CHAPITRE I

GÉNÉRALITÉS

version du 14 octobre 2002

1 Objectifs et organisation du cours

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

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

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

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

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

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

2 Modèle de répartition

L'environnement réparti que nous considérons est composé :

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

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

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

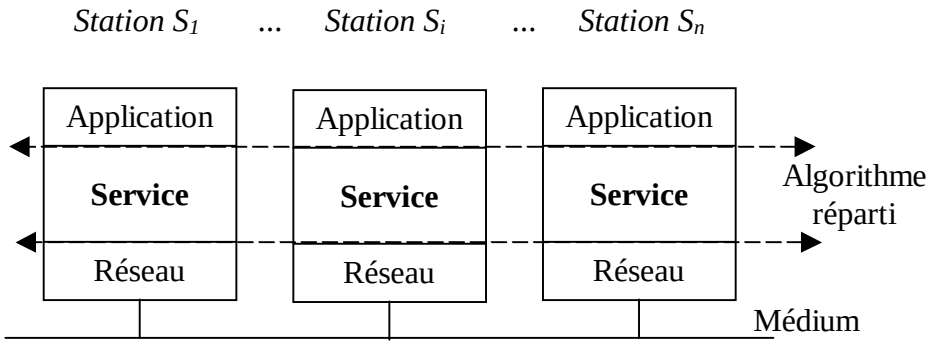

Figure 1.1 : L'environnement réparti

2.1 Le réseau et la couche réseau des stations

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

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

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

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

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

Plus précisément, les primitives d'émission retenues seront :

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

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

La primitive de réception sera :

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

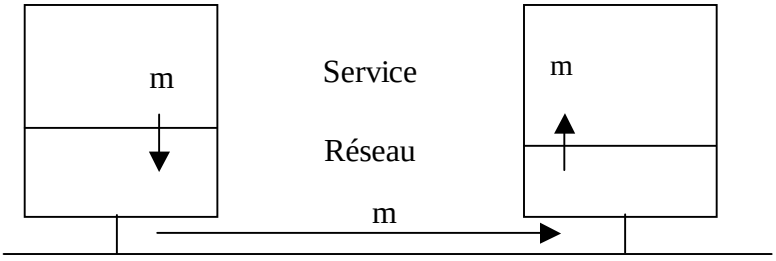

Figure 1.2 : Le trajet d'un message à travers les couches

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

2.2 La couche service

La couche service sera composée :

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

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

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

2.3 La couche application

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

2.4 Déroulement de l'application

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

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

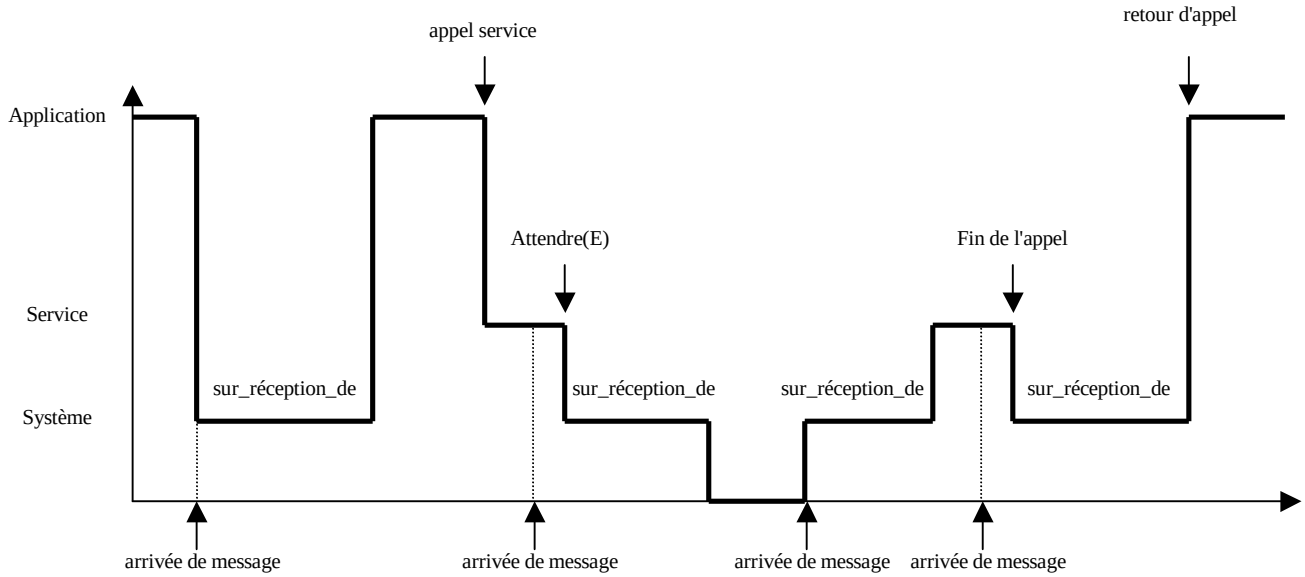

Figure 1.3 : Un déroulement d'une application sur un site

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

3 Scénario, évaluation et vérification d'algorithmes répartis

3.1 Scénario d'exécution

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

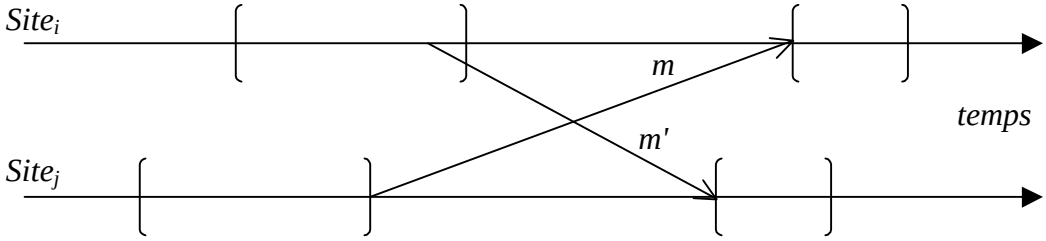

Figure 1.3 : Un exemple de chronogramme

3.2 Evaluation

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

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

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

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

3.3 Vérification

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

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

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

4 Références

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

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

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

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

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