les deux propriétés suivantes doivent être assurées :

- une propriété de sûreté assurant l'absence de fausse détection représentée par un invariant :

$$\texttt{invariant} \quad P.Algo \Rightarrow P.SR$$

- une propriété de vivacité assurant que si l'état stable est atteint, il sera finalement détecté par l'algorithme :

$$P.SR \quad \mapsto \quad P.Algo$$

Symétrie

Lorsque les N processus P_i du calcul exécutent le même code, l'algorithme est dit symétrique. Seules les valeurs initiales des variables (répliquées) de chaque processus peuvent être différentes. Le comportement des processus peut donc être différent selon leurs états locaux initiaux.

2.2.7 Flots de contrôle réparti (ou comment mettre de l'ordre dans les messages)

Si l'on n'impose aucune règle dans l'échange des messages, un calcul réparti a de forte chance d'apparaître chaotique et impossible à mettre au point. Un ensemble de règles ou de protocoles particuliers ont donc été proposés pour maîtriser la complexité en structurant le flot global des messages.

La démarche consiste à assujettir les échanges de messages à certaines règles qui permettent ainsi de «structurer» l'évolution du contrôle. En effet, les «flux» de messages de nœud en nœud jouent un rôle déterminant dans l'enchaînement des traitements. Puisqu'il s'agit d'échanges de messages, un premier type de règle possible concerne la topologie logique de communication à laquelle on se conformera, bien sûr compatible avec les possibilités offertes par l'architecture réelle. Par exemple, une architecture peut offrir un maillage complet des nœuds. Cependant, pour implanter un service d'exclusion mutuelle, on se limitera à faire circuler un message jeton sur un anneau logique passant par tous les sites.

Un deuxième type de règle porte sur le découpage de l'algorithme en phases successives ou vagues. La notion de vague permet par exemple d'étendre la notion de boucle au contexte réparti.

Enfin, la notion de synchronisme virtuel permet d'assurer qu'un groupe de processus reçoit les mêmes messages en respectant les règles de causalité qui les relient. Nous envisageons successivement : l'anneau, l'arbre de recouvrement pour les aspects de topologie logique, la vague pour la notion d'itération répartie, les protocoles ordonnés et le synchronisme virtuel.

Anneau

L'utilisation d'une topologie d'anneau restreint fortement le flux de messages d'un calcul réparti. Cette structure est cependant très utile et utilisée. Nous avons vu un exemple avec la notion de jeton qui permet de

trouver un solution simple (sur un réseau fiable) au problème de l'exclusion mutuelle ou de l'élection. Cette topologie permet aussi une mise en œuvre simple de la diffusion d'un message à tous les sites connectés à l'anneau.

Arbre de recouvrement

La topologie d'arbre d'un arbre de recouvrement (c'est-à-dire incluant tous les nœuds) permet, comme l'anneau, de propager un message vers tous les autres sites. Cette structure est bien adaptée à un calcul réparti comportant une ou plusieurs étapes de type (enquête, collecte) lancées par différents nœuds quand ils le veulent. Il faut cependant que l'on dispose donc d'un arbre de recouvrement spécifique à chaque nœud de départ, nœud qui doit alors être racine de l'arbre. En conséquence un problème classique est de résoudre la construction même de l'arbre en mettant en œuvre un algorithme réparti.

Hypothèses On considère un ensemble de nœuds connectés par des canaux bidirectionnels fiables. La topologie du réseau est connexe. Chaque site connait son identité et les canaux avec ses voisins immédiats.

Résultat Si un nœud i décide de lancer l'algorithme, celui-ci devra conduire à la construction d'un arbre de recouvrement ayant pour racine i. Une fois l'algorithme terminé, chaque nœud doit être placé dans l'arbre c'est-à-dire connaître son père et la liste de ses fils ($\equiv$ la liste des canaux conduisant à des fils).

Nous présentons la définition d'une classe Java implantant un nœud. Si l'on admet que la méthode *parcourir* peut être appelée à distance par un protocole de type RMI (Remote Method Interface), une construction séquentielle de l'arbre peut être obtenue par l'appel de *Racine.parcourir*(*null*). On suppose que le voisinage de chaque noeud est initialisé après la création de tous les noeuds (liste publique voisins).

```
import java.util.*;
public class Noeud  {
    private Noeud pere = null ;                // père du noeud
    public  List     voisins ;                 // voisins du noeud
    private List fils = new ArrayList();       // fils du noeud
    // Place le noeud this et poursuit ou stoppe le parcours si this est en place
    public Noeud parcourir ( Noeud appelant ) {
         if (pere != null) return null ;       // Déjà placé
         pere = appelant;                      // Placement
         Iterator lnd = voisins.iterator();    // Recherche des descendants
         while (lnd.hasNext()) {
               Noeud unNd = (Noeud) lnd.next();
               // Appel de type RPC
               if (unNd.parcourir(this) == unN) { fils.add(unNd);}
          }
         return this ;
     }
}
```

Exercice

1. On voudrait pouvoir construire des arbres de recouvrement issus des différents nœuds. Il faut alors identifier les arbres. Modifier la classe Nœud pour offrir cette possibilité.
2. Il est possible de parcourir tous les voisins d'un nœud en parallèle. Modifier la classe pour obtenir ce parcours parallèle.

Itération répartie (notion de vague)

De nombreux algorithmes obéissent à un schéma générique de type (enquête, collecte) auprès des autres sites participants. Par exemple, pour savoir si une application de type calcul diffusant est terminée, un processus superviseur peut déclencher une enquête pour connaître l'état des autres. Il sera amené à répéter cette enquête ultérieurement si la collecte des résultats fait apparaître un ou plusieurs sites actifs et/ou des messages encore en transit.

Ce type de comportement, que l'on peut qualifier d'itération répartie, peut être implanté grâce au mécanisme de vague [22]. Une vague est caractérisée par les opérations suivantes :

```
interface Vague  {
    public void Enqueter (Object Quoi ) ;
    /* lance une nouvelle vague */
    public void Visiter (Object Quoi, Collecte) ;
    /* passage d'une vague sur un site */
    public void Faire_Suivre(Object Quoi, Collecte) ;
    /* propagation de la vague par un site */
    public Collecte Collecter() ;
    /* collecte des résultats et fin de la vague */
 }
```

Propriétés

Un site origine peut lancer une nouvelle vague par l'opération «Enqueter» en précisant la nature de l'enquête. Chaque nœud cible de la vague doit accepter l'opération de visite. Celle-ci permet de collecter des informations. Après quoi, le nœud fera suivre la vague vers d'autres sites. Sur le site origine, l'opération «Collecter» permet de collecter l'ensemble des résultats et la terminaison de cette opération coincidera avec la fin de la vague.

Remarque Lorsque les nœuds participent à l'exécution d'une vague, ils respectent les règles d'un calcul diffusant. Une vague est donc un exemple de calcul diffusant.

Règles de causalité

Si l'on considère les opérations d'une vague comme des opérations atomiques, il est possible d'associer un événement à toute occurrence d'opération. On notera op^i_s, l'exécution de op sur le site s pour la i-ième vague.

Une itération répartie issue d'un site s_0 doit alors obéir aux règles de causalité suivantes :

Chronologie des événements dans une vague

$$\forall\, s \;:\; Enqueter^i_{s_0} \;\prec\; Visiter^i_s \;\prec\; Faire\_Suivre^i_s \;\prec\; Collecter^i_{s_0}$$

Séquentialité des vagues

$$\forall\, i \;:\; Collecter^i_{s_0} \;\prec\; Enqueter^{i+1}_{s_0}$$

Réalisation

Une vague peut être implantée de plusieurs manières. C'est en cela même qu'elle constitue une abstraction en masquant la topologie exacte de communication utilisée pour mener à bien la collecte.

À titre d'exemple, une vague peut-être implantée à l'aide d'un jeton circulant. Le tableau de la figure (2.4) précise l'implantation de chaque opération d'une vague à l'aide d'un jeton.

Site	Vague	Jeton
Site origine	Enqueter	Emettre(Quoi)
Site visité	Visiter	Recevoir(Quoi,Collecte)
Site visité	Faire_Suivre	Emettre(Quoi,Collecte)
Site origine	Collecter	Recevoir(Collecte)

FIGURE 2.4 – Implantation d'une vague à l'aide d'un jeton

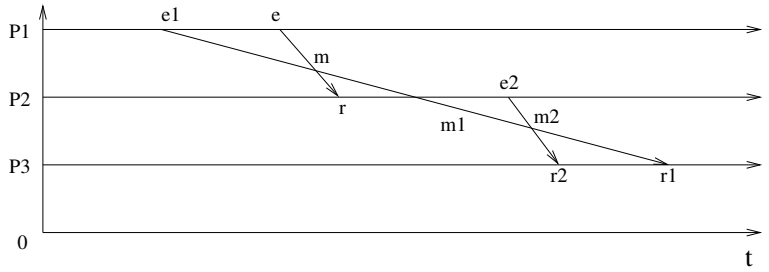


FIGURE 2.5 – Liaison causale en émission

Exercices

1. Vérifier que l'implantation d'une vague à l'aide d'un jeton respecte les règles de causalité des événements associés à une vague ;
2. Quelle possibilité d'optimisation des performances n'est pas utilisée dans l'implantation d'une vague à l'aide d'un jeton ?
3. Reprendre le tableau de la figure (2.4) en supposant que la vague est propagée via un arbre de recouvrement ayant pour racine le nœud origine de la vague.

Les protocoles ordonnés

Nous avons vu qu'un nœud possède une perception approximative de son environnement. En particulier, si deux messages lui parviennent de deux sites distincts, il peut très bien ignorer quel message a été émis le premier (dans un référentiel de temps global). Or, il peut exister une causalité entre les événements d'émission des deux messages. Il suffit de considérer le chronogramme (2.5) :

Le message m_1 est reçu après m_2 sur le site $P3$ alors qu'un lien causal existe entre les deux messages en émission ($e_1 \prec (e \prec r \prec) e_2$). Autrement dit, le message $m2$ reçu par $P3$ peut être incompréhensible pour celui-ci si d'une part, $m2$ est une conséquence de $m1$ (via m) et d'autre part, $P3$ n'a pas vu $m1$.

Il est donc intéressant et important de concevoir des protocoles qui évitent ce genre d'incohérence potentielle. Pour de tels protocoles, on est amené à distinguer d'une part l'événement de réception du message par le nœud et d'autre part, la délivrance effective du message à l'applicatif. On doit donc associer un événement de délivrance d_m à tout message m.

Un protocole ordonné (d'ordre causal) est un protocole qui assure la propriété suivante pour tout site de destination S :

$$\forall\, m, m'\ vers\ S :\ e_m \prec e_{m'} \Rightarrow d_m \prec d_{m'}$$

Diverses implantations de tels protocoles sont possibles. On peut remarquer que le contrôle pour la délivrance d'un message est toujours local au site récepteur : on a toujours à comparer des événements d'émission de messages reçus sur le **même** site.

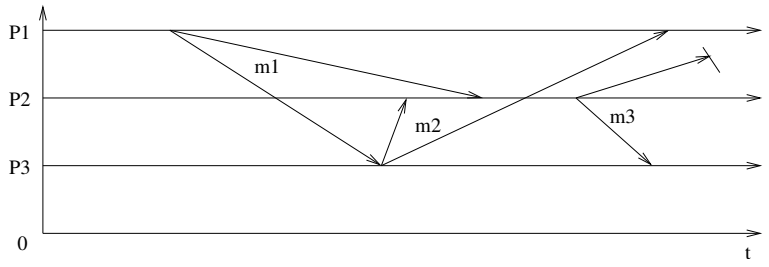


FIGURE 2.6 – Diffusion simple

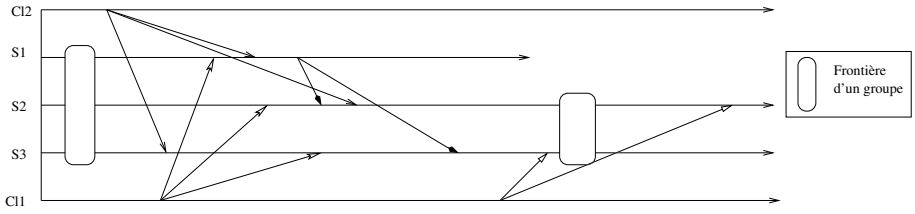


FIGURE 2.7 – Diffusion anarchique

Le synchronisme virtuel

L'un des intérêts importants de la répartition est de pouvoir répliquer des traitements ou des données de façon à obtenir des services plus fiables et disponibles. Cependant, la réplication implique l'utilisation de protocoles non plus point à point mais de diffusion.

Or, la complexité des échanges de messages est plus grande lorsque l'on utilise des protocoles de diffusion. Deux problèmes majeurs apparaissent :

- Séquentialité : Les messages successifs émis par un nœud seront-ils reçus dans le même ordre par tous les nœuds cibles de la diffusion ;
- Atomicité : Tous les sites cibles recevront-ils chaque message diffusé ?

Le chronogramme de la figure (2.6) montre l'imbroglio de messages pouvant résulter d'un protocole de diffusion à faible coût.

Or, la diffusion vers des groupes de processus (multicast) est utile pour réaliser des applications robustes et à haute disponibilité. En effet, une méthode classique pour obtenir ces propriétés de robustesse et de disponibilité est d'utiliser la redondance des ressources offertes par une architecture répartie.

Dans un schéma classique client-serveur, le service peut être rendu plus robuste en activant deux serveurs «jumeaux». L'arrêt de l'un n'entraînera pas l'arrêt du service.

Cependant, il faut alors diffuser les requêtes vers les deux serveurs en assurant que celles-ci arriveront dans le même ordre et que toute requête sera prise en compte par les deux serveurs ou par aucun.

Le modèle d'exécution qualifié de synchronisme virtuel a pour objectif de fournir des protocoles capables de respecter de telles propriétés. Les systèmes ISIS et HORUS [4] sont des exemples d'environnements d'exécution répartie offrant de tels protocoles.

Un calcul réparti virtuellement synchrone assure que les processus appartenant à un même groupe peuvent être cibles de diffusions totalement ordonnées atomiques.

Les figures (2.7) et (2.8) résument les différences essentielles entre un protocole de diffusion de base et un protocole assurant un modèle d'exécution virtuellement synchrone.

La réalisation d'un noyau d'exécution support du synchronisme virtuel nécessite de fournir deux types de primitives :

- les primitives de gestion des groupes : on distingue l'opération d'entrée dans un groupe (de sortie d'un groupe) et l'opération de connexion en tant que client à un groupe (de déconnexion) ;
- les primitives de diffusion : protocole de diffusion ordonnée assurant le respect de la causalité, protocole

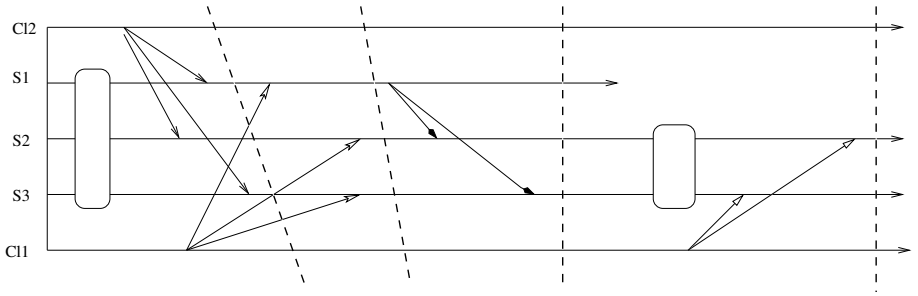


FIGURE 2.8 – Diffusion virtuellement synchrone

de diffusion atomique assurant un ordre total et le caractère atomique de la diffusion.

2.2.8 Évaluation de la complexité

La complexité des algorithmes répartis se mesure essentiellement en terme des paramètres liés à la communication. Des paramètres clés sont la longueur des messages et leur nombre. À un niveau d'abstraction un peu plus élevé, on peut souvent distinguer des phases séquentielles de calcul (vagues successives par exemple). Il s'agit alors de minimiser le nombre de phases successives. Enfin, des propriétés particulières peuvent être prises en compte telles que : la tolérance aux fautes de l'algorithme, son caractère auto-stabilisant, sa symétrie, etc ...

2.2.9 Un exemple d'algorithme : le problème de l'élection

Hypothèses

On considère N processus $< P_i >_{0 \leq i < N}$ communiquant via un réseau en anneau unidirectionnel. Chaque processus connaît donc un prédécesseur $P_i.pred$ dont il peut recevoir des messages et un successeur $P_i.suc$ auquel il peut envoyer des messages. Chaque processus a une identité distincte des autres mais ne connait pas l'identité des autres.

Il existe un ordre total entre les identités de chaque processus (par exemple l'ordre lexical si les processus sont identifiés par un nom symbolique).

Le problème de l'élection consiste à ce qu'un processus unique se reconnaisse comme élu au terme de l'exécution d'un algoritme d'élection identique pour tous les processus.

Compte tenu des hypothèses faites, une solution simple consiste à propager, via des messages sur l'anneau, les identités de chaque processus afin qu'un processus P_{i_0} finissent par avoir la connaissance globale suivante :

$$P_{i_0}.Elu \Rightarrow \forall i \ : \ 0 \leq i < N \wedge i \neq i_0 \ :: \ P_{i_0}.nom < P_i.nom$$

Autrement dit, le processus i_0 sait qu'il possède la plus petite identité et sera déclaré élu.

Une spécification du problème

De façon habituelle, on peut abstraire les processus participants en considérant seulement deux états et une transition « clé ». Initialement, tout processus est dans l'état non élu. L'objectif est de parvenir à un état où un seul processus finit par être dans l'état élu. Le graphe (2.2.9) illustre cette abstraction de chaque processus en terme d'un système de transitions.

L'algorithme doit assurer l'unicité de l'élu en garantissant l'invariant :

$$\texttt{invariant} \ \ \forall i, j \ : \ 0 \leq i, j < N \ :: \ P_i.Elu \wedge P_j.Elu \Rightarrow i = j$$

FIGURE 2.9 – Abstraction des états d'un processus participant

Seule la transition de l'état non élu à élu est autorisée :

$$\texttt{stable} \quad P_i.Elu$$

Par ailleurs, il doit garantir que finalement un processus sera élu :

$$true \mapsto \exists i_0 \ : \ P_{i_0}.Elu$$

Pour décider de l'élection, nous avons vu qu'il fallait évaluer, pour chaque processus P_i, le prédicat global

$$\forall j \ : \ 0 \leq j < N \wedge j \neq i \ :: \ P_i.nom < P_j.nom$$

Les échanges de messages selon les règles imposées par une topologie du réseau en anneau permettent de développer une solution simple pour évaluer ce prédicat global.

Chaque processus commence par envoyer son identité à son successeur. Puis il entre dans une boucle consistant à accepter les noms envoyés par son prédécesseur. Pour chaque nom reçu, il adopte le comportement suivant :

- soit le nom reçu est plus grand que celui du processus récepteur : celui-ci ne retransmet pas le nom reçu à son successeur ;
- soit le nom reçu est plus petit que celui du récepteur : celui-ci propage ce nom à son successeur ;
- soit le nom reçu est celui du processus récepteur : alors celui-ci peut se déclarer élu. En effet, si le nom qu'il a émis est revenu, c'est qu'il est passé par tous les processus et donc que le prédicat $P_{i_0}.nom < P_i.nom$ a été trouvé vrai par chaque processus P_i, $i \neq i_0$.

La réception d'un nouveau nom permet à un processus i d'évaluer un nouveau terme $P_i.nom < P_j.nom$. Le fait qu'un nom revienne à son émetteur signifie qu'il a été comparé à tous les autres et trouvé plus petit que tous les autres. La décision d'élire le processus récepteur peut donc être prise.

L'algorithme réalise donc simplement une évaluation répartie du prédicat global.

Une description de l'algorithme peut se faire sous forme d'une classe. Le déploiement pour exécution consistera, pour chaque processus de l'anneau, à instancier un objet de cette classe, l'anneau de communication étant supposé créé :

```
public class Election {
   private String nom = java.net.InetAddress.getLocalhost().getHostName();
   private String unCandidat = null;
   private boolean elu = false ;
   public void Election() {
     Anneau.envoyer(nom) ;
     while (!elu) {
        Anneau.recevoir(unCandidat) ;
        /* propagation éventuelle */
        if (unCandidat < nom) Anneau.envoyer(unCandidat) ;
        /* test d'élection */
        elu = (unCandidat == nom) ;
     }
}
```

Remarques

- Une telle description suppose que le media de communication « Anneau » est capable de mémoriser des messages en transit. En effet, tous les processus commencent par émettre. Il faut donc que ces émissions ne soient pas toutes bloquantes. La communication devra être de type asynchrone ;
- Tous les processus tentent leur chance en commençant par émettre leur nom vers leur successeur. En fait, il suffirait qu'un seul en prenne l'initiative et que chaque processus modifie son comportement lors de l'arrivée d'un nom. Si le nom reçu ne doit pas être propagé (il est plus grand que le nom du récepteur), alors le récepteur décide de tenter sa chance en envoyant cette fois-ci son nom ;
- Le comportement des processus présente les propriétés d'un calcul diffusant.

Exercices

1. Dans l'exemple présenté, seul l'élu se termine. Compléter l'algorithme de façon à ce que tous les participants connaissent le nom de l'élu et se terminent ;
2. Pour ce même algorithme, il suffirait qu'un seul processus prenne l'initiative d'envoyer son nom à son successeur pour déclencher le calcul. Quelle modification doit-on faire dans le comportement de chaque processus pour obtenir finalement la terminaison de l'élection ; alors le récepteur décide de tenter sa chance en envoyant cette fois-ci son nom ;
3. Vérifer qu'un tel calcul réparti satisfait les propriétés d'un calcul diffusant ;
4. Évaluer la complexité de l'algorithme en nombre moyen de messages échangés. En déduire s'il est moins complexe qu'un algorithme qui consisterait, pour tous les processus, à diffuser leur nom à tous les autres.

2.3 Au-delà du modèle standard (ou comment oublier les messages)

La description de la communication en terme de messages est d'un niveau d'abstraction peu élevé. C'est pourquoi, les concepteurs de systèmes d'exploitation ont proposé des mécanismes de communication plus élaborés. Un objectif ambitieux de ces mécanismes est souvent de masquer le phénomène de communication (de répartition).

En ce qui concerne les traitements, un concept bien accepté est celui de procédure accessible à distance. Elle est la base du modèle client/serveur qui structure la plupart des applications réparties actuelles. La mise en œuvre d'un mécanisme d'appel procédural à distance (RPC) est cependant délicate. Plusieurs variantes d'implantation peuvent être envisagées avec des sémantiques distinctes.

En ce qui concerne les données, on peut distinguer d'une part la communication par mémoire partagée et d'autre part, la communication par fichiers partagés.

La notion de mémoire partagée répartie a pour objectif de fournir au programmeur d'applications un modèle de programmation «centralisé». Il peut disposer d'une mémoire globale pour communiquer entre processus distants. La difficulté majeure pour réaliser une telle abstraction sur une architecture répartie est d'éviter une trop forte synchronisation des accès à cette mémoire partagée répartie. L'utilisation de la réplication permet d'augmenter le parallélisme d'accès à la mémoire mais des contraintes de cohérence des copies doivent être garanties pour garder une sémantique de mémoire globale.

Enfin, un service de fichiers répartis a été un des premiers services à être implanté. L'accès à des fichiers distants en assurant la transparence d'accès (même interface que pour un fichier local) et de localisation (désignation logique indépendante du site) constituent les deux propriétés de base garanties par les systèmes de gestion de fichiers répartis. De nombreux systèmes ont été développés. Un standard de fait est le système NFS (Network File System) de SUN. Une difficulté majeure est de garantir une sémantique équivalente à celle d'un système de fichiers centralisés. En effet, le partage de fichiers pose des problèmes voisins du partage mémoire précédent. Des problèmes d'ordonnancement des accès en lecture-écriture par plusieurs clients doivent être contrôlés pour garantir que la version lue par un client est la dernière version écrite. En la matière, les concepteurs ont dû trouver un compromis entre une sémantique «centralisée» garantie à coup sûr et des performances acceptables.

Chapitre 3

Temps et causalité

3.1 Datation

Un problème de base des systèmes répartis est la datation des événements significatifs (émission et réception des messages en particulier) durant l'activité du système. Ceci permet de reconstituer l'exécution à des fins de mise au point par exemple ou de contraindre la prise en compte de requêtes dans un certain ordre.

En général, un mécanisme de datation doit respecter une règle fondamentale : être compatible avec la relation de causalité qui peut exister entre toute paire d'événements. Autrement dit,

$$\forall \ e, e' \ : \ e \prec e' \quad \Rightarrow \quad date(e) < date(e')$$

On peut distinguer deux approches :

- une approche "temps réel" consistant à dater les événements avec une horloge la plus précise possible. La difficulté est alors de disposer de cette horloge globale ;
- une approche "temps logique" consistant à dater les événements en respectant la causalité selon la règle énoncée.

3.1.1 La datation Temps réel

Cette approche pose le problème de la disponibilité d'une horloge globale. En effet, comme nous l'avons souligné, un système réparti ne dispose justement pas d'un référentiel global de temps. Chaque nœud possède une horloge locale plus ou moins précise et surtout non synchronisée a priori avec les horloges des autres nœuds. Une datation directe à l'aide de telles horloges ne convient donc pas car des anomalies causales peuvent être engendrées. Il suffit que l'horloge du nœud émetteur soit en avance sur celle du nœud récepteur pour obtenir un message dont la date de réception est antérieure à la date d'émission.

Pour résoudre ce problème, des algorithmes de synchronisation d'horloges ont été conçus et implantés. Ils permettent de recaler les horloges des nœuds de façon à ce que leur différence reste dans un intervalle borné connu. L'algorithme doit maintenir un invariant du type :

$$\texttt{invariant} \quad Max(h_i \ : \ i = 1, N) - Min(h_i \ : \ i = 1, N) < \epsilon$$

On obtient ainsi une précision ϵ qui garantira une datation correcte si tous les événements causalement liés sont séparés par un délai supérieur à la précision de l'horloge globale ainsi implantée.

La datation temps réel nécessite donc un protocole de synchronisation d'horloge complexe et relativement couteux. La disponibilité d'un émetteur unique (diffusion de tops par une horloge atomique par exemple) peut apporter une simplification dans la mise en œuvre et plus de précision. Cependant, la solution est alors centralisée par nature et donc moins tolérante aux défaillances : défaillance de l'émetteur, mais aussi défaillance locale des récepteurs.

Enfin, pour de nombreuses applications, seul le respect de la causalité est important. Il est même parfois souhaitable de savoir si deux événements sont causalement liés ou non. Une datation temps réelle ordonne totalement tous les événements et ne permet donc pas de distinguer ceux qui sont indépendants (sans causalité) malgré leur précédence temporelle.

Face à ces inconvénients, des solutions fondées sur un temps logique ont été étudiées.

3.1.2 Datation Temps logique

Deux solutions ont été découvertes :

- la première, due à L. Lamport, permet de dater les événements selon un ordre total. L'inconvénient éventuel de cette approche est donc de même nature que celui d'une datation réelle : l'introduction d'un ordre arbitraire entre des événements indépendants.
- la seconde, due à F. Mattern, permet de dater les événements selon un ordre partiel isomorphe à la relation de causalité. Ce mécanisme est plus précis mais plus coûteux à implanter et permet de distinguer (détecter) les événements indépendants. Toutefois, un tel mécanisme ne permet pas de décider de l'existence d'un événement entre deux événements causalement liés.

Horloges de Lamport

Une date est un couple (s, cpt), où s est un numéro de site et cpt est un entier, la numérotation des sites étant supposée totalement ordonnée. L'entier permet de comparer deux dates et en cas d'égalité, le numéro de site permet de distinguer les deux dates. Si deux dates, ayant le même champ site, n'ont jamais la même valeur entière, toutes les dates sont différentes et comparables.

Une horloge locale à chaque site permet de dater les évéments ayant lieu sur le site correspondant. Cette horloge mémorise une date courante (s, cpt) où s est donc le site local de l'horloge. Pour que les dates obtenues à partir de ces horloges soient toutes différentes mais comparables, l'algorithme de gestion d'horloge doit assurer l'invariant :

$$\texttt{invariant}\ \ \forall d, d' \ :\ d.s = d'.s \ \ \Rightarrow \ \ (d.cpt \neq d'.cpt)$$

Pour cela, il suffit que toute consultation de l'horloge pour dater un événement, entraîne l'incrémentation de l'entier compteur.

Les classes *Date* et *Horloge* suivante donnent une implantation possible du mécanisme de datation.

```
class Date implements Cloneable { // définition et comparaison de date

   protected int s ; protected int cpt ;

   static boolean Prec ( Date d1, Date d2 )  {
      return (d1.cpt < d2.cpt) ||( (d1.cpt == d2.cpt) && (d1.s < d2.s)) ;
   }
}

class Horloge extends Date {
   // Création de l'horloge
   Horloge( int où )  { s = où ; cpt = 0 ; }

   // Lecture-Incrémentation de l'horloge
   Date Top()
     { Date dc = (Date) super.clone(); this.cpt++; return dc ; }

   // Recalage de l'horloge
   void Recaler( Date d ) { if (Prec(this,d))  this.cpt = d.cpt + 1 ; }
   // soit encore this.cpt = Max(this.cpt,d.cpt) + 1
}
```

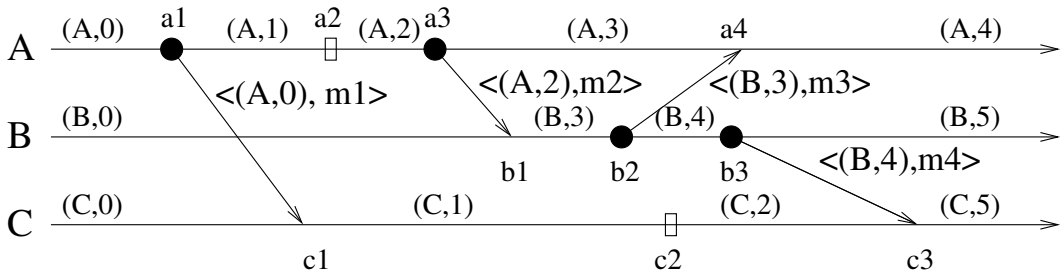


FIGURE 3.1 – Horloges de Lamport

Les actions suivantes seront exécutées sur occurrence de chaque type d'événement :

Type d'événement sur un site s	Action mettant en jeu l'horloge du site s
Événement interne sur s	$H_s.Top()$
Émission sur s de m	$dm = H_s.Top()$; envoi de $< dm, m >$
Réception sur s de $< dm, m >$	$H_s.Recaler(dm)$

Le chronogramme de la figure (3.1) montre l'évolution des horloges de chaque site et la surcharge des messages par la date d'émission de chaque message. On remarquera que les événements de réception ne sont pas datés. Ils ne sont l'occasion que d'un recalage de l'horloge du site de réception mais n'entraîne pas d'opération $Top()$.

Les horloges de Lamport permettent un ordonnancement total des événements d'un calcul réparti en respectant la causalité qui peut exister entre ces événements. Néanmoins, l'ordre introduit entre des événements causalement indépendants ($\equiv$ logiquement simultanés) est arbitraire. À titre d'exemple, la figure (3.1) montre que l'événement a_3 a pour date $(A, 2)$ et l'événement c_2 a pour date $(C, 1)$: on a donc c_2 qui précède a_3 avec cette datation alors que dans le temps absolu c'est l'inverse qui s'est produit. Ceci n'est pas erroné puisque ces deux événements ne sont pas causalement liés.

Horloges de Mattern

Nous venons de voir que le mécanisme de datation par les horloges de Lamport respecte l'ordre causal mais ne permet pas d'avoir l'implication réciproque : $\forall e, e' \ : \ d_e < d_{e'} \ \Rightarrow \ e \prec e'$. C'est ce que les horloges de Mattern vont assurer.

Pour ce faire, il faut passer à une datation vectorielle. Une date (globale) est un vecteur D de dimension égale au nombre de sites. La composante $D[i]$ d'un tel vecteur indique le nombre d'événements ayant eu lieu sur le site i et qui précède causalement cette date. La relation d'ordre entre 2 dates devient alors :

$$\forall D, D' \ : \ D \leq D' \ \equiv \ \forall i \ : \ D[i] \leq D'[i]$$

et

$$\forall D, D' \ : \ D < D' \ \equiv \ D \leq D' \wedge \exists k \ : \ D[k] < D'[k]$$

On peut alors constater que deux dates peuvent évidemment être non comparables puisqu'il peut exister des dates telles que :

$$D \ || \ D' \ \equiv \ \neg(D < D') \wedge \neg(D' < D)$$

En fait, la relation d'ordre définie peut correctement traduire la relation de causalité par un mécanisme adéquat de gestion des horloges correspondantes.

Chaque site s gère une horloge vectorielle H_s. La datation d'un événement se produisant sur un site s entraînera l'incrémentation de la composante correspondante de l'horloge locale au site. Deux événements distincts quelconques n'auront donc jamais la même date et par conséquent seule la relation $<$ nous intéresse.

Comme précédemment, chaque message sera surchargé par la date de l'événement d'émission.

La classe décrivant une date vectorielle devient :

```
class DateV { // définition et comparaison de date

   protected int cpt[] ;

   // Opérateur de précédence
   public boolean Prec ( DateV d1, DateV d2 )  {
      for ( int i = 0 ; i < cpt.length ; i++)
          if (d1.cpt[i] > d2.cpt[i]) return false ;
      return true;
   }

   // Constructeur
   DateV(int dim) {
      cpt = new int[dim] ; for ( int i=0 ; i < cpt.length ; i++) cpt[i] = 0 ;
   }
}
```

On peut remarquer que la notion de site n'intervient plus en composante d'une date puisque l'on a une valeur propre à chaque site. Par contre, un objet horloge reste présent sur chaque site.

Les actions associées aux événements internes, d'émission et de réception restent les mêmes que pour les horloges de Lamport, seul le type date ayant changé (voir 3.1.2).

```
class Horloge extends DateV {
   protected int s ;  // localisation de l'horloge
   // Création de l'horloge
   Horloge( int où , int dim )  { super(dim) ; this.s = où ; cpt[où] = 1 ;}

   // Lecture-Incrémentation de l'horloge
   DateV Top() throws Exception
      { DateV dc = (DateV)super.clone(); this.cpt[s]++; return dc ; }

   // Recalage de l'horloge
   void Recaler( DateV D ) {
      for (int i=0 ; i < cpt.length ;i++)
            this.cpt[i] = Math.max(this.cpt[i],D.cpt[i]) ;
      this.cpt[s]++ ;   // Top implicite
      }
   }
```

Le tableau suivant précise les actions exécutées lors sur les événements internes ainsi que lors des émissions et réceptions.

Type d'événement sur un site s	Action mettant en jeu l'horloge du site s
Événement interne sur s	$H_s.Top()$
Émission sur s de m	$dvm = H_s.Top()$; envoi de $< dvm, m >$
Réception sur s de $< dvm, m >$	$H_s.Recaler(dvm)$

La figure (3.2) illustre l'évolution des horloges vectorielles sur 3 sites. On remarquera que la composante "locale" du vecteur ($H_s[s]$) comptabilise exactement le nombre d'événements ayant lieu sur le site. Par ailleurs, toute date "vecteur" d_e associée à un événement e mémorise le nombre d'événements causalement liés à e. À titre d'exemple, l'événement c_3 est daté $(3, 3, 3)$ et l'on constate qu'il y a effectivement 3 événements sur les sites A (a_1, a_2, a_3) et B (b_1, b_2, b_3) et 2 événements sur C (c_1, c_2) qui précèdent c_3.

Enfin, on remarquera que les événements de réceptions sont datés explicitement et comptabilisés.

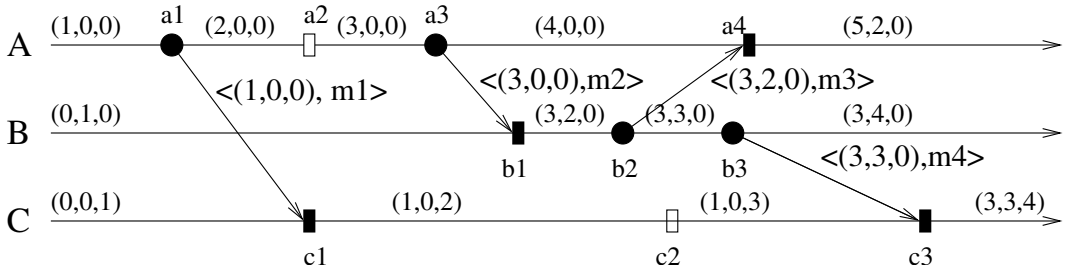


FIGURE 3.2 – Horloges de Mattern

Exercices

1. Dans le mécanisme d'horloge de Lamport, pourrait-on ne pas dater explicitement les événements d'émission ?
2. On essaie de définir un mécanisme de datation avec seulement un compteur local par site (on élimine la composante site des horloges de Lamport). Donner les règles de gestion de telles horloges pour assurer le respect de la relation : $\forall e, e' \ : \ e \prec e' \ \rightarrow \ d_e < d_{e'}$
3. Montrer sur un exemple, que le mécanisme précédent n'assure pas la réciproque :

$$\forall e, e' \ : \ d_e < d_{e'} \ \rightarrow \ e \prec e'$$

4. Montrer que les horloges de Mattern assurent :

$$\forall e, e' \ : \ e \prec e' \ \equiv \ d_e < d_{e'}$$

5. Pourrait-on ne pas dater explicitement les événements de réception avec des horloges de Mattern ? (Tout en conservant leur propriété précédente).

3.2 Protocoles d'ordre causal

Le transfert des messages dans un réseau ne respecte pas forcément des règles d'ordonnancement strictes. Par exemple, dans un protocole point à point, l'ordre de réception des messages issus d'un **même** site par un site fixé n'est pas forcément identique à leur ordre d'envoi.

Plus généralement, il peut être intéressant d'ordonner la délivrance des messages reçus par un site de façon à respecter la causalité qui peut exister entre les événements d'émission de ces messages. Pour ce faire, on doit donc distinguer d'une part, l'événement de réception d'un message par un site et d'autre part, l'événement de délivrance de ce message au niveau applicatif. En effet, des messages reçus dans l'ordre inverse de leur causalité d'émission devront être réordonnés sur le site de réception pour être délivrés dans le bon ordre.

Cette contrainte peut être formalisée de la façon suivante :

$$\forall s \ : \ (\forall m, m' \ : \ r_s \wedge r'_s \ :: \ e \prec e' \ \Rightarrow d \prec d'))$$

La figure (3.3) montre un exemple qui peut très bien se produire lorsqu'on communique par e-mail. Un usager A pose une question à deux usagers B et C en leur envoyant un message à chacun. L'usager B reçoit en premier le message « question » m. Il répond alors à A et à C (constatant que la question a été aussi envoyé à C grâce au champ destinataires du message). L'usager C reçoit alors via le message $m2$ une réponse dont il ne connaît pas la question. Celle-ci arrivera plus tard sous la forme du message $m1$. On remarquera que cette anomalie tient au fait que les deux messages $m1$ et $m2$ qui sont reçus par le site C dans l'ordre inverse de leur précédence causale d'émission puisque $e_1 \prec e_2$. Dans ce cas un protocole d'ordre causal délivrera les messages dans l'ordre inverse. Si l'on note d_1 et d_2 les événements de délivrance des deux messages, alors on aura $d_1 \prec d_2$.

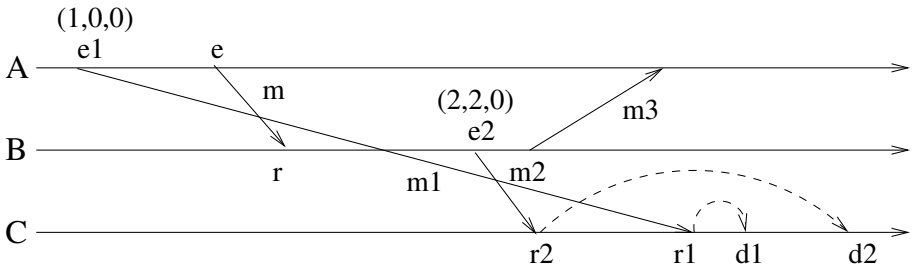


FIGURE 3.3 – Protocole point à point d'ordre causal

3.2.1 Implantation du protocole dans le cas point à point

Si l'on utilise le mécanisme d'horloge de Mattern pour dater les événements, les messages m_1 et m_2 emportent avec eux la date de leur émission. Sur l'exemple de la figure (3.3), on constate que l'on a bien $d_{e_1} \prec d_{e_2}$ (puisque $(1,0,0) < (2,2,0)$). Mais, lors de la réception du message m_2, la date d_{e_2} ne permet pas de conclure qu'un message m_1 existe et aurait dû être déjà reçu et délivré. Ce n'est que lorsque le message m_1 arrivera que l'on détectera l'anomalie de causalité dans la réception des deux messages. Par conséquent, il faut mettre en œuvre un protocole spécifique.

Approche par matrice de précédence (Raynal)

Pour décider si un message m peut être délivré à l'applicatif d'un site s, il faut savoir si tous les messages qui devaient être reçus par s et dont l'émission précède causalement ce message m sont arrivés et ont été délivrés.

Pour ce faire, un site s doit gérer :

- d'une part, un vecteur $Dernier[N]$ indiquant le numéro d'ordre du dernier message reçu par ce site et provenant de chaque site origine possible.
- d'autre part, une matrice carrée de précédence $MP[N,N]$ dont chaque élément $MP[i,j]$ indique le nombre d'émissions du site i vers le site j connu du site s.

Les actions suivantes seront exécutées sur occurrence des événements d'émission et de réception :

Type d'événement sur un site s	Action mettant en jeu l'horloge du site s
émission sur s de m vers s'	$MP_s[s,s']++$; envoi de $< MP_s, m >$
Réception sur s' de $< MP_s, m >$	Réordonner(MP_s, m)

Une description en Java de ce protocole comporte deux classes :

- d'une part, la classe *Précédence* qui décrit le type de matrice carrée NxN captant la causalité des messages ;

```
import java.util.* ;

class Précédence {
   protected int[][] cpt ;  // matrice de précédence
   protected int     orig;  // identité du créateur
   protected int     N ;    // nombre de sites

   // Création de la matrice de précédence
   Précédence ( int loc, int nbSite ) {
      cpt = new int[nbSite][nbSite] ;
      for (int i=0 ; i < nbSite; i++)
          for (int j=0 ; j < nbSite ; j++) cpt[i][j] = 0 ;
      orig = loc ; N = nbSite ;
   }
```

```
      // Recaler la matrice sur une autre
      void Max(Précédence MX) {
         for (int i=0 ; i < N ; i++)
            for (int j=0 ; j < N ; j++)
               cpt[i][j]=Math.max(cpt[i][j],MX.cpt[i][j]);
      }
   }
```

– d'autre part, la classe *Protocole* qui décrit l'algorithme de traitement des émissions (*TopEnvoi*) et des réceptions (*Réordonner*).
Le contrôle de l'ordonnancement des messages reçus pour assurer leur délivrance dans un ordre causalement correct s'appuie sur la précondition suivante pour un message $< MP, m >$ reçu par s :
  – d'une part, ce message issu du site *MP.orig* doit bien être le prochain message à délivrer (caractère FIFO de la communication entre le site émetteur et le site récepteur). Autrement dit le prédicat suivant doit être vérifié :

$$with\ Protocole@s\ ::\ \ MP.cpt[MP.orig][s] = Dernier[MP.orig] + 1$$

  – d'autre part, ce message ne doit pas précéder des messages dont l'émission le précède causalement, c'est-à-dire :

$$with\ Protocole@s\ ::\ \ \forall i \neq s\ ::\ MP.cpt[i,s] \leq Dernier[i]$$

C'est cette conjonction qui est testée par la fonction *délivrable*.
Toute réception de message se traduira par un appel à la méthode *Réordonner* qui retardera éventuellement la délivrance du message correspondant.Les messages applicatifs seront consommés dans l'ordre chronologique de terminaison des opérations *Réordonner*.

```
class Protocole  extends Précédence {

   protected int s ;          // localisation
   protected int[] Dernier ;  // vecteur de comptage des reçus
   protected List AttDel   ;  // liste d'attente de délivrance

   // Top sur envoi de message vers le site $dest$
   Précédence TopEnvoi(int dest) throws Exception {
        this.cpt[s,dest]++;
        return (Précédence)super.clone();
   }
      // test si le message associé à la matrice de précédence est délivrable
      boolean délivrable(Précédence MP) {
         for (int i=0 ; i < N ;i++)
            if ((i == s && MP.cpt[MP.orig][s] != Dernier[MP.orig] + 1)
            || (i != s && MP.cpt[i][s] > Dernier[i])) return false ;
         return true ;
      }

   // Ordonnancement de la délivrance des messages
   synchronized void Réordonner( Précédence MP, String Message ) throws Exception {
      if (!délivrable) { AttDel.add(MP) ; MP.wait() ; }
      // délivrable(MP) =>  mise-à-jour de this
      this.Dernier[MP.orig]++ ; this.Max(MP) ;
      // Réveil de messages en attente
      ListIterator curseur = AttDel.listIterator(0) ;
```

```
            while (curseur.hasNext()) {
               Précédence mp = (Précédence)  curseur.next() ;
               if (délivrable(mp)) { mp.notify() ; curseur.remove() ; }
            }
         }
         // Constructeur
         Protocole(int où , int dim) {
            super(où,dim) ; s = où ; Dernier = new int[dim] ;
            for (int i=0 ; i < N ; i++) Dernier[i]=0 ;
            AttDel = Collections.synchronizedList(new LinkedList());
         }
      }
```

Le transport d'une matrice carrée de dimension N égale au nombre de sites pose deux problèmes : d'une part, il faut connaître N, d'autre part, pour N grand, le coût en communication devient important. C'est pourquoi une autre approche a été utilisée.

Approche par gestion d'histoires

Une autre approche consiste à concaténer à tout message la suite des messages qui le précède causalement. On appelle cette technique "piggybacking". La figure (3.4) montre que la réception du message m_2 sur le site C s'accompagne en fait de la réception d'une copie du message m_1 qui était lui aussi destiné à C et placé dans l'histoire causale de m_2. Dans un tel cas, c'est le message m_1 qui sera délivré, puis le message m_2. La réception ultérieure du message original m_1 issu du site A se traduira par l'oubli pur et simple de ce message (dont une copie a déjà été délivrée à l'applicatif).

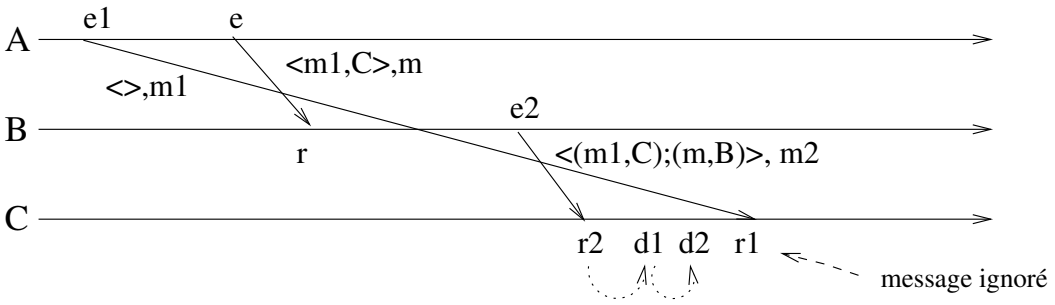


FIGURE 3.4 – Approche par piggybacking

Un avantage de cette approche est qu'elle apporte une certaine redondance par les copies de messages qui sont engendrées apportant ainsi un degré de tolérance vis-à-vis des pertes de messages. Par contre, la surcharge des messages est importante. L'histoire causale des nouveaux messages s'alourdit au fur et à mesure du calcul. Il faut pouvoir optimiser ce coût qui peut devenir prohibitif en réduisant la taille de l'histoire emportée par un nouveau message. La figure (3.5) montre un exemple d'élimination d'un message dans une histoire. Après la réception des messages m_2 et m_3, le site B "sait" que le message m_1 a été envoyé ET délivré. Par conséquent, il peut être désormais ignoré par B.

3.2.2 Implantation du protocole dans la cas de la diffusion

Pour un protocole de diffusion se pose les mêmes problèmes de causalité. La figure (3.6) illustre un début de calcul réparti utilisant la diffusion. Cependant, l'implantation d'un protocole de diffusion à ordre causal devient plus simple lorsqu'on utilise l'approche par compteurs. En effet, la matrice de précédence se simplifie : un site envoie un message vers tous les autres sites systématiquement. Par conséquent, dans cette matrice,

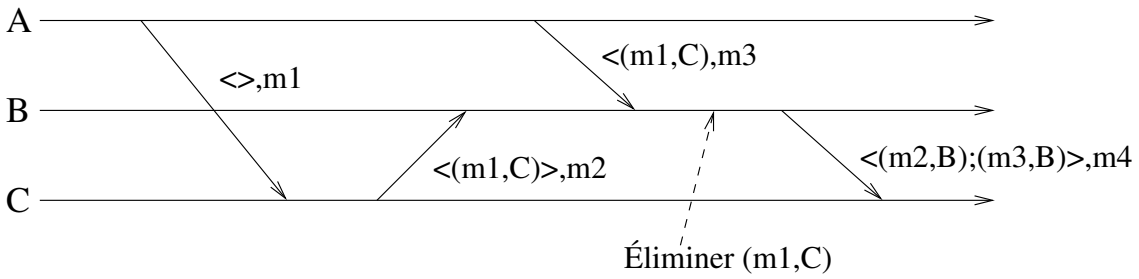


FIGURE 3.5 – Optimisation des histoires

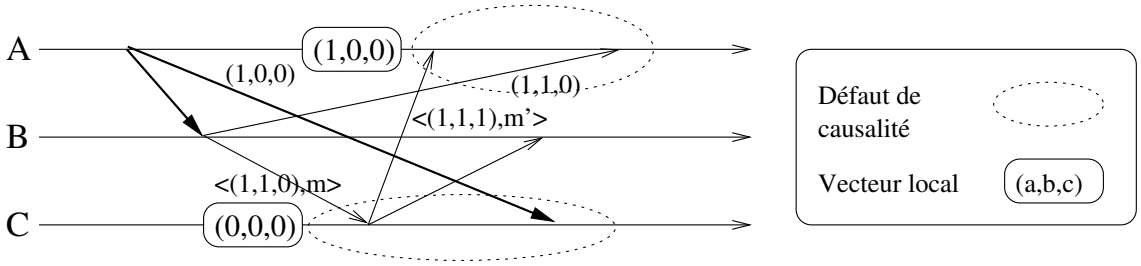


FIGURE 3.6 – Diffusion et causalité

on a $\forall j \ : \ MP[i,j] = k$. Il suffit donc d'un vecteur $VP[N]$ tel que $VP[i]$ indique le nombre de messages envoyé par i vers tout site.

Le contrôle de l'ordonnancement des messages reçus s'appuie cette fois-ci sur la précondition suivante pour un message $< VP, m >$ reçu par le site s :

– d'une part, ce message issu du site $VP.orig$ doit bien être le prochain message à délivrer :

$$with\ Protocole@s \ :: \ \ cpt[VP.orig] = Dernier[MP.orig] + 1$$

– d'autre part, ce message ne doit pas précéder des messages dont la diffusion le précède causalement, c'est-à-dire :

$$with\ Protocole@s \ :: \ \ \forall i \neq s \ :: \ VP.cpt[i] \leq Dernier[i]$$

La figure (3.6) montre que les message m et m' seront retardés puisque les vecteurs associés à ces messages ne satisfont pas la précondition attendue.

3.3 Tolérance aux fautes : modèle d'exécution ISIS, HORUS

Remarque préliminaire : Ce cours est inspiré de différents articles de Kenneth Birman, principalement les deux articles dans les Communications de l'ACM, respectivement sur Isis [3] et Horus [28], le livre réunissant divers articles sur Isis [7] et bien évidemment, le manuel d'Isis [5].

La boîte à outils Isis est issue de travaux de l'équipe de Kenneth P. Birman à Cornell University, de 1983 à 1990. Ces mêmes travaux ont défini la notion de synchronisme virtuel, sur lequel repose Isis. Isis a ensuite été commercialisé par une compagnie ultérieurement rachetée par Stratus Computer, Inc. Après avoir poursuivi le développement de la boîte à outils Isis, Stratus a intégré Isis dans Orbix pour définir un ORB Corba tolérant aux fautes. Actuellement, Isis semble avoir disparu à l'intérieur du produit Radio pour l'administration de serveurs tolérants aux fautes sous Windows NT. Parallèlement, la recherche à Cornell s'est poursuivie au travers du nouveau projet Horus. Ce projet, initialement une refonte d'Isis, a notamment étudié l'utilisation modulaire de micro-protocoles pour construire des protocoles de communication évolués.

Isis est une boîte à outils fournissant principalement des primitives de communication pour construire des applications distribuées tolérantes aux fautes, basées sur des groupes de processus coopérants. à ce titre, Isis fournit des primitives pour gérer les groupes, détecter et traiter les fautes, synchroniser des processus et, bien sûr, échanger des messages.

3.3.1 Groupe de processus

Isis est basé sur la notion de groupe de processus avec un modèle d'exécution bien défini : le synchronisme virtuel. On rencontre principalement deux types de groupes :

- Les groupes anonymes : ils apparaissent quand une application publie des données selon des "thèmes", et que d'autres processus souhaitent s'abonner à ce thème. Pour qu'une telle application soit tolérante aux fautes, les groupes anonymes doivent fournir plusieurs propriétés :
  - Il doit être possible d'envoyer un message à un groupe via une adresse, sans que le programmeur ait besoin d'expanser lui-même les membres du groupe.
  - Si l'émetteur et les souscripteurs fonctionnent correctement, un message doit être délivré exactement une fois. Si l'émetteur s'arrête, un message doit être délivré à tous les souscripteurs ou à aucun d'entre eux. Le programmeur de l'application n'a pas à se préoccuper des messages perdus ou dupliqués.
  - Les messages doivent être délivrés aux abonnés selon un ordre bien défini. Par exemple, une application pourra nécessiter un ordonnancement causal.
  - Il doit être possible à un abonné d'obtenir un historique du groupe : l'ensemble des événements importants (entrée et sortie de membres du groupe) et des messages échangés, ainsi que l'ordre de ces événements.
- Les groupes explicites : un groupe est explicite quand ses membres coopèrent directement. Ils savent qu'ils sont membres du groupe et utilisent explicitement la liste des membres. Un changement dans la liste des membres est une information visible et utilisée dans un groupe explicite. Par exemple, un service tolérant aux fautes peut être réalisé avec un serveur primaire qui réalise le service et un ensemble ordonné de serveurs secondaires qui prendront successivement en charge le service si le primaire s'arrête. Si tous les événements ne sont pas vus dans le même ordre par tous les sites, il peut arriver qu'aucun primaire n'existe, ou, à l'inverse, que plusieurs sites pensent être le serveur primaire. On utilise aussi un groupe explicite lorsqu'on souhaite paralléliser un traitement sur n sites, en assurant un partage correct du travail.

Lorsque l'on souhaite utiliser des groupes de processus, trois problèmes doivent être résolus :

- support pour la communication de groupe : il faut pouvoir adresser un groupe, échanger des messages en respectant certaines propriétés d'ordonnancement (fifo, causal, total), et gérer atomiquement les défaillances ;
- connaissance de la relation d'appartenance (membership) : il faut pouvoir connaître à quel(s) groupe(s) appartient un site, de quels sites est composé un groupe donné, quelle a été l'évolution d'un groupe...
- synchronisation : pour obtenir un comportement correct d'une application distribuée, il est nécessaire de synchroniser l'ordre dans lequel les actions des différents membres d'un groupe sont effectuées.

L'intégration de ces trois problèmes interdépendants forme la base du modèle d'exécution nommé synchronisme virtuel. Avant de présenter ce modèle, commençons par un rapide survol des solutions traditionnelles.

3.3.2 Les solutions traditionnelles

Transfert de messages

Les systèmes d'exploitation fournissent généralement les mécanismes suivants pour communiquer :

- datagrammes : le seul service assuré est la détection de corruption. Hors cela, les messages peuvent être perdus, dupliqués ou réordonnés.
- appel de procédure à distance : un RPC est relativement fiable mais en cas d'échec, l'émetteur ne peut généralement pas savoir si le réseau a perdu la requête ou perdu la réponse ou si le destinataire s'est arrêté avant ou après avoir traité la requête.

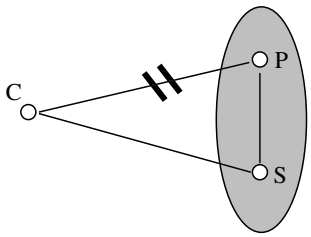


FIGURE 3.7 – Etat incohérent lors d'une faute

– canaux fiables : il s'agit du moyen de communication le plus courant. Un canal fiable assure la délivrance fiable des messages dans leur ordre d'émission. Cependant, des difficultés similaires à celles du RPC existent aussi. Considérons l'exemple de la figure 3.7 : le client C est connecté à un serveur primaire P et à son secours S. La connexion entre C et P est rompue. Le client C et le primaire P ne peuvent connaître l'état de l'autre site (bien souvent, ils sont même incapables de savoir si la rupture de la connexion provient d'une défaillance du réseau ou d'un arrêt de l'autre site). Pire, le secours S n'est pas au courant du problème, et, constatant que le primaire est toujours opérationnel, ignorera les requêtes de C.

Gestion des groupes et ordonnancement

La gestion des groupes se fait au moyen d'un service d'appartenance (membership service). Le rôle de ce service est d'assurer la liaison entre l'adresse d'un groupe et la liste de ses membres. Ce service doit évidemment être tolérant aux fautes. Lorsqu'un site diffuse un message aux membres d'un groupe, il est nécessaire que cette liste de membres soit valide, et ne corresponde pas un état intermédiaire du service d'appartenance (atomicité des mises à jour).

La notion de temps logique compatible avec la causalité, introduite par Lamport en 1978, est fondamentale à la bonne intégration entre évolution des groupes et délivrance des messages. L'idée est qu'un message diffusé à un groupe doit être délivré à tous les membres au même instant. En l'absence d'horloges synchronisées, il est impossible d'assurer cette délivrance au même instant physique. Par contre, il est possible d'assurer que, sur tous les sites, l'ordre de délivrance est "raisonnable", c'est-à-dire compatible avec les liens de causalité. Un observateur est alors incapable de distinguer entre une délivrance au même instant et une délivrance "au bon moment".

La figure 3.8 illustre plusieurs des problèmes qui peuvent survenir si le service d'appartenance et le système de diffusion n'assurent pas un bon ordonnancement. Nous considérons un groupe composé de trois sites S_1, S_2, S_3. Les messages m_1 et m_2 sont émis sans lien causal par deux clients. Avec une diffusion causale, ils peuvent être délivrés dans des ordres différents aux membres du groupe (par exemple sur S_1 et S_2) ; une telle situation est par contre interdite avec une diffusion atomique[1]. Suite à m_2, le site S_1 diffuse un message m_3 à l'intérieur du groupe. Le schéma présente une délivrance non causale sur S_3 : ce site reçoit le message m_3 avant le message responsable m_2. Enfin, nous supposons que le site S_3 s'arrête suite à une défaillance. Une nouvelle liste d'appartenance est constituée avec seulement S_1 et S_2. Le message m_4 est délivré à S_1 avant qu'il n'ait connaissance de l'arrêt de S_3, mais il est délivré à S_2 après la nouvelle constitution. De manière symétrique, lorsque S_3 rejoint le groupe, le message m_5 est délivré et traité par S_1 avant la jonction, mais S_2 n'en a pas connaissance lorsqu'il transmet son état à S_3.

Tolérance aux fautes

En se basant sur les mécanismes de communication précédemment évoqués, il est possible de construire un protocole de diffusion qui tolère les arrêts de site, en particulier de l'émetteur. Cependant de tels protocoles

1. Dans Isis, une diffusion est dite atomique si elle impose un ordre total compatible avec la causalité vis-à-vis des autres diffusions atomiques.

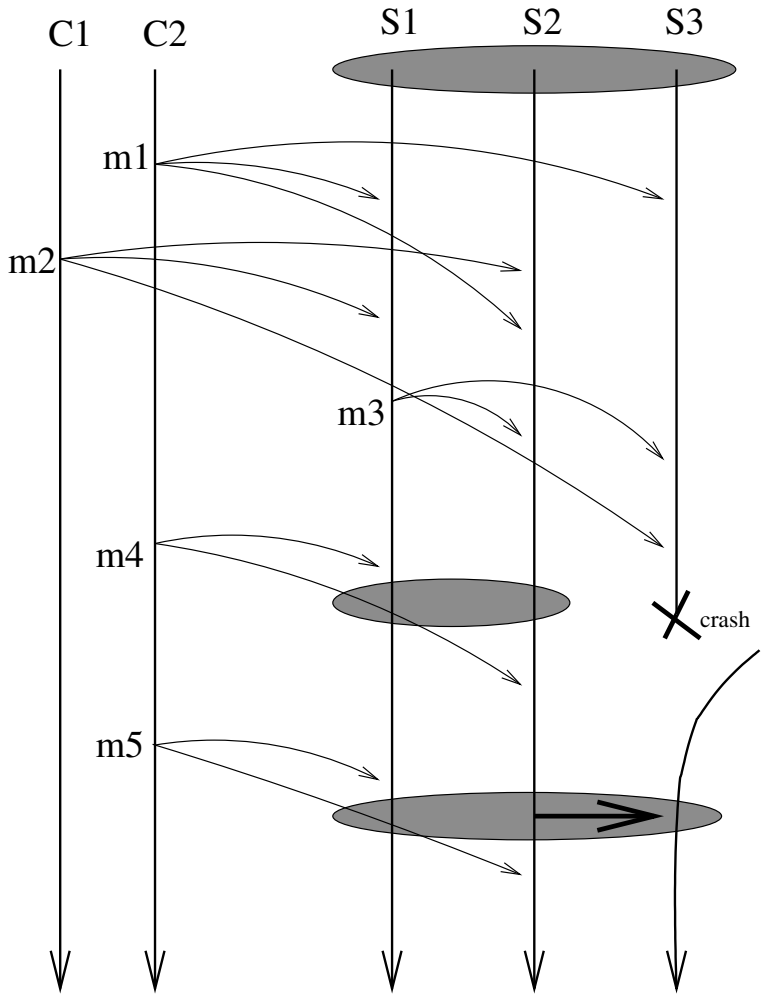


FIGURE 3.8 – Mauvais ordonnancements de messages

sont relativement complexes et coûteux (la méthode naïve conduit à $5n$ messages, où n est la taille du groupe). En outre, il est difficile d'intégrer de manière cohérente la délivrance d'événements comme l'apparition ou la disparition d'un membre.

3.3.3 Le synchronisme virtuel

Pour résumer, l'utilisation des systèmes d'exploitation traditionnels soulève plusieurs problèmes : le support pour les communications fiables (notamment les diffusions) est faible et conduit à des incohérences lors des ruptures de canaux ; l'expansion de l'adresse d'un groupe en sa liste de membres doit être faite explicitement par le programmeur ; l'ordonnancement des messages concurrents ou causalement liés doit être fait au sein même de l'application ; la gestion cohérente des fautes est complexe et coûteuse. C'est dans ce cadre qu'Isis propose un ensemble de primitives de diffusion et de gestion des groupes associées à un modèle d'exécution simplifiant l'écriture d'applications distribuées.

Synchronisme fort

Idéalement, on voudrait pouvoir développer des applications dans le modèle de synchronisme fort (close synchrony), dans lequel les propriétés suivantes sont garanties :

– les communications sont fiables ;
– une diffusion est considérée comme un événement atomique, i.e. de temps logique nul ;
– l'expansion d'une adresse de groupe en sa liste de membres est un événement atomique qui se produit à l'instant logique de la diffusion ;
– les messages concurrents sont délivrés à tous les destinataires dans le même ordre ;
– les messages causalement liés sont délivrés selon un ordre compatible avec cette causalité ;
– le transfert de l'état d'un site lors de la jonction d'un nouveau site est effectué atomiquement vis-à-vis de tous les autres événements (notamment les diffusions) ;
– les fautes sont atomiques vis-à-vis des diffusions, et sont ordonnées identiquement sur tous les sites.

Malheureusement, un tel synchronisme fort est, d'une part, impossible à réaliser en présence de fautes, et d'autre part, extrêmement coûteux à obtenir. Il impose généralement une exécution pas à pas de tous les sites, et désavantage donc les interactions asynchrones, dans lesquelles un site continue son exécution immédiatement après avoir initié une communication sans attendre la délivrance du message. Le modèle de synchronisme virtuel est un affaiblissement du modèle de synchronisme fort, où l'atomicité des communications est relâchée de manière à privilégier les communications asynchrones mais où le respect d'un ordre bien défini de délivrance est conservé.

Ordre total et ordre causal

Le synchronisme fort impose un ordre total entre tous les événements. Cet ordre total peut être obtenu au moyen de la *diffusion atomique*, nommée ABCAST dans Isis. Cependant, dans nombre de cas, l'ordre causal suffit. Selon l'ordre causal, des événements causalement liés doivent être délivrés selon ce lien causal. Par exemple, sur la figure 3.8, le message m_3 est causalement ultérieur à m_2, il doit donc être délivré après. Par contre, il n'y a pas de lien causal entre m_1 et m_2 qui peuvent donc être délivrés dans des ordres différents selon les sites. La diffusion causale, ou CBCAST, assure un tel ordre causal. L'intérêt principal de la diffusion causale est son faible coût, tant d'un point de vue retard de délivrance que d'un point de vue surcoût de communication : un message d'une diffusion atomique ne peut être délivré que quand le site est sûr que tous les ABCAST précédents (vis-à-vis de l'ordre total) sont terminés ; pour une diffusion causale, il suffit de s'assurer que les messages causalement précédents ont été délivrés. Cela signifie en outre que, si l'émetteur est membre du groupe auquel il diffuse, il peut se délivrer immédiatement le message. L'implantation performante du CBCAST d'isis se base sur des vecteurs d'horloges, ce qui entraîne un faible surcoût dans les messages.

Les changements dans les groupes (dus à un retrait volontaire ou à une défaillance pour les réductions) doivent par contre être ordonnés totalement vis-à-vis des délivrances, pour éviter des incohérences similaires à celles illustrées par les messages m_4 et m_5 de la figure 3.8.

3.3.4 La boîte à outils Isis

Hypothèses sur l'environnement

La boîte à outil Isis fait les hypothèses suivantes sur son environnement : le réseau est constitué de réseaux locaux (LAN) interconnectés par des réseaux à grande distance (WAN), beaucoup plus lents. Les horloges des différents sites ne sont pas synchronisées. Sur un LAN, les messages peuvent être perdus, dupliqués ou ré-ordonnés ; si une partition se produit, une partition majoritaire est déterminée et sera la seule à pouvoir continuer à travailler. Les seules fautes de sites sont des arrêts, sans action ou émission illégale (crash failure). Du fait de la règle de la majorité pour les partitions, il est déconseillé d'utiliser des groupes répartis sur plusieurs réseaux locaux, mais plutôt d'utiliser un paquetage spécial de communication pour les réseaux à grande distance.

Styles de groupes

Isis est optimisé pour traiter de manière efficace quatre styles d'utilisation des groupes (figure 3.9), sans qu'il soit pour autant nécessaire de se conformer à l'une de ces structures :

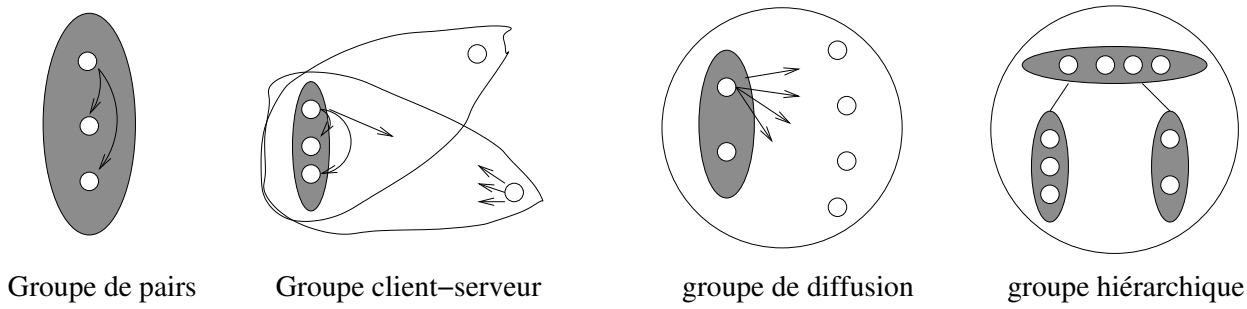


FIGURE 3.9 – Styles de groupes

- groupe de pairs (peer group) : tous les processus coopèrent étroitement, par exemple pour répliquer des données. L'application utilise explicitement l'appartenance et un ordre parmi les membres.
- groupe client-serveur : un client communique avec un groupe qui remplit un service. Le client n'est pas membre du groupe, mais, pour des raisons de performance, il peut s'enregistrer en tant que *client*.
- groupe de diffusion : c'est un groupe client-serveur où les clients sont enregistrés et où les serveurs diffusent à tous les clients sans que ceux-ci ne déposent de requêtes.
- groupes hiérarchiques : pour éviter qu'un groupe ne soit composés de trop de membres, il peut être utile de construire une structure hiérarchique arborescente pour structurer ces membres. Une application contacte initialement le groupe racine puis ne communique plus qu'avec un des sous-groupes. La plupart des communications se limitent à l'intérieur d'un sous-groupe, et ne concernent que rarement l'ensemble des sous-groupes.

3.3.5 Programmer avec Isis

La programmation avec Isis est un programmation de type événementiel, où le développeur déclare des points d'entrée pour gérer la réception d'un message. Quand un message est reçu, Isis crée une nouvelle tâche pour exécuter le code associé au point d'entrée.

Structure de l'application

Une application suit les étapes suivantes :

1. initialise les librairies Isis et se connecte à Isis ;
2. déclare les tâches et les points d'entrées ;
3. déclare les gestionnaires de signaux et d'entrées-sorties ;
4. initialise l'état de l'application ;
5. se joint à des groupes et/ou devient client ;
6. indique à Isis qu'elle est prête à recevoir les messages.

Initialisation et connexion à Isis

Trois méthodes d'initialisation sont disponibles :

- connexion à l'Isis local :
  `int isis_init (int client_port);`
  Se connecte à Isis via la port tcp `client_port`. Il s'agit du deuxième port listé dans le fichier *sites* (généralement 1602).
  Une version longue permet de spécifier de nombreux drapeaux, notamment `ISIS_AUTORESTART` (la bibliothèque tente de démarrer Isis s'il ne tourne pas actuellement sur la machine) et `ISIS_TRACE` (Isis affichera tous les messages reçus ou envoyés).

– connexion à un Isis distant :
`int isis_remote (char *mother_host, int flags);`
Tente de se connecter à l'Isis tournant sur `mother_host`
– connexion libre : les deux précédentes formes d'initialisation sont remplacées par une unique routine :
`int isis_remote_init (char *hostlist, int lport, int rport, int flags);`
`hostlist` est une chaîne de caractères contenant la liste des machines, séparées par des virgules, où tenter de se connecter. Utiliser `(char *)0` pour ne tenter que la machine locale. `lport` est le port local de connexion à une machine locale (port tcp, 1602). `rport` est le port à utiliser pour trouver Isis sur une machine distante (port bcast, 1603).

Déclaration des tâches et point d'entrée

Déclaration des tâches Cette partie n'est pas obligatoire, mais est très utile pour que les enregistrements ou les traces contiennent des noms symboliques, plutôt que des numéros arbitraires alloués par Isis.

`int isis_task (void (*routine) (void *arg), char *taskname);`

Une tâche est une procédure prenant un paramètre de type pointeur.

Déclaration des points d'entrées Cette partie définit les points d'entrées où seront reçus les messages. Elle est donc obligatoire, si jamais l'application espère recevoir des messages.

`int isis_entry (int entry, void (*handler) (message *mp), char *name);`

`entry` est le numéro de l'entrée. Il doit être strictement positif, et identifie l'entrée (voir 3.3.5).

`handler` est la routine à appeler quand un message est reçu. La routine sera appelée avec le message reçu en paramètre. Pour exécuter cette routine, Isis créera une nouvelle tâche qui sera détruite à la fin du traitement.

`name` permet à Isis d'être plus bavard dans ses explications.

Initialisation de l'application et entrée dans Isis

Ceci se fait par :

`void isis_mainloop (void (*task) (void *arg), void *arg);`

Isis crée alors une tâche pour exécuter `task` (avec l'argument passé en paramètre). Cette tâche a pour but d'initialiser complètement l'application, de se joindre à des groupes ou de devenir client d'autres groupes.

Quand l'application a fini de s'initialiser (ie s'est connectée aux groupes souhaités, ...), il est nécessaire d'indiquer à Isis que l'application est maintenant prête à recevoir des messages. Ceci est fait au moyen de :

`void isis_start_done (void);`

Tant que l'application n'a pas appelée cette routine, elle ne peut pas émettre ou recevoir de messages, et aucune tâche ne peut être créée implicitement. Si la tâche principale créée par `isis_mainloop` se termine sans avoir appelée `isis_start_done`, Isis le fait automatiquement.

Groupes et clients

Jonction à des groupes La routine de jonction à un groupe est :

`address *pg_join (char *gname, [int key, [arg,]...]..., 0);`

`gname` est le nom du groupe auquel l'application essaie de se connecter. Un nom de groupe a une syntaxe de nom de fichier Unix, ie "/sys/databases/CCP". Par défaut, le "working group" est /. Contrairement à Unix, les groupes intermédiaires n'ont pas besoin d'exister. En cas de succès, `pg_join` renvoie une adresse qui sera utilisée pour envoyer des messages.

Pour tester l'échec, il convient d'utiliser la routine :

`int addr_isnull (address *adr);`

qui renvoie vrai si l'adresse est nulle (invalide).

Enfin, `pg_join` accepte de nombreux paramètres optionnels. Les principaux sont :

– `PG_DONTCREATE` (pas de paramètre) : ne crée pas le groupe s'il est inexistant (par défaut le groupe est créé).

- `PG_LOGGED` (4 paramètres non détaillés) : pour enregistrer tout ce qui se passe dans le groupe.
- `PG_INIT`, `void (*routine) (address *)` : la routine est appelée quand un groupe est créé et qu'aucun enregistrement ne permet de reconstruire un état.
- `PG_MONITOR`, `void (*routine) (groupview *, void *arg), void *arg` : la routine est appelée à chaque changement dans la composition du groupe (jonction ou retrait de membres). Noter que la routine sera aussi appelée pour **cette** jonction.
- `PG_XFER`, `int domain, void (*send) (int position, address *group, address *whojoined), void (*receive) (int position, message *msg)` : permet l'initialisation de l'état d'un nouveau membre par copie de l'état d'un autre membre. Dans notre cas, `domain` est généralement spécifié à 0. Voir 3.3.5 pour les détails.

Client Un client obtient l'adresse d'un groupe avec :

```
address *pg_lookup (char *groupname);
```

Comme précédemment, on utilise `addr_isnull` pour vérifier si le groupe existe. Un client peut utiliser l'adresse qu'il a obtenue par `pg_lookup` pour envoyer une requête et obtenir la réponse.

Par ailleurs, un client régulier d'un groupe peut avoir intérêt à se déclarer comme tel. Outre le fait qu'il existe des moyens pour obtenir la liste des clients déclarés d'un groupe, cela permettra aussi au client de détecter des événements survenant dans le groupe. Un processus se déclare client avec :

```
int pg_client (address *groupaddr, char *credentials);
```

Le paramètre `credentials` sert à l'authentification, et peut être `NULL` si le groupe ne fait pas d'authentification.

Surveiller l'état d'un groupe Il est possible de surveiller l'état d'un groupe, de façon encore plus riche que le `PG_MONITOR` de `pg_join`. En outre, il n'est pas nécessaire d'être membre du groupe. Nous ne présentons ici que la routine qui détecte la disparition d'un groupe. Il s'agit de :

```
int pg_detect_failure (address *group, int (*routine) (address *, int what, void
*arg), void *arg);
```

Quand le groupe spécifié disparaît, une tâche est créée pour exécuter le paramètre procédure de `pg_detect_failure` avec `routine(group, W_FAIL, arg);`. Une telle surveillance est très coûteuse sauf dans deux cas particuliers : soit le processus est membre du groupe qu'il surveille, soit il est client *et* il s'est déclaré comme tel avec `pg_client`.

Il existe deux autres routines pour surveiller un groupe ou un processus, qui sont `pg_monitor` et `pg_watch`. Ces routines ne marchent que pour un membre du groupe ou un client déclaré. Pour un membre du groupe, `pg_monitor` est identique à l'argument `PG_MONITOR` de `pg_join`.

Transfert de l'état Lorsqu'un nouveau membre est introduit dans un groupe, il peut être nécessaire de lui donner l'état courant du groupe. Pour cela, `PG_XFER` est fourni. Isis sélectionne un membre actif du groupe, et appelle la routine `send` de ce membre. Si tout va bien, ce membre enverra (au moyen de `xfer_out`, voir 3.3.5) l'état au nouveau membre. La routine `receive` du nouveau membre est appelée pour chacun des messages envoyés, et il récupérera l'état au moyen de `msg_get` (voir 3.3.5).

Comme le transfert d'un état peut nécessiter plusieurs messages, le paramètre `position` de `send` et `receive` permet de déterminer où en est le transfert. Noter enfin que Isis s'occupe de tout vis-à-vis des éventuelles défaillances du membre retenu pour transférer l'état (un autre membre sera choisi et Isis lui fournira, au moyen du paramètre position, l'endroit où le précédent s'était arrêté).

La routine `void xfer_refuse` peut être utilisée par un membre du groupe pour signifier à Isis que ce membre n'est pas apte à transférer l'état (et Isis en trouvera alors un autre).

Autres routines sur les groupes Quelques routines utiles :

`int pg_leave (address *);` pour quitter un groupe (que le processus soit membre ou client déclaré).

`int pg_delete (address *);` pour détruire un groupe.

`int pg_rank (address *group, address *paddr);` renvoie le rang du processus `paddr` dans le groupe spécifié. Tout processus possède un rang unique dans le groupe (commençant à 0), qui est fonction du moment où `pg_join` a été appelé. `pg_rank` retourne -1 si `paddr` n'est pas membre du groupe. Ce rang est souvent utilisé lorsqu'il faut élire un coordinateur ou un maître.

`address my_address;` est l'adresse du processus courant. Elle peut être utilisée avec `pg_rank` pour obtenir son propre rang dans un groupe.

Diffusions et messages

Nous ne présentons ici que les formes les plus simples de l'envoi de messages. Il en existe de nombreuses variantes pour les situations les plus particulières. Toutes les routines d'émission et de réception de messages utilisent des formats à la printf/scanf, plus de nombreuses extensions pour manipuler les tableaux ou les entités d'Isis (`address`, `message`,...).

émission de message

Diffusion sans réponse : Pour envoyer un message sans réponse, utiliser :

`int bcast (address *addr, int entry, char *outformat, [arg,]..., 0);`

où `addr` est l'adresse du groupe ou du processus concerné, et `entry` est l'entrée sur laquelle délivrer le message. `bcast` peut être `fbcast`, `cbcast`, `abcast` ou `gbcast`, selon que l'on souhaite une diffusion FIFO, causale, atomique ou de groupe (la diffusion de groupe est une diffusion imposant un ordre total vis-à-vis de *toutes* les diffusions. En ce sens, elle est similaire à un événement de changement dans la composition d'un groupe).

Dans le cas où aucune réponse n'est attendue, il s'agit d'un envoi asynchrone (l'application continue immédiatement après le bcast).

Diffusion avec réponse : Pour envoyer un message avec réponses attendues, utiliser :

`int bcast (address *addr, int entry, char *outformat, [arg,]..., int nwant, char *replyformat [, replyarg] ...);`

`nwant` indique le nombre de réponses attendues. Ce peut être un entier (notamment 1) ou les constantes `MAJORITY` ou `ALL`. Si `nwant` n'est pas nul, il s'agit d'un appel bloquant, jusqu'à réception des `nwant` messages. Si l'application est membre du groupe vers lequel elle diffuse, elle recevra aussi ce message.

Transfert de l'état : Le transfert de l'état se fait au moyen de :

`int xfer_out (int position, char *format [, arg]...);`

Réception d'un message

Un message récupéré comme paramètre d'un point d'entrée (ou de la routine de réception de l'état) est analysé au moyen de :

`int msg_get (message *msg, char *format [, arg]...);`

Réponse à un message

Pour répondre à un message, utiliser :

`int reply (message *m, char *outformat [, arg]...);`

`m` est le message auquel le processus répond.

Pour éviter de répondre (alors qu'une réponse était attendue), on utilise :

`int nullreply (message *m);` Ceci est différent de `reply (m, "");` qui indique une réponse vide.

Enfin, il est possible de répondre avec

`int abortreply (message *m);`

Dans ce cas, la diffusion du client se termine immédiatement en retournant -1, et `isis_errno` contient `IE_ABORT`.

Vérification de la cohérence questions/réponses

Quand un processus diffuse un message avec `ALL` réponses, comment sait-il qu'il a bien eu autant de réponses que de messages envoyés ? Deux variables peuvent alors servir :

```
int isis_nsent;
int isis_nreplies;
```

La variable `isis_nsent` contient le nombre de messages envoyés par le dernier bcast. La variable `isis_nreplies` contient le nombre de réponses récupérées. Les non-réponses générées par `nullreply` ne sont pas comptabilisées dans `isis_nreplies`.

Pour être précis, une diffusion se termine quand l'une des trois conditions suivantes devient vraie :

- l'initiateur de la diffusion a reçu le nombre de messages désiré ;
- chacun des processus qui a reçu le message a renvoyé une réponse (par `reply`), ou a renvoyé une non-réponse (par `nullreply`), ou s'est arrêté ;
- un des processus a renvoyé un abort (par `abortreply`).

Les tâches

Les tâches dans Isis sont non-préemptives sauf si Isis a été construit avec une librairie de tâches préemptives (Sun lwp ou Mach), et que l'application le demande explicitement. Chaque tâche possède sa propre pile.

Des tâches sont créées implicitement pour appeler les points d'entrée de messages ou les handlers de signaux et d'entrées-sorties.

Il est par ailleurs possible de créer explicitement une tâche avec :

```
int t_fork (void (*routine)(void *arg), void *arg);
int t_fork_msg (void (*routine) (message *arg), message *arg);
```

La routine est alors appelée avec le deuxième paramètre de `t_fork`. Quand la routine se termine, la tâche est terminée.

Si une tâche bloque, une autre tâche active peut être exécutée, et la tâche bloquée sera réactivée quand la condition de bloquage disparaîtra. Si aucune tâche n'est active, Isis attend qu'une tâche devienne active. Une tâche bloque sur certaines opérations telles l'envoi de messages avec réponses, ou `t_wait`.

Synchronisation par jeton

à partir de la diffusion atomique, il est aisé de construire un jeton d'exclusion mutuelle dans un groupe. La gestion de l'arrêt du processus possédant le jeton est par contre un peu plus délicate. Isis fournit directement des routines de manipulations de jeton.

Obtention du jeton `int t_request (address *gaddr, char *tokenname, int pass_on_fail);`

Un jeton est identifié par son nom (une chaîne de caractères). Lors de `t_request`, si le jeton n'existe pas, il est créé. Le paramètre `pass_on_fail` est `TRUE` si le jeton doit être transmis quand le processus le possédant s'arrête sans l'avoir transmis. S'il vaut `FALSE`, le jeton n'est pas transmis automatiquement en cas de panne.

Passage du jeton `int t_pass (address *gaddr, char *name);`

Normalement, seul le processus possédant le jeton peut le passer. Cependant, si `t_request` avait été appelé avec `pass_on_fail` à `FALSE`, n'importe quel processus peut utiliser `t_pass` en cas de défaillance du processus possédant. Il obtient alors le jeton.

Possesseur du jeton `address *t_holder (address *gaddr, char *name);`

permet d'obtenir l'adresse du possesseur du jeton.

3.3.6 Horus

Horus présente un modèle de communication de groupes extrêmement flexible. Pour cela, les protocoles évolués sont obtenus en empilant des micro-protocoles rendant des services bien spécifiques. Basé sur l'expérience d'Isis, Horus permet d'obtenir une implantation très performante du modèle de synchronisme virtuel, mais il n'interdit pas d'autres modèles plus adaptés à une application donnée, par exemple dans le domaine du multimédia ou du temps réel. Horus fournit une architecture où le protocole supportant un groupe peut être modifié dynamiquement à l'exécution, en fonction des besoins de l'application et des évolutions de l'environnement.

Voici quelques-uns des modules fournis par Horus :

COM : émission de message vers une liste de destinataires et réception d'un message, en utilisant des protocoles de plus bas niveau tel UDP ou ATM ;

NAK : assure l'ordre FIFO et la fiabilité, au moyen d'accusés négatifs de réception ;

FRAG : fragmentation/réassamblage de messages ;

MBRSHIP : assure, au moyen d'un algorithme de consensus, la correspondance entre nom de groupe et liste des membres accessibles ;

TOTAL : assure la délivrance selon un ordre total ;

CRYPT : chiffrage/déchiffrage ;

MERGE : transfert d'état lors de l'intégration d'un nouveau membre ou lors de la résolution d'une partition.

Par exemple, le protocole TOTAL :MBRSHIP :FRAG :NAK :COM :ATM correspond à une implantation de la diffusion atomique sur ATM. MERGE :MBRSHIP :CRYPT :FRAG :NAK :COM :UDP correspond à la diffusion causale avec déchiffrage des messages et transfert de l'état.

Pour pouvoir empiler flexiblement les modules, les interfaces (montantes et descendantes) sont standardisées. Par exemple, il existe un appel descendant pour émettre un message, et un appel montant pour recevoir un message. Ces interfaces sont définies par le Horus Common Protocol Interface. Ces interfaces se décomposent en deux catégories : celles qui traitent l'échange de messages, et celles qui concernent la gestion des groupes.

Ensemble est une nouvelle étape dans le projet Horus. Ensemble est une implantation des protocoles d'Horus en Caml qui s'attache spécifiquement à la manipulation des micro-protocoles pour reconfigurer dynamiquement des macro-protocoles. La structure en pile permet de simplifier le problème de la reconfiguration : la décision est à la charge du niveau supérieur (souvent l'application) et la réalisation est à la charge de la couche inférieure. Ces travaux ont aussi permis de montrer qu'il est possible de compiler une pile de protocoles pour le cas général de manière à obtenir une performance accrue en évitant le surcoût dû à un nombre trop élevé de couches. Par exemple, sur un réseau local, les messages sont rarement perdus ou délivrés dans le désordre et les changements dans la composition des groupes sont relativement peu fréquents. On souhaite donc pouvoir court-circuiter les couches responsables de ces traitements.

3.3.7 Bibliographie

Compte tenu de la durée du projet et de son ampleur, les publications sur Isis et Horus sont très nombreuses. Pour débuter, l'article [3] est le premier article à lire, qui, plus qu'Isis lui-même, présente les concepts de groupe et de synchronisme virtuel tel qu'ils ont été utilisés dans Isis. Pour continuer, le livre [7] réunit de nombreux papiers sur Isis, sur les concepts (reprenant l'article précédent et détaillant le synchronisme virtuel ainsi que les notions de causalité), l'implantation des différents protocoles et de la gestion des groupes dans Isis, et différentes applications développées au-dessus d'Isis. Le problème des réseaux à grande distance (WAN) est abordé dans [16], ce problème difficile étant malheureusement trop souvent ignoré. Sur la tolérance aux fautes dans Isis, [6], [29] et [2] présentent clairement le problème et les choix effectués.

La littérature sur Horus est heureusement encore assez limitée. On trouve un premier papier dans [7], mais l'article [28] est la meilleure introduction à Horus, à compléter ensuite par [26]. Ensemble est présenté dans [27]. Enfin, de manière plus générale, je ne peux que conseiller la lecture enrichissante du livre [4].

Chapitre 4

Synchronisation et supervision

4.1 L'exclusion mutuelle

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

On peut distinguer plusieurs classes d'algorithmes :

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

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

4.1.1 Rappel de la spécification du problème

On considère N processus (ou sites) P_i dont le graphe de transitions d'états peut être abstrait vis-à-vis du problème par la figure (4.1.1). Un processus cycle sur les trois états successifs : *hors*, *demandeur*, *exclusion*.

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

1. Sûreté 1 : exclusion mutuelle

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

2. Sûreté 2 : maintien des demandes

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

3. Vivacité : toute requête est finalement satisfaite

$$\forall\, i \;::\; P_i.demandeur \;\mapsto\; P_i.exclusion$$

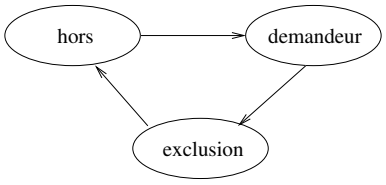


FIGURE 4.1 – Comportement d'un processus

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

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

4.1.2 L'utilisation d'un jeton circulant

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

- la circulation du jeton entre les processus consomme des ressources même lorsque les processus ne sont pas demandeurs. Pour palier ce défaut, des algorithmes ont été proposés pour bloquer le jeton sur un site s'il n'existe pas de requête en attente. Un algorithme optimal vis-à-vis de la circulation du jeton consiste à guider celui-ci de site en site demandeurs (Algorithme de Naïmi Trehel [20])
- cette approche repose sur la fiabilité de la communication. En effet, la perte du message jeton empêche tout processus d'entrer en exclusion. C'est pourquoi des algorithmes permettant de réengendrer un jeton ont été proposés. La difficulté principale tient alors à la détection de la perte du jeton (pas de fausse détection) et à l'unicité du jeton engendré (pas de double), ce qui soulève un problème d'élection (un seul processus "élu" doit engendré un seul jeton).

4.1.3 Les algorithmes à base de permission

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

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

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

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

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

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

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

Exemple d'algorithme à permissions d'arbitre

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

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

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

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

i prend pour arbitre :	1	2	3	4	5
1		•	•		
2			•	•	
3		•		•	
4		•		•	
5		•	•		

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

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

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

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

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

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

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

1	2	3
4	5	6
7	8	9

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

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

Exemple d'algorithme à permissions individuelles

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

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

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

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

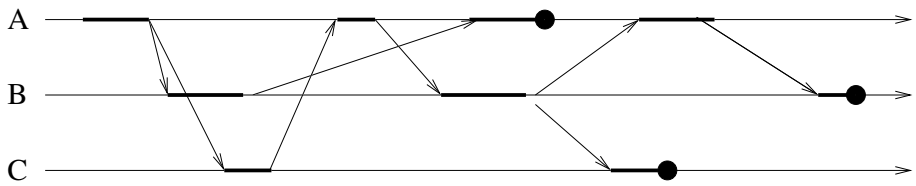


FIGURE 4.2 – Un exemple de calcul diffusant

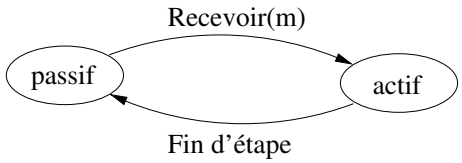


FIGURE 4.3 – Comportement d'un processus

4.2 Le problème de la terminaison d'un calcul réparti

La terminaison d'un calcul dans un contexte réparti est un problème délicat et beaucoup plus complexe qu'en milieu centralisé. Il a été particulièrement étudié sur un modèle de calcul générique appelé calcul diffusant. Ce modèle a donné lieu à de nombreux algorithmes depuis Dijkstra [11] jusqu'à Mattern [17, 18]. Il a été notamment mis en évidence l'étroite relation entre ce problème et ceux du ramasse-miettes [31] ou de la détection d'un interblocage dans un contexte d'exécution répartie [23].

4.2.1 La notion de calcul diffusant

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

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

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

Le chronogramme de la figure (4.2) montre un exemple de calcul diffusant. Un processus commute ainsi de l'état passif en attente de message à l'état actif pour exécuter un pas de calcul qui se terminera par l'envoi éventuel d'un ou plusieurs messages.

4.2.2 Spécification du problème

On considère N processus (ou sites) P_i dont le graphe de transitions d'états peut être abstrait vis-à-vis du problème par la figure (4.2.2). Un processus cycle sur les deux états successifs : $passif$, $actif$. Initialement, tous les processus sont passifs sauf un.

La détection de la terminaison repose sur un prédicat stable : tous les processus sont passifs et il n'existe plus de messages en transit. Si un tel état global est atteint, le calcul est terminé puisqu'aucun message ne pourra plus réactiver au moins un processus. Il s'agit d'un état stable que l'on peut spécifier sous la forme

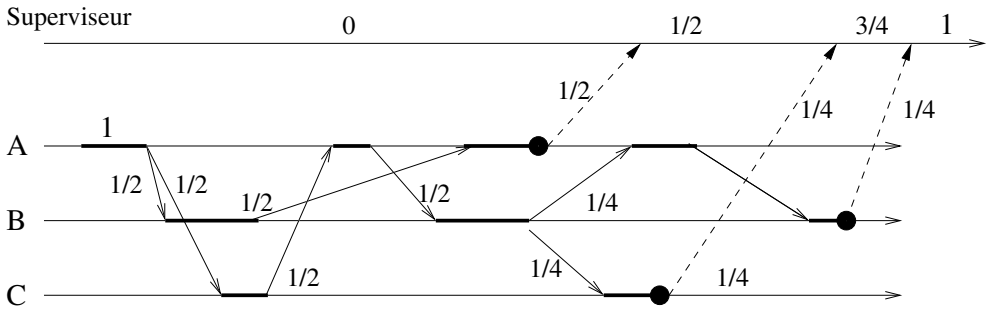


FIGURE 4.4 – Détection par la méthode du crédit

d'une propriété de sûreté :

$$\texttt{stable} \quad \forall\, i \;::\; P_i.passif \wedge EnTransit = \emptyset \tag{4.1}$$

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

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

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

– Sûreté : pas de fausse détection :

$$Term \Rightarrow \; (\forall\, i \;::\; P_i.passif \wedge EnTransit = \emptyset)$$

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

$$(\forall\, i \;::\; P_i.passif \wedge EnTransit = \emptyset) \mapsto \; Term$$

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

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

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

Le processus superviseur reçoit donc au cours du déroulement du calcul diffusant, les parcelles du crédit initial transmis par les processus ne propageant pas le calcul. Lorsque la somme de ces crédits atteint le crédit initial (ici 1), le calcul est terminé. La figure 4.4 montre un exemple avec un crédit initial de valeur 1.

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

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

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

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

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

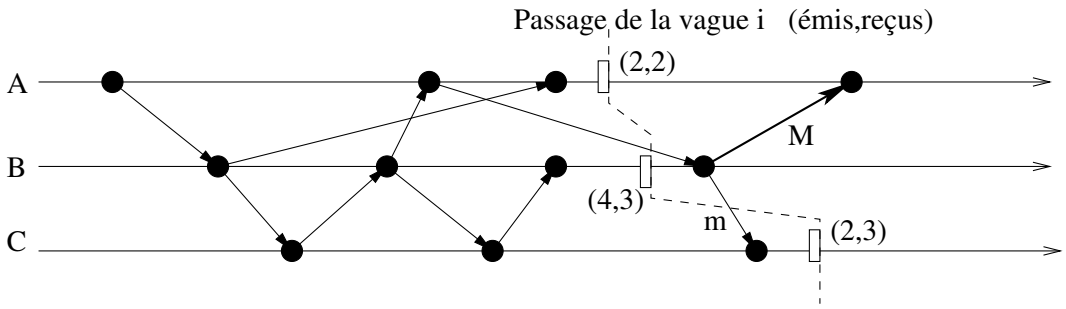


FIGURE 4.5 – Un exemple de fausse détection

4.2.4 Algorithme fondé sur la gestion de compteurs (Mattern)

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

$$E(t) = R(t) \Rightarrow EnTransit = \emptyset$$

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

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

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

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

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

Malheureusement, la figure (4.5) montre que ce n'est pas le cas. Le passage de la vague aux instants précisés sur le chronogramme conduit à une fausse détection. L'égalité entre les deux sommes est vraie, mais le calcul n'est pas terminé car un message en transit existe encore (M). L'anomalie tient à la prise en compte d'un événement de réception d'un message (m) dont l'événement d'émission n'a pas été comptabilisé. Nous reviendrons sur ce genre d'anomalie avec la notion de coupure cohérente.

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

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

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

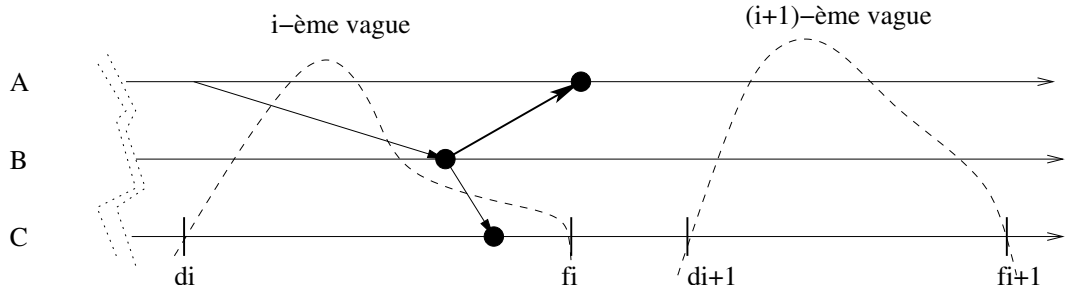


FIGURE 4.6 – Vagues séquentielles

La preuve de la validité du prédicat peut s'appuyer sur les instants de début d_i, d_{i+1} et de fin f_i, f_{i+1} des vagues correspondantes, ces instants étant ordonnés dans le temps : $d_i < f_i < d_{i+1} < f_{i+1}$ comme sur la figure (4.6).

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

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

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

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

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

4.2.5 Algorithme fondé sur la notion de chemin réparti

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

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

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

Ainsi, un calcul diffusant apparaît plutôt sous la forme d'un calcul arborescent comme la figure (4.7) en donne une idée.

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

Quelques remarques importantes avant d'aborder l'algorithme :

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

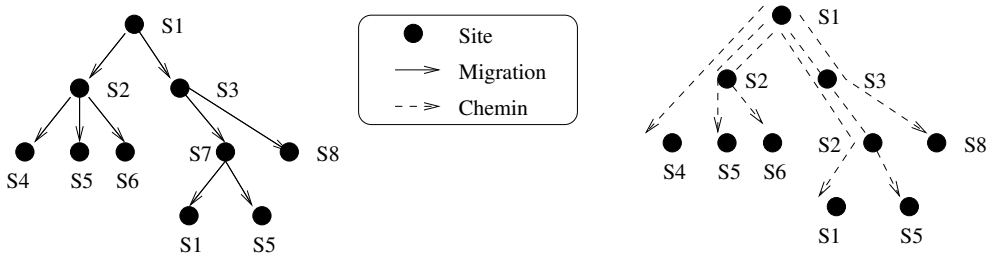


FIGURE 4.7 – Une vision par chemins du calcul

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

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

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

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

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

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

La modélisation du calcul par chemins

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

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

1. **Follow** : l'action locale traite le message entrant et en réponse poursuit le calcul en envoyant un unique message vers un autre processus : dans ce cas, on peut considérer que le chemin entrant se poursuit ;
2. **End** : l'action locale traite le message entrant et n'envoie aucun message. Ce cas correspond à la fin d'un chemin ;
3. **Split** : l'action locale traite le message entrant et envoie p messages ($p > 1$) vers d'autres processus [2]. Ce cas correspond à la création de $p-1$ nouveaux chemins. L'algorithme proposé impose que l'envoi de ces p messages soit vu comme atomique c'est-à-dire que le protocole de terminaison doit savoir combien

2. éventuellement, le processus peut s'envoyer un message à lui-même, ce qui peut être interprété comme une continuation du même processus.

de messages seront envoyés avant l'envoi du premier de ces p messages. Ceci peut être assuré en gardant les messages dans un tampon associé à l'action locale émettrice tant que celle-ci n'a pas explicitement dit qu'elle se terminait. Dans le système Chorus, cette approche était adoptée pour assurer l'atomicité des étapes d'un acteur [32].

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

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

Mise en œuvre de l'algorithme

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

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

Algorithme des processus participants

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

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

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

Algorithme du processus collecteur

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

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

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

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

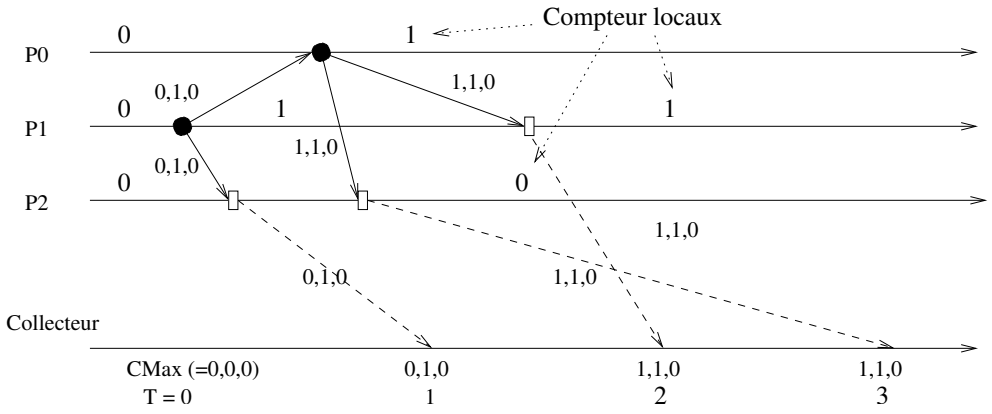


FIGURE 4.8 – Exemple de détection

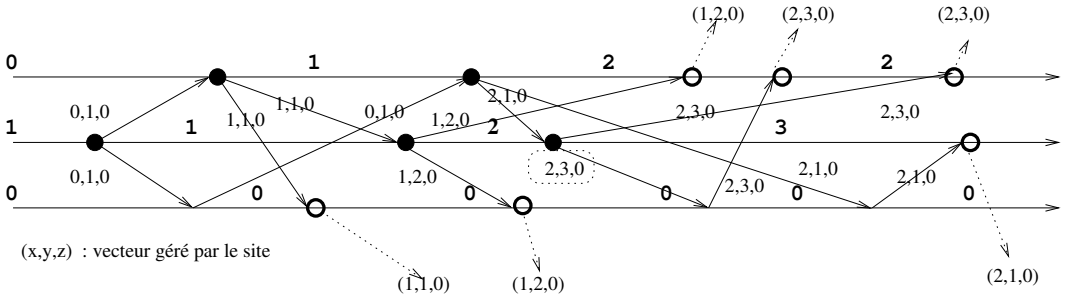


FIGURE 4.9 – Exemple plus complet

- enregistre la terminaison d'un chemin (incrémentation de T)
- et teste le prédicat de terminaison en comparant la somme des éléments du vecteur MT [3] avec le nombre de chemins terminés indiqué par la variable T.

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

Le chronogramme de la figure (4.8) montre l'évolution du calcul sur un exemple simple.

L'exemple de la figure (4.9) présente un autre cas plus complet montrant l'importance du compteur local à chaque processus. Le processus collecteur n'est pas représenté. On notera que les messages envoyés vers le collecteur peuvent être reçus dans un ordre quelconque.

4.3 Interblocage

Le problème de l'interblocage ne disparaît naturellement pas avec la répartition. Tout schéma d'allocation de ressources critiques réparties conduit à un système parallèle sujet au risque d'interblocage. Pire, la

3. On utilise la notation $\Sigma\ X$ pour abréger : $\Sigma\ X =< + : 0 \leq k < N\ ::\ X[k] >$.

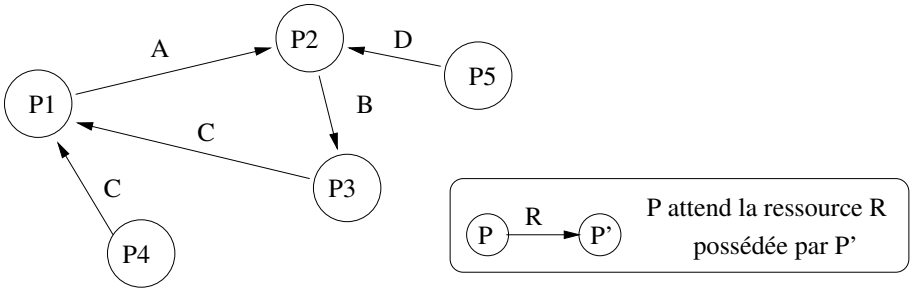


FIGURE 4.10 – Cycle d'interblocage dans le modèle **et**

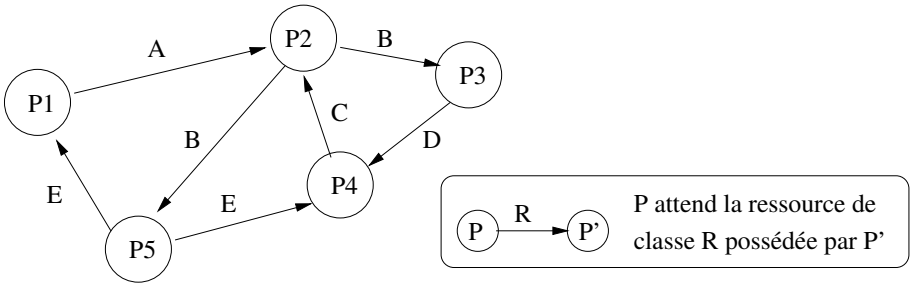


FIGURE 4.11 – Graphe CFCT d'interblocage dans le modèle **ou**

communication elle-même peut être à l'origine de situations d'interblocage sous certaines conditions.

4.3.1 Rappel

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

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

Dans le modèle de type *et*, un état d'interblocage est caractérisé par la détection d'un cycle dans le graphe d'attente des processus. La figure (4.10) montre un tel graphe dans lequel les processus P_1, P_2 et P_3 sont en état d'interblocage.

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

$$\forall s \in S \ : \ suc(s) \neq \emptyset \wedge suc(s) \subseteq S$$

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

La figure (4.11) illustre un tel graphe entre 5 processus. Le processus P_2 (resp.P_5) attend une ressource de classe B (resp. E) sachant que les exemplaires de cette classe de ressources sont possédés par P_3 et P_5 (resp. P_1 et P_4).

4.3.2 Interblocage des communications

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

Alors, le ou les processus émetteurs possibles apparaissent comme des ressources critiques demandées par les récepteurs. Dans la figure (4.11), on peut donner ainsi à chaque arc la signification suivante : un arc existe de P vers P' si P attend un message issu de P'. On remarquera que cette interprétation implique de connaître le ou les émetteurs qui peuvent envoyer un message vers le récepteur [4].

Compte tenu des arcs existants, les processus de la figure (4.11) sont tous dans l'état « en attente de réception ». à titre d'exemple, le processus P_5 peut être considéré comme en attente d'un message issu du processus P_1 ou P_4.

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

4.3.3 Un algorithme de détection (Chandy et Misra)

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

Modélisation des processus applicatifs

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

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

Déroulement du calcul diffusant

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

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

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

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

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

Autrement dit, la détection de la CFCT est équivalente à la réussite de la construction de l'arbre ou, ce qui est équivalent, à la terminaison du calcul diffusant qui le construit). Si l'arbre se construit ($\equiv$ le calcul diffusant se termine), il existe une CFCT. Si la construction de l'arbre ne se termine pas ($\equiv$ le calcul diffusant ne se termine pas), il n'y a pas interblocage incluant le processus enquêteur.

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

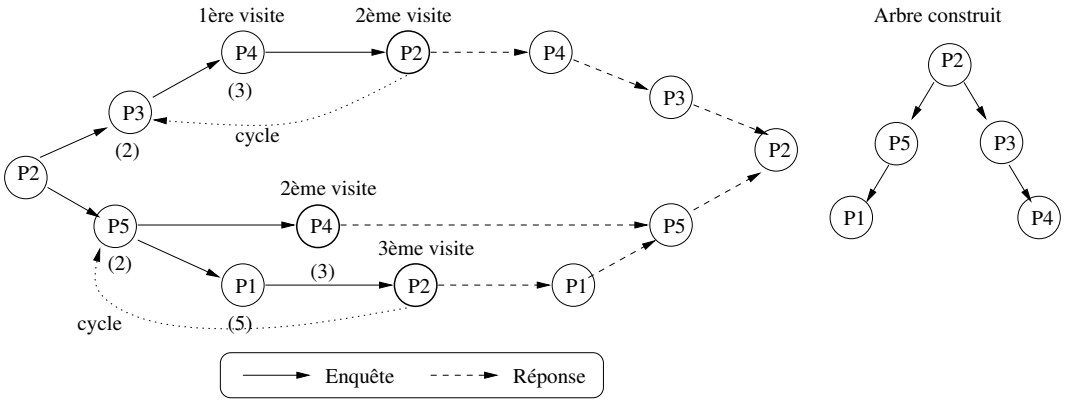


FIGURE 4.12 – Calcul diffusant de détection d'une CFCT

à titre d'exemple, la figure (4.12) montre le déroulement du calcul diffusant débuté par P_2 et conduisant à la conclusion que ce dernier appartient bien à une CFCT englobant les 5 processus. La détection de la CFCT repose ici sur la détection de 2 cycles.

4.4 Exercices

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

Deuxième partie

Systèmes et services répartis

Chapitre 5

Les services de base

5.1 Calcul d'un état global : prise de cliché

5.1.1 Introduction

L'absence d'état global est une des caractéristiques essentielle (et malheureuse...) des systèmes répartis. Un site ou processus n'a qu'une connaissance approximative de l'état des autres sites ou processus puisque, seul, l'échange de messages permet d'obtenir des informations sur les partenaires distants (logiquement).

L'intérêt de capter un cliché global d'un calcul réparti est avant tout de pouvoir, en cas de défaillance, reprendre le calcul à partir d'un tel cliché. Il s'agit donc de définir des points de reprise potentiels.

Un grand nombre d'algorithmes ont donc été proposés sur le thème de l'évaluation d'états globaux cohérents d'un calcul réparti. L'objectif de ces algorithmes est d'arriver à collecter un ensemble d'états locaux aux processus afin d'obtenir un état global passé cohérent du calcul. Cette prise de cliché (snapshot) pose surtout un problème de cohérence des informations collectées incluant d'éventuels messages en transit.

La figure (5.1) montre cette difficulté : le processus collecteur capte un état local E_A du processus A pour lequel le message m n'a pas encore été envoyé, alors que l'état local E_B du processus B est postérieur à la réception de m. C'est ce genre d'incohérence qui peut par exemple, dans un algorithme de détection d'un état stable, provoquer une fausse détection. Dans cette exemple, s'il s'agit de détecter la terminaison d'un calcul diffusant par comptage des messages envoyés et reçus par chaque processus, l'état global incohérent collecté peut conduire à une fausse détection du fait de la prise en compte de la réception du message m sans avoir pris en compte son émission.

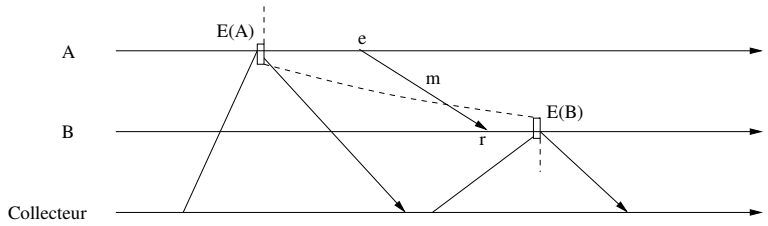


FIGURE 5.1 – Prise de cliché incohérent

Une propriété de base à préciser est donc la notion de cohérence d'un état global. Pour ce faire, on utilise la notion de coupure cohérente.

5.1.2 Notion de coupure cohérente

De façon standard, le calcul réparti est abstrait sous la forme d'un ensemble d'événements partiellement ordonnés. On appelle coupure cohérente C un sous-ensemble des événements $\mathcal{CR}$ d'un calcul réparti vérifiant

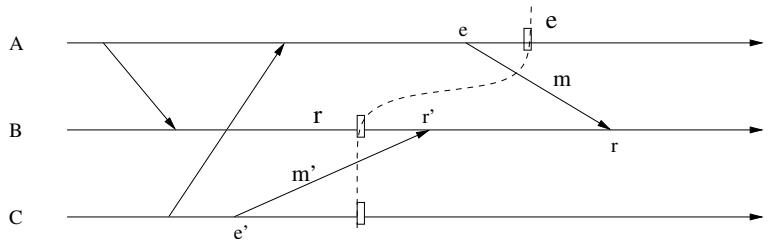


FIGURE 5.2 – Prise de cliché cohérente mais d'un état potentiel

la propriété suivante :

$$\forall e \in C \ : \ \forall e' \in \mathcal{CR} \ : \ e' \prec e \Rightarrow e' \in C$$

Autrement dit, si un événement e appartient à une coupure, alors tous les événements du calcul qui le précèdent causalement appartiennent aussi à la coupure. Une coupure, comme son nom le suggère, établit une frontière sur tous les sites entre un ensemble d'événements qui sont « avant » et la suite du calcul « après ». L'exemple de la figure (5.1) montre une coupure incohérente puisque l'événement d'émission e de m n'est pas dans l'ensemble des événements collectés qui contient, par contre, l'événement r.

Un algorithme de prise de cliché devra donc tout d'abord réaliser une coupure cohérente du calcul. Cependant, l'état global construit n'aura pas forcément existé dans le temps global. à titre d'exemple, la figure (5.2) montre une coupure cohérente bien qu'elle suppose un état global dans lequel l'événement e est arrivé et l'événement r' est encore à venir alors que dans la réalité l'événement r' a eu lieu avant e. Ceci n'a cependant pas d'importance puisque les deux événements ne sont pas causalement liés.

Par ailleurs, l'événement de réception du message m n'est pas capté dans la prise de l'état local du site B, alors que l'émission de ce message appartient au passé de l'état local capté du site A. Pour obtenir un état cohérent, il faudra donc aussi mémoriser que la message m est un message en transit. Un état global cohérent associé à la coupure de la figure (5.2) devra donc aussi préciser que le message m est en transit.

5.1.3 Un algorithme de prise de cliché (Chandy et Lamport)

Cet algorithme assure la collecte d'un cliché cohérent constitué des états locaux de chaque site et des messages en transit associés. Pour ce faire, certaines hypothèses simplificatrices sont faites :

- la communication est supposée point à point sur des canaux FIFO unidirectionnels. Chaque site i connaît d'une part ses canaux entrants E_i et d'autre part ses canaux sortants S_i ;
- la topologie du réseau de communication doit être fortement connexe ; cette contrainte est due au principe de marquage utilisé.

Comme il n'est pas possible de synchroniser la prise de chaque état local à un instant t, la prise d'un cliché global est asynchrone. Chaque processus peut décider de débuter une phase d'évaluation d'un état local complété par d'éventuels messages en transit.

Phase d'évaluation d'un cliché local

Cette phase consiste à mémoriser son état local et à capter, à partir de cet instant, les messages en transit sur ses canaux entrants. Pour cela, une fois cet état local mémorisé, un message « marqueur » est émis sur chaque canal de sortie S_i. Après quoi, tout message entrant est mémorisé comme message en transit appartenant à un canal donné. Si un message marqueur arrive parmi ces messages entrant, alors tous les messages en transit sur ce canal ont été pris en compte. Lorsqu'un message marqueur aura été capté sur chaque canal entrant, l'état local mémorisé complété par les messages en transit de chaque canal entrant peut être envoyé au processus collecteur.

Lorsqu'un processus reçoit un message marqueur, deux cas sont possibles :

- soit il a déjà mémorisé son état local, auquel cas, il est en phase de réception des messages marqueurs et il n'a donc qu'à enregistrer que la collecte des messages en transit sur un canal entrant est terminée ;

– soit il n'a pas encore mémorisé son état local, auquel cas, il débute la phase d'évaluation d'un cliché local.

5.2 Le problème du consensus

Le problème du consensus est considéré par certains comme LE problème fondamental posé aux systèmes répartis pour les rendre tolérants aux fautes sous les hypothèses asynchrones. Il est en effet sous-jacent à toute situation où l'on utilise la redondance pour obtenir un système tolérant certaines fautes.

Par exemple, dans un système de gestion de transactions réparties, tous les processus ayant participés à une transaction doivent finalement décider de sa validation ou de son annulation. Ils doivent TOUS prendre la même décision. Dans les protocoles de diffusion, il faut pouvoir décider si un message a bien été reçu par tous les processus cibles. Chacun d'entre eux devra là encore prendre la même décision vis-à-vis d'un message donné. Lorsqu'on met en œuvre de la réplication, il faudra que chaque processus répliquant un calcul se mette d'accord avec les autres sur le résultat.

Le problème majeur du consensus est de tolérer les défaillances de sites ou de communication. Il faudra en particulier distinguer les processus corrects des processus défaillants. La défaillance même d'un processus pourra prendre plusieurs formes. Une forme simple est l'arrêt. Mais des comportements plus complexes pourront être considérés comme par exemple, au pire, un comportement arbitraire : un processus émet par exemple des messages erronés (voir problème des généraux byzantins). Pourtant, le problème est simple à spécifier : comment N sites (processus) peuvent se mettre d'accord, donc atteindre un consensus, en communiquant par messages ?

Sous ces apparences simples, le consensus se révèle un problème délicat qui ne doit pas être réduit à un simple problème d'acquittement de message. Il suffit pour s'en convaincre d'un exemple comportant seulement deux interlocuteurs.

Roméo et Juliette se donnent rendez-vous par e-mail

Supposons que Roméo souhaite donner rendez-vous à Juliette à une date et en un lieu fixé en utilisant l'e-mail et supposons que le service e-mail puisse perdre des messages.

Roméo envoie donc un premier message pour proposer un rendez-vous à Juliette. Tout ce passe bien et celle-ci le reçoit 5 mns plus tard. Elle répond oui par un message qui parvient 10mns plus tard à Roméo. Le consensus semble alors atteint entre Roméo et Juliette et ils devraient donc se rencontrer. Cependant, Roméo est alors pris d'un doute : Si Juliette ne sait pas que j'ai reçu sa réponse positive, elle peut supposer que son message réponse s'est perdu et peut-être n'ira-t-elle pas au rendez-vous. Je devrais donc lui envoyer un message confirmant que j'ai bien reçu sa réponse. Mais dans ce cas, une suite infernale d'acquittements réciproques débute. En fait, sous ces hypothèses, il n'existe alors pas de protocole qui permette d'être sûr que les deux prendront la même décision.

Le scénario précédent amène les remarques suivantes :

– Même pour seulement deux processus, le problème est insoluble ;
– Le problème du consensus est strictement distinct de celui de l'acquittement d'un message. Roméo reçoit bien un acquittement de son premier message lorsque lui parvient la réponse de Juliette. Il sait alors que son premier message est bien parvenu à Juliette. Par contre, Juliette ne sait pas qu'il possède cette connaissance. Elle peut avoir un doute ! Le problème du consensus soulève donc un problème plus difficile de coordination mutuelle ;
– Si Roméo et Juliette avaient pris un téléphone, ils seraient parvenus à un consensus sans difficulté. En effet, la communication assure alors une borne supérieure à la délivrance ou à la perte d'un message (coupure de la ligne). C'est l'hypothèse d'un délai non borné de délivrance des messages qui rend le consensus impossible ;
– Une approche probabiliste peut être adoptée : Roméo et Juliette auraient pu dupliquer leurs messages de façon à minimiser le risque de perte d'un message. Il est alors intéressant de concevoir des algorithmes de consensus de type probabiliste garantissant qu'un consensus sera atteint dans un délai **borné** avec

une **forte** probabilité. Ceci est possible car les situations rendant le protocole impossible à 100% sont rares.

5.2.1 Spécification du problème

Plus précisément, une modélisation du problème peut être faite sous la forme des hypothèses de base suivantes :

- le système est composé de N processus ;
- chaque processus possède une valeur initiale $v_0 \in \mathcal{D}$;

Un algorithme correct doit vérifier les spécifications suivantes :

1. Terminaison : l'algorithme doit être vivace c'est-à-dire que tout processus correct doit finalement décider d'une valeur finale ;
2. Validité : si tous les processus ont initialement la même valeur v_0, tous les processus corrects choisiront finalement pour valeur finale v_0 ;
3. Cohérence : si un processus correct choisit une valeur v, tous les autres processus choisiront aussi v.

Des variantes existent selon les contraintes de validité de la valeur finale : par exemple, la valeur finale doit être la valeur initiale possédée par une majorité de sites, voire par tous les sites. À titre d'exemple, on peut supposer que les processus doivent se mettre d'accord sur une valeur booléenne ($\mathcal{D}$ = Booléen). La contrainte de validité peut être alors pour la validation d'une transaction :

- FAUX n'engendre pas VRAI : $(\forall i \ : \ \neg P_i.v_0) \Rightarrow (\forall i \ : \ \neg P_i.v_f)$
- VRAI n'engendre pas FAUX : $(\forall i \ : \ P_i.v_0) \Rightarrow (\forall i \ : \ P_i.v_f)$
- Une seule valeur initiale fausse entraine l'invalidation :

$$(\exists i \ : \ \neg P_i.v_0) \Rightarrow (\forall i \ : \ \neg P_i.v_f)$$

5.2.2 Les différents contextes de résolution

Le problème du consensus peut être envisagé dans différents contextes de communication :

- le support des échanges d'information entre processus peut être aussi bien une mémoire partagée accessible aux différents processus pour communiquer (architecture multiprocesseurs dotée d'une zone mémoire globale) aussi bien qu'un simple réseau de communication par messages (architecture distribuée) pour communiquer entre processus,
- système synchrone ou système asynchrone : les processus s'exécutent en synchronisme réel ou sont indépendants les uns des autres.

La difficulté survient dès que l'on se place dans un contexte où tous les composants du système ne sont pas fiables et peuvent donc provoquer des fautes. Les défaillances peuvent se situer au niveau des communications (perte de message par exemple) ou au niveau des processus (processeurs). Pour ces derniers, il faudra distinguer d'une part l'arrêt pur et simple d'un processus et d'autre part, le passage à un comportement arbitraire (comportement byzantin).

Un autre paramètre important est, dans le cas d'une communication par messages, l'existence ou non d'une borne connue sur le délai de transfert d'un message. L'existence d'un délai maximum connu permet de retrouver des hypothèses proches de celle d'un système synchrone au sens strict.

Un résultat important et « symbolique » est qu'il n'existe pas d'algorithme résolvant le consensus dans le cas d'un système asynchrone dès qu'un seul processus peut être défaillant (par arrêt).

5.2.3 Preuve de l'impossibilité du consensus en asynchrone

Il est important de bien préciser les hypothèses :

- Chaque processus évolue à son rythme (système asynchrone) ;
- Les processus communiquent par un protocole point-à-point [1] ;

1. Même si le protocole est un protcole de diffusion, il n'existe pas de solution si ce protocole n'est pas ordonné

– Il n'y a pas de délai maximal connu pour la transmission des messages ;

La preuve, due à M. Fisher, N. Lynch et M. Paterson [12], repose sur la mise en évidence d'une séquence d'exécution empêchant tout consensus en présence d'une défaillance d'un **seul** processeur. Plus exactement, l'arrêt d'un processus peut entraîner l'impossibilité de conclure.

De façon standard, l'exécution du protocole peut être abstraite sous la forme des événements engendrés par les messages et en particulier on s'intéresse aux réceptions des messages. On note $E(t)$ l'état courant du système à l'instant t composé, classiquement, d'une part par les états locaux des processus et d'autre part, par les messages en transit.

Définition 5.2.1 *Un état $E(t)$ est dit déterministe s'il conduit toujours à la **même** valeur finale consensuelle. Dans le cas contraire, il est dit bivalent.*

La démonstration repose sur les deux lemmes suivants :

Lemme 5.2.1 *Il existe un état initial bivalent.*

Lemme 5.2.2 *Partant d'un état bivalent, on peut trouver (il existe potentiellement) une séquence non vide de transitions vers un autre état bivalent.*

L'existence d'un état initial bivalent (lemme 5.2.1) et l'existence d'un « chemin infini » d'état bivalent à état bivalent amène à conclure à l'inexistence du protocole.

Pour simplifier, on suppose que les processus doivent se mettre d'accord sur une valeur booléenne. Initialement, ils possèdent donc chacun une valeur V ou F. Les processus devront se mettre d'accord sur l'une de ces deux valeurs.

Preuve du lemme 5.2.1

Si l'on considère les états initiaux possibles, on peut constater qu'il est possible de les ordonner de telle façon que 2 états adjacents ne diffèrent que par une seule valeur correspondant à un processus fixé.

	P_1	P_2	...	P_{N-1}	P_N	Résultat
E_1	V	V	...	V	V	V
E_2	F	V	...	V	V	?
E_3	F	F	...	V	V	?
			...			
E_N	F	F	...	F	V	?
E_{N+1}	F	F	...	F	F	F

Supposons que tous les états initiaux soient déterministes (≡ pas d'état initial bivalent). Il existe alors forcément parmi ces états E_i, deux états E_v et E_f l'un conduisant à la valeur consensuelle V et l'autre à la valeur F mais ne différant que par un seul composant :

$$\forall\ i \neq i_0\ :\ E_v[i] = E_f[i]\ \wedge\ E_v[i_0] = \neg E_f[i_0]$$

Ce qui revient à dire qu'il faut donc qu'il existe une fonction *Consensus* prenant en paramètre les états et telle que :

$$Consensus(E_v) = \neg Consensus(E_f)$$

Or, si le processus i_0 est à l'arrêt dès le début du protocole (ou avant d'avoir pu communiquer quoi que se soit aux autres processus) les deux états précédents deviennent :

$$\forall\ i \neq i_0\ :\ E'_v[i] = E'_f[i]\ \wedge\ E'_v[i_0] = E'_f[i_0] = \perp \equiv\ E'_v = E'_f$$

Et par conséquent, on aura $Consensus(E'_v) = Consensus(E'_f)$. Les deux configurations E_v et E_f conduisent donc alors au même consensus contrairement à l'hypothèse faite.

Il existe donc bien des états initiaux bivalents.

Preuve du lemme 5.2.2

Partant d'un état bivalent $E(t)$, il existerait un algorithme si aucune séquence de transition vers un autre état bivalent ne pouvait être trouvée. Supposons donc que partant un état bivalent, il existe un événement de réception r_v d'un message provoquant le passage dans un état déterministe conduisant à V, et il existe donc aussi un événement de réception r_f d'un message provoquant le passage dans un état déterministe conduisant à F. L'hypothèse d'un état bivalent impose l'existence de ces deux événements.

Cas 1 : les deux événements r_v et r_f ont lieu dans des processus récepteurs distincts L'occurrence de ces deux événements peut avoir lieu dans n'importe quel ordre et donne le même état résultat. Or, selon l'ordre d'occurrence, l'état résultat serait V ou F. Un même état ne peut donner un résultat différent.

Cas 2 : les deux événements r_v et r_f ont lieu dans le même processus Dans ce cas, un premier événement peut avoir lieu et le processus peut s'arrêter. Alors, l'état atteint ainsi est le même quel que soit l'événement arrivé avant la panne. Or, l'état avant la panne avait conduit à décider V si l'événements était r_v et à décider F si l'événement était r_f. On voit donc qu'à nouveau une contradiction existe.

L'hypothèse initiale supposant qu'il n'existait pas de séquence possible de transitions vers un autre état bivalent était donc fausse.

5.2.4 Un algorithme simple

Pour que tous les processus prennent la même décision, évaluent le même résultat final, il suffit qu'ils aient connaissance des toutes les valeurs initiales et qu'ils utilisent une même fonction pour évaluer la valeur finale :

$$\forall i \ : \ P_i.v_f = F(P_0.v_0, \ldots, P_{N-1}.v_0)$$

Initialement, un processus i ne connaît que $P_i.v_0$ et il ne peut donc évaluer F.

Si les communications sont fiables, un algorithme résolvant le problème peut être très simplement trouvé. Chaque processus diffuse sa valeur initiale à tous les autres processus. Il recevra donc lui-même la valeur initiale de tous les autres processus. Une fois en possession de toutes ces valeurs, chaque processus pourra prendre une décision à partir des **MÊMES** données. Par conséquent, la fonction F donnera le même résultat quel que soit le processus.

5.2.5 Quelques problèmes voisins

Le problème du consensus comporte de nombreuses variantes applicatives. Parmi elles, on peut en distinguer deux classiques :

- Un processus « Maître » m propose une valeur V_m qu'il diffuse aux autres et finalement, tout processus correct choisit la même valeur qui peut être soit la valeur proposée V_m si le processus « maître » était correct, soit une valeur par défaut si celui-ci était incorrect. Ce problème est plus connu sous le terme de problème des généraux byzantins. Il a été largement étudié dans le domaine de la tolérance aux fautes.
- Chaque processus propose une valeur v_i et les processus corrects doivent finalement construire un vecteur Vd_i identique dans lequel $Vd_i[j] = v_j$ pour tout processus i correct. C'est de cette façon que des processus peuvent se mettre d'accord sur un état global par exemple.

5.3 Conclusion

L'absence d'algorithme pour obtenir un consensus dans le cas d'un système asynchrone n'empêche pas de trouver des solutions dès que l'on change quelque peu les hypothèses sur les propriétés de la communication. En effet, il suffit, par exemple, d'avoir un délai borné connu pour la transmission des messages, pour obtenir un cadre où des algorithmes existent. L'existence d'hypothèses minimales a même été démontrée. Il faut

introduire la possibilité pour chaque participant d'avoir une certaine connaissance de l'état des autres. Cette connaissance peut être cependant imparfaite. La modélisation consiste à introduire le concept de détecteur de défaillance (failure détector)[9]. Un tel détecteur est associé à chaque processus participant et est capable de détecter la défaillance (l'arrêt) des autres processus. Les propriétés garanties par les détecteurs de défaillance peuvent prendre diverses formes.

À titre d'exemple, les plus faibles propriétés que doivent garantir les détecteurs sont dénotées par $\diamond\mathcal{W}$:

Condition faible $\diamond\mathcal{W}$

- Complétude faible : finalement tout processus défaillant est suspecté de façon permanente par au moins un processus correct ;
- Exactitude faible : il existe une date à partir de laquelle au moins un processus correct n'est pas suspecté par tous les autres processus corrects.

Un autre type de détecteur, dénoté $\diamond\mathcal{S}$, garantit une complétude plus forte sur la détection des défaillances :

Condition forte $\diamond\mathcal{S}$

- Complétude forte : finalement tout processus défaillant est suspecté de façon permanente par tous les processus corrects ;
- Exactitude faible : il existe une date à partir de laquelle au moins un processus correct n'est pas suspecté par tous les autres processus corrects.

L'existence de ces détecteurs vérifiant les conditions $\diamond\mathcal{W}$ ou $\diamond\mathcal{S}$ permet de développer des algorithmes corrects apportant des solutions au problème du consensus dans un contexte restant fondamentalement asynchrone et non fiable [30].

Enfin, le problème du consensus est équivalent au problème d'une diffusion vers un groupe (multicast) fiable totalement ordonnée. En particulier, si l'on dispose d'un tel protocole de diffusion, le problème du consensus peut être simplement résolu. Il suffit que chaque processus diffuse sa valeur au groupe. La première valeur reçue par chaque processus est la même et peut donc constituer la valeur choisie par tous.

Chapitre 6

Conclusion

Les principes fondamentaux de la répartition sont maintenant bien établis. Le modèle standard événementiel d'un calcul réparti permet de capter un tel calcul à un niveau d'abstraction suffisant pour observer et contrôler une exécution répartie. La relation de causalité apporte une formalisation de l'ordre partiel existant entre les événements.

Une algorithmique dites « répartie » s'est développée pour classifier et spécifier les problèmes génériques de la répartition. Cet effort de recherche se poursuit aujourd'hui compte tenu de l'évolution des architectures réparties. En effet, la plupart de ces algorithmes étaient proposés dans un contexte très statique : nombre fixe de processus, communication figée entre ces processus et hypothèses de fiabilité simples. Aujourd'hui, avec les réseaux mobiles en particulier ad hoc, ces hypothèses ne tiennent plus. Il faut envisager des algorithmes fondés sur hypothèses plus « souples » en terme de communication et de nombre de processus participants. Par ailleurs, le nombre même de participants a tendance à augmenter de façon très importante. Là encore, ce facteur d'échelle peut conduire à des remises en cause de l'existant. Les réseaux de capteurs sont un exemple de cette évolution.

Si le schéma client-serveur reste d'actualité, d'autres formes de communication prennent aussi de l'importance, en particulier dans les jeux en réseaux via Internet ou dans des applications métiers plus spécifiques : robotique et/ou simulation avec des contraintes de temps réel par exemple.

La preuve des algorithmes répartie reste aussi un défi difficile à atteindre. Leur complexité est très élevée et les méthodes de preuve par vérification de modèle (model-checking) se heurtent encore à l'explosion combinatoire du nombre d'états possibles de tels calculs. L'usage de plus en plus fréquent d'architectures matérielles ayant des propriétés réparties dans les systèmes critiques (avionique, espace par exemple), nécessite donc un effort important de recherche de méthodes de développement sûr de tels systèmes répartis.

Mais, sans aucun doute, c'est l'essor du *Cloud computing* qui apporte un renouveau intéressant dans le domaine de la répartition. En effet, les systèmes à mettre en place pour cette approche de l'informatique répartie à large échelle posent de nouveaux défis aux architectes « systèmes ». La mise à disposition de ressources logicielles ou matérielles à distance en assurant les niveaux de transparence les plus hauts (transparence de charge et d'échelle par exemple) est l'objectif affiché du *Cloud computing*. De nombreux projets sont en cours pour optimiser de tels systèmes et pour les rendre les plus adaptables et autonomes possibles.

Les systèmes répartis ont donc encore de beaux jours devant eux ...

Appendices

Annexe A

Quelques sujets d'examens avec ou sans correction...

A.1 Causalité, Horloge et Prise de cliché

A.1.1 Causalité (4 points)

On considère le chronogramme suivant :

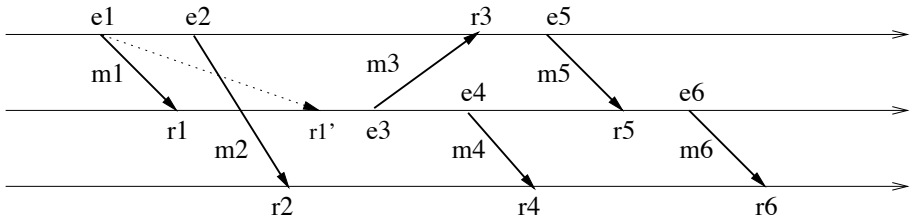


FIGURE A.1 – Chronogramme 1 : Causalité

Questions

1. Précisez si les événements r_4 et r_5 sont causalement liés ou pas. Justifiez votre réponse.
2. Montrez que la réception du message m_1 à la date réelle r'_1 au lieu de r_1 ne change pas les relations de causalité qui existaient. Autrement dit, les relations de causalité directe entre l'événement r_1 et les autres événements sont les mêmes que celles existant entre r'_1 et ces mêmes événements.

A.1.2 Une horloge minimaliste (8 points)

On considère un schéma de datation minimaliste utilisant simplement un compteur croissant sur chaque site. Chaque site possède une horloge locale H_s gérant une date logique sur laquelle les opérations d'incrémentation *Top* et de recalage *Recaler* peuvent être exécutées :

```
class Horloge {
   int cpt = 0 ;
   int Top()  { cpt++; return cpt ; }  // Incrémentation-Lecture
   void Recaler( int d ) { if (cpt < d )  cpt = d ; }  // Recalage
}
```

L'opération Top() incrémente sa valeur et renvoie la nouvelle valeur de l'horloge. L'opération Recaler(d) affecte la valeur fournie en paramètre à l'horloge si et seulement si celle-ci était en retard.

Enfin, les actions de mise à jour de ces horloges lors les différents types d'événements sont les suivantes, où d_e (respectivement et d_r) dénote la date de l'événement d'émission (respectivement la date de l'événement de réception) d'un message m :

Type d'événement sur un site s	Action mettant en jeu l'horloge du site s
Événement interne sur s	$H_s.Top()$
Émission sur s de m	int $d_e = H_s.Top()$; envoi de $< d_e, m >$
Réception sur s de $< d_e, m >$	$H_s.Recaler(d_e); d_r = H_s.Top()$

FIGURE A.2 – Tableau des actions de mise-à-jour des horloges

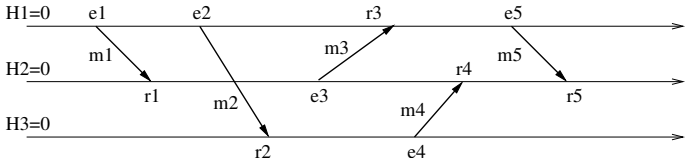


FIGURE A.3 – Chronogramme 2 : Datation

Questions

3. Décorez le chronogramme de la figure A.3 en précisant la date de chacun des événements affectés par le mécanisme d'horloge proposé. (Note : vous pouvez utiliser le chronogramme de la figure A.5 en fin de sujet comme support de la réponse.)
4. Quelle propriété peut-on déduire des dates affectées aux événements distincts e et e' lorsque $d_e = d_{e'}$?
5. On appelle chemin causal $C(e_n)$ conduisant à un événement e_n, une suite d'événements $\{e_1, e_2, \ldots, e_n\}$ vérifiant la propriété suivante : $e_1 \prec e_2 \prec \ldots \prec e_n$.

   La longueur d'un chemin causal, notée $|C(e_n)|$ est égale au nombre d'événements de la suite. [1] Montrez que la valeur de la date associée à un événement quelconque e est égale à la longueur du (ou des) plus long(s) chemin(s) causal(causaux) conduisant à e. Autrement dit, l'invariant suivant est vérifié :

   $$\forall\ e\ :\ \exists\ C(e)\ :\ d_e = |C(e)|\ \wedge\ \forall\ C'(e)\ :\ |C'(e)| \leq |C(e)|$$

   Pour cela, montrez que si cette propriété est vraie avant l'événement, elle reste vraie après l'exécution de l'action correspondant au type d'événement considéré (voir tableau A.1.2).
6. Proposez une modification de l'opération de recalage qui permettrait d'obtenir l'invariant suivant : la date d'un événement est toujours supérieure au nombre total d'événements qui précèdent l'événement daté. Autrement dit, on aurait la propriété suivante :

   $$\forall e\ :\ d_e \geq Card(G_e)\ avec\ G_e = \{e' : e' \prec e\}$$

   Note : Proposez une solution simple qui ne fasse toujours intervenir que la date " incidente" en paramètre et la valeur courante de l'objet horloge " réceptrice" .

1. Par convention, on admet qu'un chemin causal $C(e)$ réduit à l'événement lui-même ($C(e) = \{e\}$) est donc de longueur 1.

A.1.3 Prise de cliché global(8 points)

On considère un ensemble de N processus P_i observés par un processus *Collecteur*. On suppose que les communications sont fiables et point à point mais non FIFO. On cherche un algorithme permettant au *Collecteur* de construire un cliché global cohérent en utilisant le mécanisme de vague et un schéma de coloriage des processus et messages applicatifs.

Rappel Un cliché global comprend d'une part, les états locaux de chaque processus et d'autre part, les messages en transit associés à la coupure " traçée" par les états locaux collectés. Une coupure est cohérente si et seulement si les messages en transit traversent cette coupure du passé vers le futur (sur un chronogramme, l'événement d'émission est à gauche de la coupure et l'événement de réception correspondant est à droite de celle-ci). Un cliché est cohérent ssi il constitue une coupure cohérente.

Règles de coloriage On utilise seulement deux couleurs : blanc et noir.
- initialement, tous les processus et messages sont blancs ;
- un processus récepteur devient noir s'il reçoit un message noir ;
- un processus émet toujours des messages de sa propre couleur ;
- un fois noir, tout processus reste noir ;

Le processus *Collecteur* lance une vague de couleur noire. Un processus visité devient donc noir dès le passage de la vague (s'il ne l'était pas déjà). Lorsqu'un processus **blanc** reçoit la visite de la vague, il enregistre un cliché local de son état et devient noir.

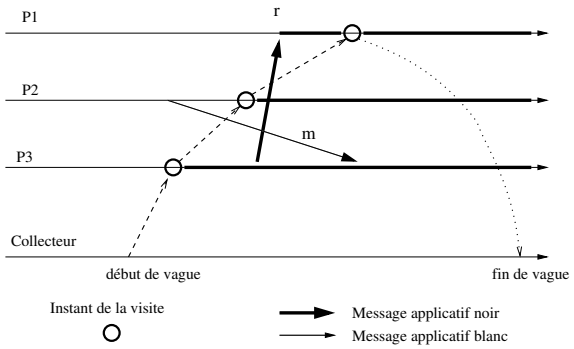


FIGURE A.4 – Prise de cliché par vague

Questions

7. Si un processus **noir** visité par la vague prend son cliché local au moment du passage de la vague (comme le fait un processus blanc), montrez que le cliché obtenu peut être incohérent (S'appuyer sur la figure A.4). En conséquence, pour obtenir un cliché cohérent, quelle mesure doit prendre un processus blanc lorsqu'il devient noir par réception d'un message applicatif noir (événement r dans la figure A.4) ?
8. Le *Collecteur* doit aussi obtenir les messages en transit tels que le message m dans la figure A.4. Caractérisez la classe d'événements de réception qui permettent de détecter les messages en transit (couleur du message reçu, couleur du processus récepteur).
9. On suppose que, à chaque événement de réception précédent (détecteur d'un message en transit), le processus récepteur transmet le message reçu au collecteur. Proposez un algorithme pour que le *Collecteur* puisse savoir qu'il a reçu tous les messages en transit associés au cliché (problème particulier de terminaison).
10. Proposer une modification de l'algorithme pour pouvoir faire plusieurs clichés successifs à partir d'un même collecteur.

Chronogramme pour la question 3

Affecter à chaque événement e_i ou r_i sa date.

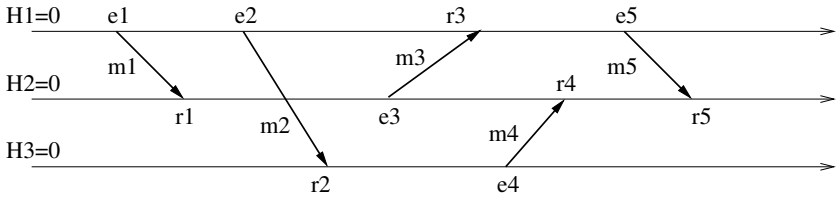


FIGURE A.5 – Chronogramme à décorer

A.2 Horloge et causalité directe (d'après Vilay K. Garg)

On considère un schéma de datation utilisant des horloges vectorielles. Une date est représentée par un vecteur de dimension N où N est le nombre de processus communicants. Chaque processus P_i possède une horloge vectorielle locale H_i dotée des opérations classiques d'incrémentation et de recalage. Initialement, l'horloge H_i du processus P_i a la valeur $H_i[i] = 1 \ \wedge \ \forall j \neq i \ :: \ H_i[j] = 0$. La classe Horloge suivante décrit l'algorithme des opérations Top et Recaler.

```
class Horloge {
   int[] cpt ; int loc = 0 ;
   // Lecture et incrémentation de l'horloge
   int[] Top()  { int[] hr = cpt.clone() ; cpt[loc]++ ; return hr ; }
   // Recalage de l'horloge
   void Recaler( int j, int vj ) { /* message issu du processus d'indice j */
      cpt[loc]  = Math.max(cpt[loc],vj + 1) ;
      cpt[j]    = Math.max(cpt[j],vj) ;
      }
   // Constructeur : "loc" localise le site de l'horloge
   Horloge(int où, int N) {
      loc = où; cpt = new int[N] ;
      for (int i = 0; i<N; i++) cpt[i]=0; cpt[loc] = 1 ;
      }
}
```

La sémantique de Recaler et Top sur une horloge H_i s'exprime sous la forme de triplets de Hoare par :
– L'opération Top renvoie la valeur courante de l'horloge et incrémente de 1 l'élément d'indice i.

$$\{H_i = h\} \ \ \mathsf{Top}() \ \ \{H_i[i] = h[i] + 1 \wedge \forall j \neq i :: \ H_i[j] = h[j]\}$$

– L'opération Recaler(j, vj) comporte deux paramètres d'entrée :
  – d'une part, le paramètre j représente l'indice du processus P_j émetteur du message conduisant à l'appel de l'opération Recaler ;
  – d'autre part, le paramètre vj représente la valeur de l'élément $h_e[j]$ datant l'événement d'émission e du message émis par P_j.

$$\{H_i = h\} \ \mathsf{Recaler}(j, vj) \ \{H_i[i] = Max(h[i], vj{+}1) \wedge H_i[j] = Max(h[j], vj) \wedge \forall k \neq i, j \ :: \ H_i[k] = h[k]\}$$

Enfin, les actions de mise à jour de ces horloges lors les différents types d'événements sont les suivantes, où h_e dénote la date de l'événement d'émission d'un message par P_i et h_r la date de l'événement de réception par P_i d'un message m émis par le processus P_j :

Type d'événement du processus P_i	Action mettant en jeu l'horloge H_i
Événement interne sur i	int[] $h = H_i.Top()$
Émission sur i de m	int[] $h_e = H_i.Top()$; envoi de $< i, h_e[i], m >$;
Réception sur i de $< j, vj, m >$ émis par P_j	$H_i.Recaler(j, vj)$; int[] $h_r = H_i.Top()$;

FIGURE A.6 – Tableau des actions de mise-à-jour de l'horloge vectorielle H_i d'un processus P_i

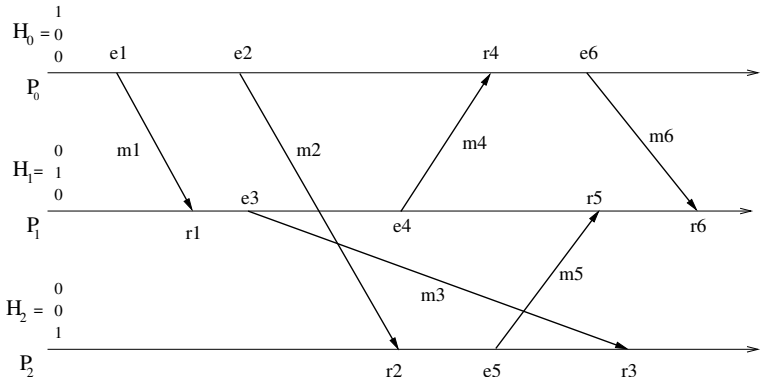


FIGURE A.7 – Chronogramme exemple

Notations

- Pour tout événement e, on note $e.p$ l'indice du processus ayant provoqué cet événement.
- La relation classique de causalité est notée $\prec$.

Remarque préliminaire Pour la plupart des questions, utiliser la sémantique par triplet de Hoare et le tableau des actions de la figure A.2 pour donner une schéma de preuve de la propriété demandée. En particulier, pour les invariants, montrer que si l'invariant est vrai avant une opération Top ou Recaler, celui-ci est encore vrai après l'exécution de l'opération.

Questions

1. Décorer le chronogramme de la figure A.7 en précisant la date affectée par le mécanisme d'horloge proposé à chacun des événements. (Note : vous pouvez utiliser le chronogramme de la figure A.8 en quatrième page comme support de la réponse.)
2. Montrer que toute horloge H_i vérifie l'invariant suivant :

$$\texttt{invariant} \quad \forall j \neq i \ :: \ H_i[j] < H_i[i]$$

3. Montrer que l'horloge d'un processus est croissante. Autrement dit, deux événements distincts e et e' ayant lieu dans le même processus i et tels que e précède causalement e', sont datés par l'horloge H_i du processus avec des dates croissantes :

$$\texttt{invariant} \quad \forall e \neq e' \ : \ e.p = e'.p = i \ :: \ e \prec e' \Rightarrow \ h_e < h_{e'}$$

4. On appelle chemin causal $C(e_n)$ conduisant à un événement e_n, une suite d'événements $\{e_1, e_2, \ldots, e_n\}$ vérifiant la propriété suivante : $e_1 \prec e_2 \prec \ldots \prec e_n$. La longueur d'un chemin causal, notée $|C(e_n)|$ est égale au nombre d'événements de la suite. [2]

2. Par convention, on admet qu'un chemin causal $C(e)$ réduit à l'événement lui-même ($C(e) = \{e\}$) est donc de longueur 1.

Montrez que, pour tout événement e d'un processus $P_{e.p}$, la valeur de l'élément $h_e[e.p]$ du vecteur horloge h_e datant cet événement est égale à la longueur du (ou des) plus long(s) chemin(s) causal(causaux) conduisant à e. Autrement dit, l'invariant suivant est vérifié :

$$\texttt{invariant} \quad \forall\, e \;:\; h_e[e.p] = Max(|C(e)| : \forall C(e))$$

5. Montrer que ce type d'horloge capte la causalité entre événements. Autrement dit, si deux événements distincts e daté par le vecteur h_e et e' daté par le vecteur $h_{e'}$ sont tels que $e \prec e'$, alors $h_e[e.p] < h_{e'}[e'.p]$:

$$\texttt{invariant} \quad \forall e \neq e' \;::\; e \prec e' \Rightarrow\; h_e[e.p] < h_{e'}[e'.p]$$

Causalité directe

On introduit la relation de causalité directe, notée $\prec_m$ dans laquelle m dénote un message. Cette relation est vérifiée entre deux événements e et e' ayant lieu dans deux processus **distincts** si et seulement si il existe un chemin causal ne comportant qu'un seul message m entre e et e'. Autrement dit, il existe au moins une communication de $P_{e.p}$ à $P_{e'.p}$ dans l'intervale de temps $[e, e']$. À titre d'exemple, dans le chronogramme de la figure A.7, on a les relations :

$$r_1 \prec_{m_4} e_6, \quad e_1 \prec_{m_1} r_6, \quad e_1 \prec_{m_6} r_6, \quad e_1 \prec_{m_2} r_3$$

Questions

6. Donner, à partir du chronogramme de la figure A.7, deux autres exemples d'événements vérifiant la relation de causalité directe et deux exemples ne vérifiant pas cette relation (paires d'événements (e, e') tels que $\neg(e \prec_m \; e' \;\vee\; e' \prec_{m'} e)$).
7. Montrez que : $\forall e, e' \;:\; e.p \neq e'.p \;::\; e \prec_m e' \Rightarrow e \prec e'$.
8. Montrer que le système de datation proposé vérifie :

$$\texttt{invariant} \quad \forall e \neq e' : e.p \neq e'.p \;::\; h_e[e.p] \leq h_{e'}[e.p] \;\Leftrightarrow\; e \prec_m e'$$

Coupe cohérente

On suppose que les processus, au cours de leur exécution, décident (séparément) de prendre un cliché de leur état local $L[i]$ en le datant avec l'horloge locale du processus. Ces clichés locaux datés sont ensuite envoyés à un processus collecteur. Lorsque celui-ci possède une copie de tous les clichés locaux $L[i]$, il dispose d'un cliché global sous la forme du tableau L.

Questions

9. Montrez que le tableau des clichés locaux $L[i]$ constitue un cliché global cohérent si aucun des états locaux n'est causalement lié. Autrement dit si :

$$\forall i \neq j \;:\; \neg(L[i].e \prec L[j].e)$$

où $L[x].e$ désigne l'événement de prise de cliché local sur P_x.

10. Comment vérifier, à partir des dates $L[i].h$ des clichés locaux collectés, la cohérence du cliché global que ces clichés locaux représentent ? Penser à utiliser (les propriétés de) la causalité directe des questions $(6, 7, 8)$.

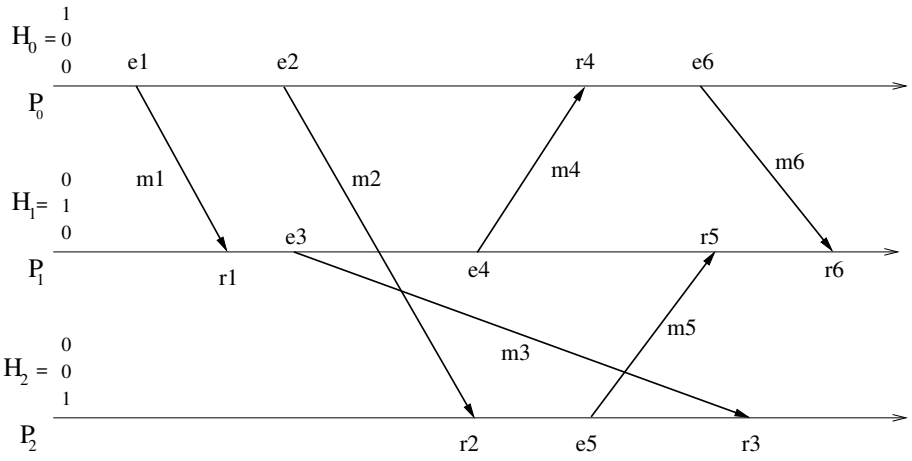


FIGURE A.8 – Chronogramme : à compléter

A.3 Causalité et horloges de Mattern, Interblocage et terminaison

A.3.1 Causalité et horloges de Mattern (10 points)

Le diagramme de la figure A.9 représente le début d'une exécution répartie. Les premiers événements sont datés à l'aide du mécanisme d'horloge de Mattern qui associe à tout événement x un vecteur horloge H_x. On note $x \prec y$ la relation de précédence causale et $H_x < H_y$ la relation d'ordre entre vecteurs horloges.

Questions (2 points par question)

1. Donner tous les chemins causaux débutant par les événements origines e_1 ou e_2 et aboutissant à l'événement r_4 dans le diagramme d'exécution de la figure A.9. Même question pour les chemins causaux débutant par les événements origines e_1 ou e_2 mais aboutissant à l'événement r_5.
2. On suppose que l'exécution réelle a ordonné les événements dans le temps global réel selon la séquence suivante :

   $$e_1; e_2; r_1; r_2; e_3; i_1; r_3; e_4; r_4; e_5; i_2; r_5$$

   Préciser, en le justifiant, si cette séquence est **causalement** équivalente à celle de la figure A.9. Autrement dit, représente-t-elle une exécution réelle équivalente à celle présentée dans la figure A.9 ?
3. Compléter le diagramme en affectant leur vecteur horloge (leur date) aux événements non datés.

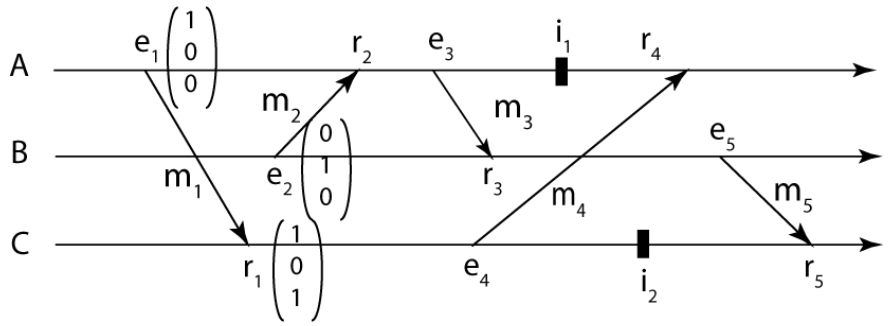


FIGURE A.9 – Diagramme événementiel d'une exécution répartie

4. Déduire du calcul précédent si les événements internes i_1 et i_2 sont causalement liés ou pas.
5. Montrer que, pour deux événements x de vecteur horloge H_x et y de vecteur horloge H_y, si les composantes des deux vecteurs horloges pour un site s vérifient $H_x[s] < H_y[s]$, alors tout chemin causal aboutissant à x est passé pour la dernière fois sur le site s avant tout chemin causal aboutissant à y.

A.3.2 Interblocage et terminaison dans un calcul réparti (11 points)

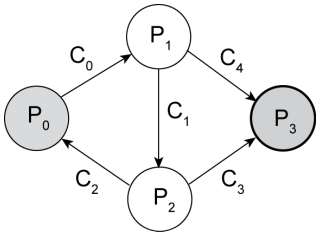


FIGURE A.10 – Graphe de communication

On considère un calcul réparti composé de N processus (N fixé et connu) qui communiquent par messages via des canaux unidirectionnels. Le graphe orienté de communication est connu et fixe comme dans l'exemple de la figure A.10 comportant 4 processus et 5 canaux. on suppose que tous les processus sont « accessibles » à partir du processus initial ($\exists$ au moins un chemin entre le processus initial P_0 et tout autre processus).

Les règles du calcul réparti constitué par ces N processus sont les suivantes :
- Tout processus reçoit des messages sur les canaux d'entrée ;
- Chaque processus P_i doit recevoir un message avant d'exécuter une étape ;
- Tout processus P_i, après l'exécution d'une étape envoie systématiquement un message sur chaque canal de sortie ;
- Au moins un processus « puits » n'a pas de canaux en sortie (P_3 dans la figure A.10) ;
- Un processus initial (P_0 dans fig. A.10) émet un message sur ses canaux de sortie pour activer le calcul.

En résumé, le comportement de tout processus participant obéit au schéma générique suivant [3] :

```
process P(i:0..N-1) {
  if (i==0) Emettre un message sur chaque canal de sortie ;
  while (true) {
    Recevoir un message sur l'un des canaux d'entrée ;
    Etape de calcul exploitant le message lu ;
    [ Emettre systématiquement un message sur chaque canal de sortie existant. ]
  }
}
```

Questions (1 point par question)

6. Dans quel modèle générique de calcul réparti peut-on classer le calcul décrit précédemment ?
7. Montrer la propriété stable suivante : si un processus appartenant à un cycle de communication (comme, par exemple, dans la figure A.10, le cycle entre P_0, P_1 et P_2) reçoit un message, alors il existera désormais toujours un message qui circulera sur ce cycle.
8. Montrer que, pour tout cycle de communication, un processus au moins du cycle recevra finalement un message. En déduire que les processus ne peuvent pas être interbloqués même s'il existe des cycles dans le graphe de communication.
9. Montrer, par contre, que le calcul ne peut pas se terminer s'il existe au moins un cycle.

3. Un processus puits ne peut évidemment pas émettre de message puisqu'il n'a pas de canal de sortie.

10. Donner une condition nécessaire et suffisante du graphe de communication pour que le calcul se termine en justifiant votre proposition.

Questions (2 points par question)

11. En supposant qu'il existe un **SEUL** processus puits, proposer un algorithme de terminaison permettant à ce processus puits de détecter la terminaison globale du calcul (s'il se termine!).
12. Adapter votre solution au cas d'un calcul comportant plusieurs processus puits.
13. On considère maintenant que chaque processus attend d'avoir reçu un message de **chaque** canal d'entrée avant de débuter une étape de calcul. Montrer que, dans un graphe sans cycle, le calcul est sans interblocage et se termine. Dans un graphe n'ayant qu'un puits, donner le critère très simple de détection de la terminaison en le justifiant.

A.4 Clichés et datation

Remarque : Toutes les questions valent 2 points

Préambule

Le problème a pour objectif d'étudier les propriétés de sûreté (cohérence) et de vivacité associées au calcul de clichés globaux à partir de clichés locaux ainsi que les relations existant entre prise de cliché global et datation par des horloges vectorielles de Fidge-Mattern. Pour simplifier, on ne s'intéresse ici qu'à la collecte d'un ensemble de clichés locaux permettant de construire un cliché global.

A.4.1 Propriété de sûreté : cohérence d'un cliché global (6 points)

Le diagramme de la figure A.11 représente le début d'un calcul réparti sous forme événementielle. Les 3 processus A, B et C échangent des messages applicatifs (de m_1 à m_5) et prennent des clichés locaux de leur état courant à chaque événement interne noté a_i, b_i ou c_i.

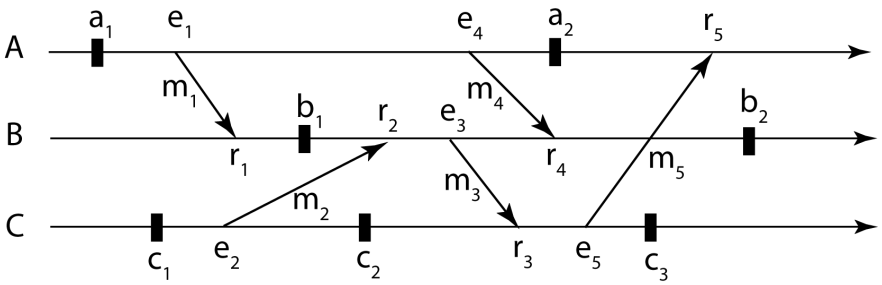


FIGURE A.11 – Calcul réparti sur 3 processus A, B et C

Questions

1. En vous basant sur la notion de coupure, préciser la propriété que doit vérifier un cliché global construit à partir d'un cliché local de chacun des processus afin que celui-ci soit cohérent.
2. Montrer que le cliché global construit à partir des clichés locaux a_1, b_1, c_2 est incohérent.
3. Montrer que le cliché global construit à partir des clichés locaux a_2, b_1, c_2 est, par contre, cohérent et préciser le(s) message(s) en transit associé(s) à ce cliché global.

A.4.2 Datation des clichés locaux (8 points)

On décide de dater tous les événements (prises de clichés locaux, émissions et réceptions de message) avec des horloges vectorielles de Fidge-Mattern. On suppose qu'il existe n processus communicants numérotés de 1 à n. On note $H_i[1..n]$ le vecteur horloge local au processus P_i et $e.D[1..n]$ la date vectorielle de l'événement e. Au début du calcul, les horloges de chaque processus vérifient : $\forall i : 0 < i \leq n : H_i[i] = 1 \wedge (\forall j \neq i : H[j] = 0)$

Questions

4. Dans la figure A.11, évaluer la date vectorielle associée à chaque événement de **prise de cliché local** (et exclusivement ceux-ci) et préciser aussi la date surchargeant chaque message envoyé de m_1 à m_5.
5. En considérant le cliché incohérent construit à partir des clichés locaux correspondant aux événements (a_1, b_1, c_2), montrer que l'incohérence est mise en évidence par les dates des clichés a_1 et b_1 qui vérifient : $a_1.D[1] < b_1.D[1]$
6. En déduire de façon plus générale qu'un cliché est cohérent si et seulement si les dates associées aux n clichés locaux D_i vérifient :

$$\begin{pmatrix} D_1[1] \\ \dots \\ D_i[i] \\ \dots \\ D_n[n] \end{pmatrix} = \mathbf{Max}(D_1, \dots, D_i, \dots, D_n) \equiv \begin{pmatrix} \mathbf{Max}(D_1[1], \dots, D_i[1], \dots, D_n[1]) \\ \dots \\ \mathbf{Max}(D_1[i], \dots, D_i[i], \dots, D_n[i]) \\ \dots \\ \mathbf{Max}(D_1[n], \dots, D_i[n], \dots, D_n[n]) \end{pmatrix} \quad \text{(A.1)}$$

7. Peut-on encore vérifier la cohérence d'un cliché global à partir des dates des clichés locaux si les seuls événements datés sont ces clichés locaux, autrement dit si les événements d'émission et de réception ne sont pas datés et si les messages ne servent qu'au recalage ? Justifier avec précision votre réponse.

A.4.3 Propriété de vivacité : collecte de clichés globaux (6 points)

On considère un calcul réparti composé de n processus ne communiquant que par un protocole d'échange d'un message jeton via un anneau logique comprenant tous les processus. Pour simplifier, on suppose que cet anneau de communication place les processus dans l'ordre de leur numérotation : $P_1, \dots, P_i, \dots, P_n$.

Le processus P_1 démarre le calcul en émettant le message jeton vers son successeur P_2. Après quoi, tous les processus ont le même comportement répétitif : attente de la réception du jeton issu du prédécesseur, traitement local, puis envoi du jeton vers le successeur. Le processus P_1 peut décider d'arrêter le calcul en ne propageant plus le jeton. On ne s'intéresse pas au contenu du message jeton.

Questions

8. Chaque processus prend un cliché local à chaque fois qu'il possède le jeton. Montrer qu'il est alors impossible, pour un processus collecteur qui recevrait ces clichés, de construire un cliché cohérent.
9. On choisit de prendre les clichés locaux lorsque le processus n'a **pas** le jeton. Un cliché local est pris au k-ème cycle s'il a été pris après la k-ème visite du jeton. Montrer qu'un cliché global construit avec les clichés pris lors d'un k-ème cycle est cohérent et préciser le nombre de message(s) en transit.
10. On suppose maintenant qu'un protocole causalement ordonné est utilisé pour transmettre les clichés locaux au collecteur. Lors de quels cycles peut-on prendre les clichés locaux pour être toujours sûr que le collecteur reçoit des séries de n clichés locaux consécutifs, chaque série contenant des clichés appartenant au même cycle et constituant un cliché global cohérent ?

Question Bonus (1 point)

Comment pourrait-on assurer **très simplement** que le collecteur détecte (sait) que le calcul est terminé ?

A.5 Flux multimedia et agents mobiles

A.5.1 Quelques problèmes de causalité entre flux multimedia

On considère une application échangeant des flux multimedia de type MPEGx par exemple. Un site A diffuse un flux vidéo vers 2 autres sites B et C. Après avoir reçu le début du flux émis par A, le site B émet un flux à son tour (flux audio par exemple) vers les sites A et C. Le chronogramme de la figure (A.12) donne une vision événementielle du déroulement de l'exécution répartie en ne conservant que les événements de début et fin de flux qu'il s'agisse d'un flux en émission ou d'un flux en réception.

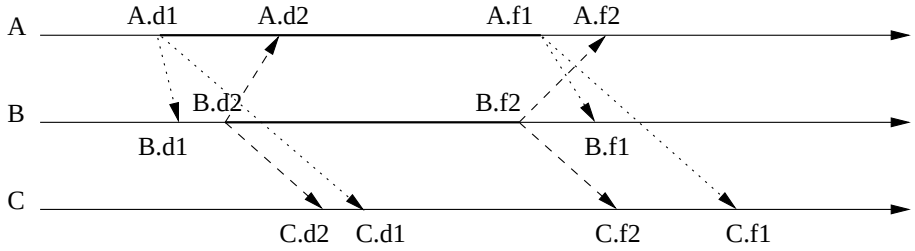


FIGURE A.12 – Vision événementielle des flux

Questions

1. Sur le chronogramme de la figure (A.12), les paires d'événements suivants sont-elles liées par la relation de causalité : $(A.d1, B.d2), (A.f1, B.f2)$, $(A.d2, B.f2)$, $(A.d2, C.f1)$? Justifiez vos réponses.

   Réponse
   $A.d1 \prec B.d2 \quad car \quad A.d1 \prec B.d1 \wedge B.d1 \prec B.d2$
   $A.f1 \parallel B.f2 \quad car \quad \neg(A.f1 \prec B.f2 \vee B.f2 \prec A.f1)$
   $A.d2 \parallel B.f2 \quad car \quad \neg(A.d2 \prec B.f2 \vee B.f2 \prec A.d2)$
   $A.d2 \prec C.f1 \quad car \quad A.d2 \prec A.f1 \wedge A.f1 \prec C.f1$

2. Montrer que la réception des flux 1 et 2 sur le site C présente une anomalie par rapport à la causalité en ce qui concerne leur début.

   Réponse Le site C voit le flux 2 commencer avant le flux 1 alors que le site B a vu débuter le flux 1 avant de commencer la diffusion du flux 2.

   Cette anomalie se traduit en terme de causalité par : $A.d1 \prec B.d2 \ \wedge \ C.d2 \prec C.d1$

3. Un protocole ordonné pourrait-il corriger cette anomalie sur l'ordre de démarrage des flux sur le site de réception C ? Justifiez votre réponse.

   Réponse Oui, puisqu'il assurerait : $A.d1 \prec B.d2 \ \Rightarrow \ C.d1 \prec C.d2$. Les événements $C.d1$ et $C.d2$ sont alors les événements de délivrance du premier message de chaque flux au niveau applicatif.

4. De façon similaire, la figure A.12 montre que les fins de flux sur les sites A et B se produisent dans un ordre différent : sur A, la fin de l'émission $A.f1$ du flux 1 précède la fin de la réception $A.f2$ du flux 2 et sur B, la fin de l'émission $B.f2$ du flux 2 précède la fin de la réception $B.f1$ du flux 1. Montrez que malgré cette inversion, aucune anomalie causale n'existe.

   Réponse Il n'y a pas d'anomalie causale puisqu'il n'y a pas de relation causale entre les événements de fin d'émission de ces deux flux : $A.f1 \parallel B.f2$.

5. Dans le chronogramme étudié, la question précédente conduit à considérer que la fin de réception des flux 1 et 2 sur le site C peut se produire dans n'importe quel ordre. Modifier le chronogramme pour obtenir une situation où la réception des flux 1 et 2 devrait être forcément ordonnée comme dans le chronogramme fourni (dans lequel $C.f2 \prec C.f1$) sans provoquer d'anomalie causale.

Réponse Le chronogramme ci-dessous introduit une causalité entre la fin des émissions des 2 flux, en l'occurrence $B.f2 \prec A.f1$.

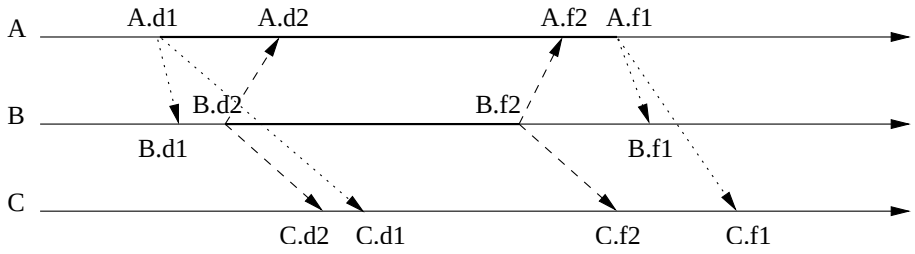


FIGURE A.13 – Terminaison causalement ordonnée des 2 flux

A.5.2 Rencontres entre agents mobiles

On considère une application répartie à base d'agents mobiles. Ces agents se rencontrent pour échanger des informations lorsqu'ils se trouvent simultanément sur le même site[4]. Une telle rencontre est considérée comme atomique et réalisée toujours entre une paire d'agents[5].

La figure (A.14) illustre le déplacement et les rencontres de tels agents désignés par $A, B, C, \ldots$

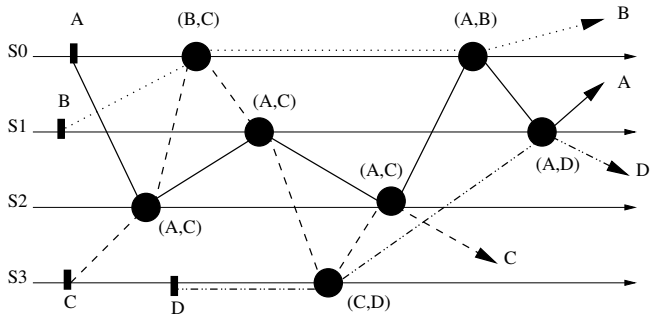


FIGURE A.14 – Rencontres d'agents mobiles sur des sites

On se propose de tracer des informations sur les rencontres qui ont lieu durant l'exécution des agents. Pour cela, on peut envisager de collecter des informations soit sur les sites de rencontre, soit dans le contexte local de chaque agent. On s'intéresse plus particulièrement à la datation des rencontres.

Trace sur les sites

On mémorise sur chaque site les dates des rencontres qui ont eu lieu. Le mécanisme de datation s'inspire du système de datation de Mattern. Une date est représentée par un vecteur de dimension N où N est le nombre de sites. Chaque site possède une horloge vectorielle H_i dotée d'une opération permettant de dater

4. On ne s'intéresse pas à leurs communications éventuelles à distance.
5. Un agent peut par contre rencontrer successivement plusieurs agents avant de se déplacer vers un autre site.

les rencontres et les agents vont transporter un vecteur courant unique représentant la date de leur dernière rencontre. On veut que la sémantique d'une date T affectée à une rencontre sur un site i_0 soit la suivante :

$$\textbf{invariant} \quad (\forall i \neq i_0 \ : \ T[i] = Card(R_i)) \wedge T[i_0] = Card(R_{i_0}) + 1$$

dans lequel l'ensemble R_i contient tous les événements de rencontre ayant eu lieu sur le site i et qui précèdent causalement la rencontre datée T. À titre d'exemple, la date T affectée à la rencontre (A, D) sur le site S_1 devrait être le vecteur $T = (2, 2, 2, 1)$.

Initialement, l'horloge H_i d'un site S_i est initialisée à zéro : $\forall j \ :: \ H_i[j] = 0$. La classe Horloge suivante possède une seule opération Dater qui ne sert qu'à dater les événements de rencontre.

```
class Horloge {
   int cpt[]; int loc;   // "loc" localise le site de l'horloge
   // Datation d'une rencontre
   public int[] Dater( int T1[], int T2[] ) {  ... }
   public Horloge(int où, int N) {
      loc = où; cpt = new int[N] ;
      for (int i = 0; i<N; i++) cpt[i]=0 ;
      }
}
```

La sémantique de Dater comporte deux paramètres d'entrée, en l'occurrence les deux vecteurs horloges transportés par les agents participant à la rencontre.

Questions

6. Donner sous forme d'un triplet de Hoare, la sémantique de Dater : $\{H_i.cpt = h\}\ H_i.\mathsf{Dater}(T1, T2)\{\ldots\}$ et programmer l'opération Dater dans la classe Horloge.

   Réponse La date de la rencontre est un vecteur ayant pour éléments le maximum de chaque vecteur et du vecteur courant du site sauf pour l'élément i qui comptabilise la nouvelle rencontre à partir de la valeur courante de l'élément i de l'horloge locale. [6]

$$\begin{gathered}\{H_i.cpt = h\} \\ H_i.\mathsf{Dater}(T1, T2) \\ \{(\forall k \neq i \ : \ H_i.cpt[k] = max(h[k], T1[k], T2[k])) \wedge H_i.cpt[i] = h[i] + 1\}\end{gathered}$$

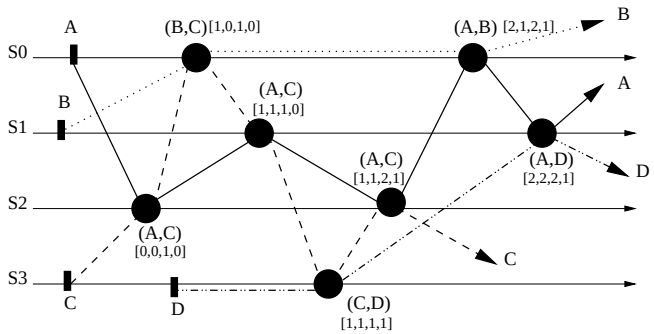
   L'opération Dater est donc :

```
public int[] Dater( int T1[], int T2[] ) {
    for (int i = 0; i<cpt.length; i++) cpt[i]=max(cpt[i],T1[i],T2[i]) ;
    cpt[loc]++ ;
    int[] res = new int[cpt.length]; System.arraycopy(cpt,0,res,0,cpt.length);
    return res ;
}
```

7. Décorer la figure avec les dates affectées aux rencontres en utilisant la feuille fournie en annexe.

   Réponse Les dates obtenues sont :

6. On a toujours pour une rencontre sur le site i : $H_i.cpt[i] \geq T1[i] \ \wedge \ H_i.cpt[i] \geq T2[i]$.



Trace dans les agents

Cette fois-ci, chaque agent a transporte une horloge vectorielle locale H_a (horloges embarquées dans les agents). Chaque horloge locale est initialisée à zéro : $\forall a \in Agents \ : \ \forall i \in 1..M \ : \ H_a[i] = 0$. Une date est donc représentée par un vecteur de dimension M où M est le nombre d'agents [7]. Par contre, les sites n'ont plus d'horloges locales. On doit donc trouver une implantation différente de l'opération Dater. On suppose que chaque agent fournit en paramètre à l'opération Dater son indice et son vecteur horloge local. La classe Datation définit la nouvelle opération Dater :

```
class Datation {
   // Datation d'une rencontre
   static public  int[] Dater(int a, int Ha[], int b, int Hb[]) { ... }
}
```

On veut que la sémantique d'une date T affectée à une rencontre entre un agent a et un agent b soit la suivante :

$$\textbf{invariant} \quad (\forall k \neq a, b \ : \ T[k] = Card(R_k)) \wedge T[a] = Card(R_a) + 1 \wedge T[b] = Card(R_b) + 1$$

dans lequel un ensemble R_x contient tous les événements de rencontre auxquels l'agent x a participé et qui précèdent causalement la rencontre datée T. À titre d'exemple, la date T affectée à la rencontre (A, C) sur le site S_1 devrait être le vecteur $T = (2, 1, 3, 0)$.

Questions

8. En désignant, dans la postcondition, le vecteur date résultat par T_r, donner sous forme d'un triplet de Hoare, la sémantique de Dater :

   $$\{H_a = h_a \wedge H_b = h_b\} \ Datation.\mathsf{Dater}(a, Ha, b, Hb)\{\ldots\}$$

   et programmer l'opération Dater dans la classe Datation.

   Réponse La solution consiste à prendre une nouvelle fois le maximum des éléments de chaque vecteur et à incrémenter de un les deux éléments concernant les agents de la rencontre :

   $$\begin{gathered}\{H_a = h_a \wedge H_b = h_b\} \\ Datation.\mathsf{Dater}(a, Ha, b, Hb) \\ \{\forall i \neq a, b \ : \ T_r[i] = max(h_a[i], h_b[i]) \wedge T_r[a] = h_a[a] + 1 \wedge T_r[b] = h_b[b] + 1\}\end{gathered}$$

   L'opération Dater correspondante est la suivante [8] :

7. On suppose que le nombre maximum d'agents est connu pour simplifier et conserver des vecteurs.
8. On remarque que, comme pour l'opération Dater précédente, on a : $\forall k \ : \ H_a[a] \geq H_k[a] \wedge H_b[b] \geq H_k[b]$

```
static public int[] Dater(int a, int Ha[], int b, int Hb[] ) {
    int[] res = new int[Ha.length];
    for (int i = 0; i<Ha.length; i++) res[i]=max(Ha[i],Hb[i]) ;
    Ha[a]++ ; Hb[b]++ ;
    return res ;
}
```

9. Montrer que l'on a l'invariant :

$$\textbf{invariant } \forall T \ : \ (\sum_{x=1}^{x=M} T[x])/2 = nr_T + 1$$

dans lequel le terme nr_T est égal au nombre total de rencontres (quel que soit les participants) qui précèdent causalement la rencontre datée par T.
Pour cela, montrer que si le précidat invariant est vrai avant l'exécution d'une opération Dater, il est encore vrai près l'exécution de l'opération.

Réponse D'après la sémantique de l'opération Dater, on a, puisque les éléments $H_a[a]$ et $H_b[b]$ sont maximaux :

$$\sum_{x=1}^{x=M} T[x] = \sum_{x=1}^{x=M} max(H_a[x], H_b[x]) + 2$$

Il reste donc à prouver que :

$$2 * nr_T = \sum_{x=1}^{x=M} max(H_a[x], H_b[x])$$

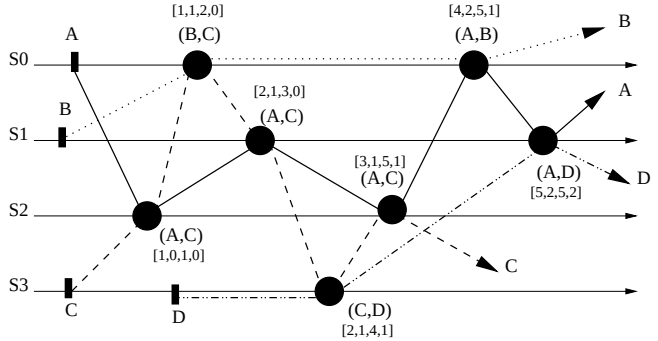
Pour les éléments $H_a[a]$ et $H_b[b]$, nous avons vu que :

$$max(H_a[a], H_b[a]) = H_a[a] \ et \ max(H_b[b], H_a[b]) = H_b[b]$$

Ces compteurs comptabilisent donc le nombre de rencontres de l'agent A (resp. B) avec n'importe quel autre agent, rencontres causalement antérieures à la rencontre courante.
Les éléments $max(H_a[x], H_b[x]), x \neq a, b$ enregistrent les rencontres de l'agent X perçues par les agents A ou B. Supposons que l'agent A a enregistré $H_a[x]$ rencontres de X et l'agent B a enregistré $H_b[x]$, avec par exemple $H_a[x] < H_b[x]$.
Puisque **TOUTES** les rencontres successives de l'agent X avec les autres sont causalement liées, cela signifie simplement que l'agent B a rencontré X **après** l'agent A et que les rencontres de X avec A sont donc toutes déjà comptabilisées. En tout état de cause, il suffit donc de prendre le maximum des deux éléments pour capter exactement le nombre de rencontres de l'agent X qui précédent cette rencontre entre A et B.

10. Décorer la figure avec les dates affectées aux rencontres en utilisant la feuille fournie en annexe.

Réponse Les dates de rencontre obtenues avec des horloges dans les agents sont :



A.6 Datation causale et simulation répartie

A.6.1 Causalité et horloges de Mattern (10 points)

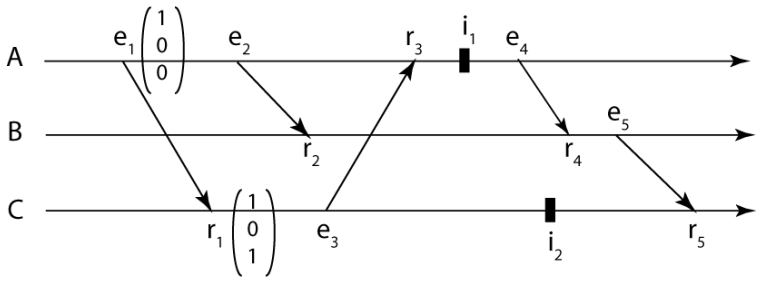


FIGURE A.15 – Diagramme événementiel d'une exécution répartie

Le diagramme de la figure A.15 représente le début d'une exécution répartie. Les premiers événements sont datés à l'aide du mécanisme d'horloge de Mattern qui associe à tout événement x un vecteur horloge H_x. On note $x \prec y$ la relation de précédence causale et $H_x < H_y$ la relation d'ordre entre vecteurs horloges.

Questions (2 points par question)

1. Compléter le diagramme en affectant leur vecteur horloge (leur date) aux événements non datés.
2. Trouver un plus court et un plus long chemin causal menant de l'événement e_1 à l'événement r_5.
3. Montrer que la somme des éléments d'un vecteur H_x associé à un événement x fournit une borne supérieure au plus long chemin causal possible conduisant à cet événement à partir du début du calcul.
4. Donner la condition nécessaire et suffisante qui doit être vérifiée par les vecteurs horloges pour qu'un événement z soit sur un chemin causal allant d'un événement x à un événement y.
5. En supposant connus l'ensemble des événements et leurs vecteurs horloges, déduire un algorithme pour trouver tous les chemins causaux entre un couple donné d'événements (x, y).

A.6.2 Simulation parallèle du traffic aérien (10 points)

On considère la simulation du traffic aérien entre des aéroports. Chaque aéroport est simulé par un simulateur particulier qui peut donc s'exécuter en **parallèle** avec les simulateurs des autres aéroports et communiquer par messages avec eux. On suppose que les interactions entre les aéroports sont réduites aux vols des avions modélisés, décrits par deux événements datés dans un temps global commun, en l'occurrence, l'événement de décollage d'un aéroport et l'événement d'atterrissage dans un autre (ou le même) aéroport.

Chaque simulateur progresse dans sa simulation en engendrant des événements de décollage et en enregistrant des événements d'atterrissage. Une horloge locale à chaque simulateur indique la date du dernier événement traité. Un simulateur d'aéroport contrôle ses envols locaux et calcule une date d'atterrissage future de chaque vol : plus précisément, lorsqu'un simulateur engendre un événement de décollage, il calcule aussi la date de l'événement d'atterrissage correspondant et il envoie un message contenant les informations relatives à cet événement (notamment sa date) au simulateur de l'aéroport sur lequel l'avion concerné atterrira (c'est-à-dire au simulateur devant traiter l'événement d'atterrissage).

La figure A.16 représente la simulation de trois aéroports avec 6 vols $(V_1, \ldots, V_6)$ et les paires d'événements de décollage et atterrissage correspondants.

Exécution de la simulation La simulation est découpée en **pas** successifs exécutés en parallèle par chaque simulateur. Un nouveau pas de simulation $i + 1$ ne peut commencer que lorsque **tous** les simulateurs ont

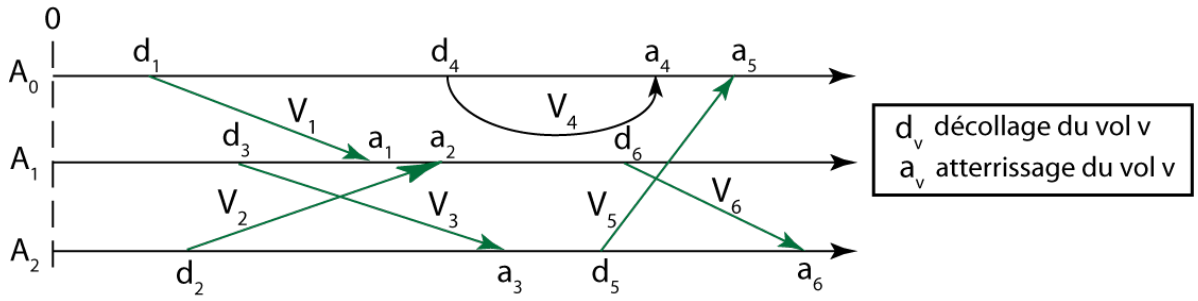


FIGURE A.16 – Diagramme de simulation du traffic aérien entre 3 aéroports

terminé le pas courant i de simulation. Le problème consiste à trouver la durée de ces pas de simulation, pour chacun des simulateurs, de façon à garantir une simulation globale cohérente dans laquelle chaque simulateur traite les événements **dans leur ordre croissant de datation** de façon sûre sans retour arrière.

Par exemple, si le simulateur (de l'aéroport) A_1 s'exécute en parallèle avec le simulateur A_0, il ne faut pas que ce simulateur A_1 fasse évoluer la simulation de cet aéroport au-delà de la date de l'atterrissage a_1 avant qu'il n'ait eu connaissance de cet événement : autrement-dit, le simulateur A_0 doit avoir auparavant engendré l'événement de décollage d_1 et l'événement d'atterrissage a_1, puis envoyé le message concernant l'atterrissage a_1 et ce message doit être parvenu au simulateur A_1. En effet, sinon, le simulateur A_1 aurait connaissance de l'événement a_1 trop tard et devrait revenir *en arrière* dans sa simulation.

Questions (2 points par question)

6. Expliquer pourquoi l'ensemble des événements (d_v, a_v) d'une simulation est assimilable à une exécution répartie modélisée de façon événementielle ;
7. En vous basant sur le diagramme de la figure A.16, précisez, partant de l'instant initial, jusqu'à quel événement limite inclus, de leur simulation, chacun des simulateurs (A_0, A_1, A_2) peut s'exécuter en parallèle avec les autres sans risquer « d'oublier » certains événements.
8. En supposant que chaque simulateur s'est exécuté jusqu'à ces points limites, donner de façon similaire le prochain pas exécutable (deuxième pas) par chaque simulateur.
9. Dans les deux pas de simulation des questions précédentes, vérifier que tout ensemble d'événements σ_i contenant les événements traités durant le i-ième pas de simulation parallèle vérifie la propriété :

$$\forall i : \forall (d_v, a_v) : d_v \in \sigma_i \Rightarrow a_v \notin \sigma_i$$

10. On suppose, maintenant, que l'on connaît la durée minimale m d'un vol, quel que soit l'aéroport de décollage. Montrer que l'on peut en déduire une durée de pas de simulation **fixe** et **commune** à tous les simulateurs garantissant des pas de simulation vérifiant la propriété précédente et donc assurant une simulation parallèle sans risque de retour arrière.

Bibliographie

[1] J.C.M. Baeten and W.P. Weijland. *Process Algebra*, volume 18. Cambridge Tracts in Theoritical Computer Science, Cambridge University Press, 1990.

[2] Michel Banâtre and Peter A. Lee, editors. *Hardware and Software Architectures for Fault Tolerance*, volume 774 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[3] Kenneth P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12) :37–53, December 1993.

[4] Kenneth P. Birman. *Building secure and reliable network applications*. Manning, 1996.

[5] Kenneth P. Birman, Robert Cooper, T. Joseph, Keith Marzullo, , Messac Makpangou, K. Kane, Frank Schmuck, and Mark Wood. *The ISIS System Manual, Version 2.1*. Cornell University, 2.1 edition, September 1990.

[6] Kenneth P. Birman and Bradford Glade. Consistent failure reporting in reliable communications systems. Technical Report 93-1349, Cornell University, May 1993.

[7] Kenneth P. Birman and Robbert van Renesse, editors. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1994.

[8] Jerszy Brzezinski, Jean-Michel Hélary, Michel Raynal, and Mukesh Singhal. Deadlock models and a general algorithm for distributed deadlock detection. *Journal of Parallel and Distributed Computing*, 31 :112–125, 1995.

[9] T.D. Chandra, V Hadzilacos, and S. Toueg. The weakest failure detector for solving concensus. *Journal of ACM*, 43(4) :685–722, 1996.

[10] Edsger W. Dijkstra and Carel S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(4) :1–4, 1980.

[11] E.W.D. Dijkstra and C.S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(4) :1–4, 1980.

[12] M. Fischer, N. Lynch, and M Paterson. Impossibility of distributed consensus with one faulty process. *Journal of ACM*, 32(2) :374–382, April 1985.

[13] Michael J. Fischer, Nancy A. Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1) :26–39, January 1986.

[14] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1984.

[15] D. LeVerrand. *Le Langage Ada : Manuel d'évaluation*. Dunod Informatique, 1982.

[16] Mesaac Makpangou and Kenneth P. Birman. Designing application software in wide area network settings. Technical Report 90-1165, Cornell University, October 1990.

[17] F. Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2 :161–175, 1987.

[18] F. Mattern. Virtual time and global states of distributed systems. In *Proceedings of the International Conference on Parallel and Distributed Computing*, pages 215–226. North-Holland Inc., 1988.

[19] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[20] M. Naïmi and M. Trehel. Un algorithme distribué d'exclusion mutuelle en log(n). *Technique et Science Informatiques*, 6(2) :141–150, 1987.

[21] B.J. Nelson. *Remote Procedure Call.* PhD thesis, DCS Carnegie Mellon University, 1981.

[22] M. Raynal. Détection répartie de la terminaison. In *Synchronisation et état global dans les systèmes répartis*, pages 157–172. Editions Eyrolles, 1992.

[23] M. Raynal. *Synchronisation et état global dans les systèmes répartis.* Editions Eyrolles, 1992.

[24] Michel Raynal. *Algorithmes Distribués et Protocoles.* Eyrolles, 1985.

[25] Michel Raynal. *Gestion des Données Réparties : Problèmes et Protocoles.* Collection Direction Études-Recherches EDF. Edition Eyrolles, 1992.

[26] Robbert van Renesse, Kenneth P. Birman, Roy Friedman, Mark Hayden, and David A. Karr. A framework for protocol composition in Horus. In *14th Symposium on Principles of Distributed Computing*, pages 80–89. ACM, August 1995.

[27] Robbert van Renesse, Kenneth P. Birman, Mark Hayden, Alexey Vaysburd, and David Karr. Building adaptative systems using Ensemble. Technical Report 97-1638, Cornell University, 1997.

[28] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus : A flexible group communications system. *Communications of the ACM*, 39(4) :76–83, April 1996.

[29] Aleta Ricciardi, André Schiper, and Kenneth P. Birman. Understanding partitions and the "no partition" assumption. In *4th Workshop on Future Trends Of Distributed Computing Systems*, pages 354–360, Lisboa, Portugal, September 1993. IEEE.

[30] A. Schiper. Early concensus in an asynchronous system with weak failure detector. *Distributed Computing*, 10(3) :149–157, 1997.

[31] Gerard Tel and Friedmann Mattern. The derivation of distributed termination detection algorithms from garbage collection schemes. *ACM Transactions on Programming Languages and Systems*, 15(1) :1–35, jan 1993.

[32] H. Zimmermann, J-S. Banino, A. Caristan, M. Guillemont, and G. Morisset. Basic concepts for the support of distributed systems : the chorus approach. In IEEE, editor, *2nd international conference on distributed computing systems*, pages 60–66, April 1981.