



SYSTEMES DISTRIBUES ET SERVICES WEB

Programme de Première Année Master en Informatique
Option Ingénierie des Logiciels Complexes

***SUPPORT DE COURS REALISE PAR
PR DJAMEL MESLATI***

VERSION 2016

INTRODUCTION

PROGRAMME OFFICIEL

DESCRIPTION

Intitulé de la matière : Systèmes distribués et services WEB (SYDSER)

Semestre : 1

Unité d'Enseignement : ILC 2

Nombre de crédits : 6

Coefficient de la matière : 3

Mode d'évaluation : Examen + Travail pratique

Objectifs de l'enseignement

Ce cours vise à introduire les principes de base et les concepts des systèmes distribués en insistant sur leurs architectures, leurs modèles de communication ainsi que les standards souvent rencontrés dans les applications WEB. Sur le plan pratique, il est vivement recommandé d'étudier les différentes possibilités de communication, d'implémenter des protocoles ou de réaliser des applications et services WEB riches.

Connaissances préalables recommandées

Langage orienté objets, systèmes d'exploitation, compilation, réseaux et communication

Contenu

Chapitre 1 Introduction aux systèmes distribués (10%)

- 1 Définitions et caractéristiques (Concurrence, partage de ressources, communication par message, absence d'horloge commune, indépendance des pannes)
- 2 Exemples de systèmes distribués (Internet, intranet et systèmes mobiles)
- 3 Problèmes (L'hétérogénéité, la concurrence, la sécurité, les pannes, absence d'informations globales sur un système distribué)
- 4 Défis et objectifs.

Chapitre 2 Architecture des systèmes distribués (30%)

- 1 Taxonomie des systèmes distribués
- 2 Taxonomie au niveau matériel
- 3 Taxonomie au niveau systèmes d'exploitation
 - Les systèmes d'exploitation distribués
 - Les systèmes d'exploitation réseaux
- 4 Architecture des applications distribuées
 - Architecture Client/serveur et multitiers
 - Le modèle Mandataire/Cache
 - Les architectures poste-à-poste (P2P)
- 5 Les middlewares (Principe, Rôle, les services pris en charge, les standards, ...)

Chapitre 3 Les paradigmes de communication (30%)

- 1 Le passage de messages (principe, synchronisation, établissement d'un canal de communication)
- 2 Le RPC (Remote Procedure Call) (Principe, mécanismes et concepts utilisés, transmission des paramètres et des résultats)
- 3 Le RMI (Remote Method Invocation) (Principe, mécanismes et concepts exemple de déroulement d'un RMI)
- 4 Communication par événements et notifications
- 5 Communication de groupe (Principe, structure d'un groupe, opérations sur les membres, élaboration de la communication, ...)
- 6 Communication par mémoire partagée (Principe, mécanisme d'utilisation, maintien de la mémoire, création, protection, ...)
- 7 Communication par flux de données

Chapitre 4 Les services WEB : Définitions, Normes et Standards (30%)

1. Introduction
2. Le Web : évolution des protocoles
3. Architectures basées services
4. Les protocoles des architectures basées services

Références bibliographiques

1. Nicola Santoro, "Design and Analysis of Distributed Algorithms", John Wiley & Sons, 2007.
2. George Coulouris, Jean Dollimore & Tim Kindberg, "Distributed Systems, Concepts and Design, Addison-Wesley", 2001.
3. Andrew Tannenbaum, "Distributed Operating Systems", Prentice Hall International, 1995.
4. The World Wide Web Consortium : www.w3c.org

INTRODUCTION AUX SYSTEMES DISTRIBUES

CONTENU

1.1 Définitions et caractéristiques

Concurrence, partage de ressources, communication par message, absence d'horloge commune, indépendance des pannes

1.2 Exemples de systèmes distribués

Internet, intranet et systèmes mobiles

1.3 Problèmes

L'hétérogénéité, la concurrence, la sécurité, les pannes, absence d'informations globales sur un système distribué

1.4 Défis et objectifs

L'interopérabilité, l'ouverture, l'invariance à l'échelle (scalability), l'efficacité, la disponibilité, la gestion de la sécurité, le maintien de la consistance des ressources, la transparence (flexibilité), la gestion des situations d'exception (détection des pannes et reprise), la gestion des transactions réparties.

1.1 DEFINITIONS ET CARACTERISTIQUES

A. Tanenbaum et M. Van Steen [Tan 2002], donne la définition suivante : Un système distribué est une collection d'ordinateurs indépendants qui apparaissent à l'utilisateur comme un seul système cohérent.

De cette définition ressortent deux aspects : Le premier est de nature matérielle, il s'agit de l'autonomie des ordinateurs où chacun peut exécuter des tâches en concurrence (c à d au même moment) avec les autres. Le second fait référence au logiciel qui laisse apparaître le système distribué comme une seule entité cohérente. Bien entendu, le second aspect suppose une interconnexion des ordinateurs moyennant un réseau de communication physique.

Selon G. Coulouris et al [Cou 2001], Un système distribué ou réparti est un système dont les composants sont répartis sur différents nœuds d'un réseau d'ordinateurs. Ces composants communiquent et coordonnent leurs actions uniquement par l'échange de messages.

Les messages étant des suites de bits représentant des informations qui ont un sens pour l'émetteur et le récepteur. D'une manière plus formelle, les systèmes répartis se caractérisent par :

- ❖ Une absence de base de temps commune (horloge commune)
- ❖ Une absence de mémoire commune semblable à celle d'un système multiprocesseur
- ❖ Les pannes des composants sont indépendantes

La motivation principale des systèmes répartis est l'amélioration du partage des ressources. Autrement dit, les ressources disponibles dans un ordinateur donné peuvent être utilisées par un autre et vice versa. Par exemple, une imprimante connectée à un ordinateur peut recevoir des informations à imprimer provenant d'un autre ordinateur. De même, les fichiers disponibles dans un répertoire d'un ordinateur donné peuvent être manipulés par les autres ordinateurs, etc.

Naturellement, ce partage de ressources entre tâches concurrentes introduit des problèmes souvent désignés par « Problèmes de concurrence ». Ces problèmes existent aussi dans les systèmes centralisés multitâches (systèmes multiprocesseurs ou monoprocesseur avec partage de temps). Cependant, dans un système distribué, les problèmes de concurrence se trouvent accentués par le fait que la gestion des ressources n'est pas centralisée et la nécessité d'opérer des communications entre ordinateurs, ce qui exige des techniques de gestion plus sophistiquées.

La communication dans les systèmes distribués s'opère entre entités physiques ou logiques et se base sur la possibilité d'échange de messages qui est toujours supportée par le réseau physique d'ordinateurs. Par exemple, l'échange entre deux cartes de communication réseau est un échange entre entités physiques. L'échange entre deux systèmes de gestions de fichiers est un échange entre entités logiques. Il en est de même pour la communication entre applications où l'échange s'opère entre entités logiques appelées processus. Il existe diverses approches de communications allant d'un simple échange de bits à des protocoles complexes tel que l'invocation de méthodes à distance (couramment nommé RMI qui est une abréviation de Remote Method Invocation) ou la communication par événements. Notons cependant que l'échange entre entités logiques passe par l'échange entre entités physiques.

L'absence d'horloge commune citée précédemment comme caractéristique des systèmes répartis, signifie que seule la communication de messages peut être utilisée pour coordonner les activités des différentes tâches du système et garantir la cohérence lors du partage des ressources. En effet, il n'est pas possible de réaliser une synchronisation des horloges des différents ordinateurs et coordonner par la suite les tâches en fonction d'un référentiel temporel unique.

Dans un système distribué, les ordinateurs peuvent tomber en pannes indépendamment les uns des autres. De même, les lignes de communication physiques peuvent être défaillantes. Mais au-delà des aspects matériels, les tâches exécutées sur chaque ordinateur peuvent échouées. Ces pannes d'ordre physique ou logique peuvent

avoir un effet local plus ou moins étendu. Par exemple, une panne au niveau d'un réseau de communication engendre l'isolation des ordinateurs qui y sont connectés sans pour autant arrêter l'exécution sur ces derniers. Les programmes (processus) sur ces ordinateurs peuvent ne pas être capables de détecter si le réseau est en panne ou s'il est simplement devenu lent. De même, l'échec d'une activité quelconque sur un ordinateur n'est pas directement détectable par les autres ordinateurs sur le réseau. Cependant, une des caractéristiques des systèmes distribués est l'indépendance de ces pannes.

Lorsque, une application fait intervenir plusieurs composants où chacun s'exécute sur un ordinateur séparé, il incombe au concepteur de cette application d'anticiper les différentes pannes qui peuvent arrivées et prévoir des mécanismes pour garantir la poursuite de la coordination et l'atteinte de l'objectif visé.

1.2 EXEMPLES DE SYSTEMES DISTRIBUES

Les exemples des systèmes distribués sont nombreux et variés. Un même système distribué peut être fondé sur plusieurs autres systèmes distribués de niveaux différents. En effet, une application distribuée est un système composé de processus qui s'exécute sur plusieurs ordinateurs connectés par un réseau. Cet ensemble d'ordinateurs connectés est lui-même un système distribué. Les couches des systèmes d'exploitation qui assurent le maintien du réseau constituent un système distribué, etc. Comme exemples illustratifs, nous considérons Internet, les intranets et les systèmes mobiles.

1.2.1 Internet

Internet est une vaste collection de réseaux d'ordinateurs interconnectés. Les programmes qui s'exécutent sur ces ordinateurs interagissent par l'échange de messages en utilisant un moyen de communication ou un autre. La mise en œuvre des moyens de communication (mécanismes et protocoles) constitue un défi considérable qui a permis à un ordinateur quelconque d'échanger des messages avec n'importe quel autre ordinateur dans le réseau.

Internet constitue actuellement le système distribué le plus large au monde et ceci sur quasiment tous les plans. Des utilisateurs de n'importe quelle position dans le monde peuvent utiliser les services offerts par le WWW (World Wide Web), le FTP (File Transfert Protocole), et autres applications. Les services disponibles peuvent être étendus librement et le système peut être agrandi par l'ajout d'ordinateurs à n'importe quel moment.

Internet a une structure composée d'intranets connectés par des backbones. Ces derniers étant des moyens de communication de haute capacité (utilisant des satellites, la fibre optique, etc.). Les intranets sont la propriété de compagnie et d'organisations

diverses. Les fournisseurs des services Internet (Internet Service Providers) sont en fait des compagnies qui offrent des connexions aux utilisateurs d'Internet (personne ou organisation) par des modems et d'autre dispositifs qui leurs permettent l'accès aux services d'internet aussi bien que l'accès à des services locaux tels que l'hébergement de pages WEB.

Bien que couramment confondus, Internet se distingue du WWW. Le World Wide Web n'est autre qu'une application parmi tant d'autres que Internet supporte. Par exemple, le FTP est une autre application très utilisée. De même, l'échange Peer-to-Peer (poste à poste) connaît actuellement un succès grandissant. Le WWW représente un système distribué logique consistant en un nombre considérable de ressources (pages Web, fichiers de données et services) référencées par des URL (Uniform Resource Locator). Les pages Web contiennent des liens (des URLs) qui réfèrent des ressources et d'autres pages Web. L'ensemble des pages Web constitue une toile immense interconnectée par des liens qui permettent de naviguer d'une page à l'autre et de déclencher l'exécution de services divers. Le WWW est supporté par des standards et normes tels que http et Html qui régissent sa structure et le comportement des logiciels qui le supportent tels que les serveurs http et les navigateurs internet.

La figure 1.1 donne un aperçu sur l'architecture Internet

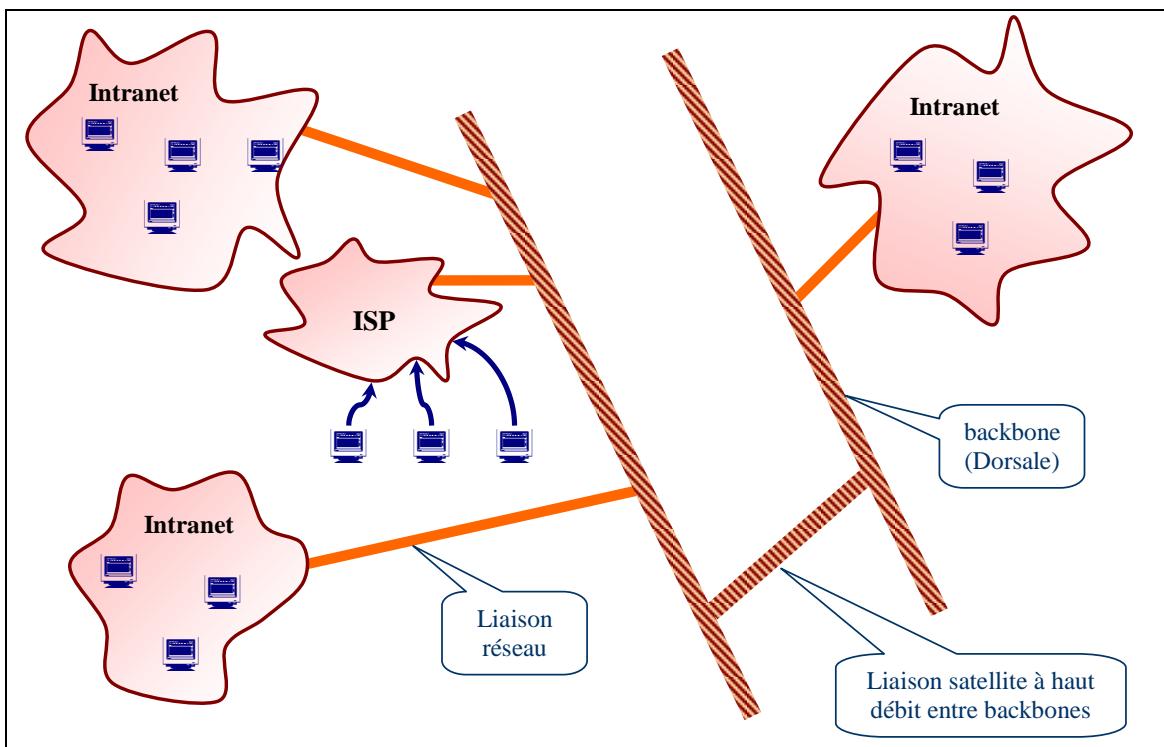


Figure 1.1. Structure d'un fragment Internet

1.2.2 Intranet

L'intranet est une portion Internet administrée séparément et délimité de telle sorte à garantir une politique de sécurité locale.

L'intranet est typiquement composé de plusieurs réseaux locaux (LAN ou Local Area Networks) reliés par des backbones. Chaque réseau local dispose de ressources gérées par des serveurs. La configuration d'un intranet est à la charge de l'organisation qui l'administre et peut varier largement.

Les intranets se connectent à Internet via des routeurs qui permettent aux utilisateurs d'avoir accès aux services Internet mais aussi d'avoir accès aux services d'autres intranets.

Les organisations qui administrent les intranets se protègent des menaces extérieures en utilisant des Firewalls. Ces derniers sont des programmes qui filtrent les données en entrée et en sortie en fonction des politiques de sécurité adoptées par les organisations.

La figure 1.2 donne un exemple de structure d'intranet.

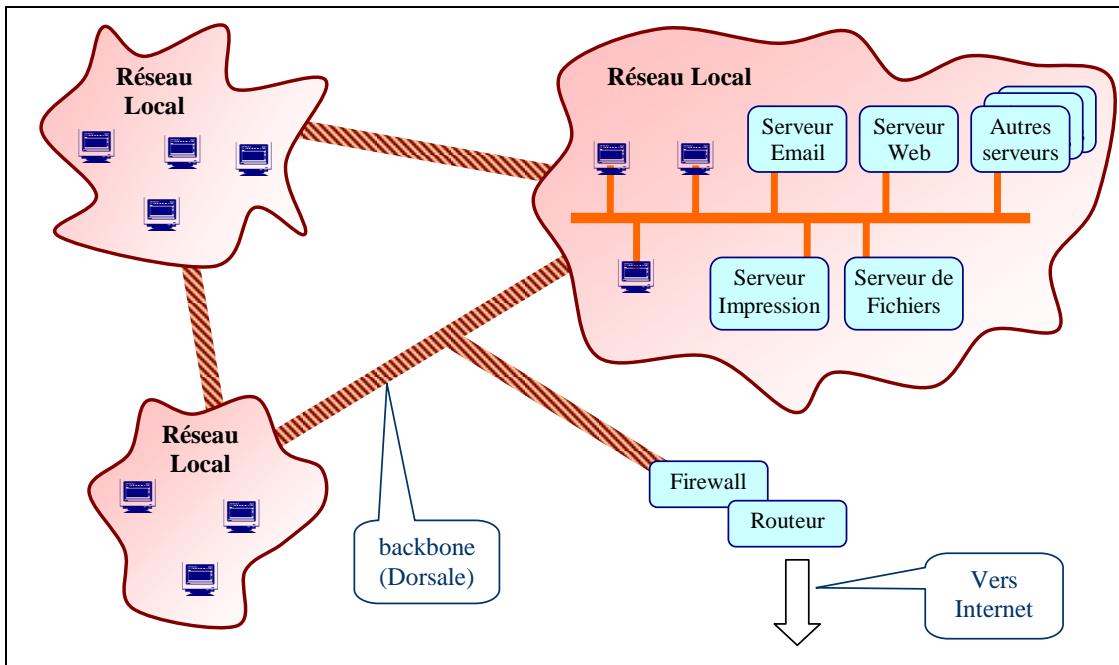


Figure 1.2. Structure typique d'un intranet

1.2.3 Les systèmes mobiles

Le progrès dans la miniaturisation et les réseaux sans fil laisse envisager le concept de *l'espace d'information totalement connecté* (fully connected information space) où chaque entité dotée d'une certaine fonctionnalité est connectée aux autres entités moyennant des dispositifs électroniques et des liaisons sans fil (voir figure 1.3). Ces liaisons permettent à chacune d'avoir des informations sur les autres et éventuellement invoquer les services qu'elles offrent. Comme la majorité des entités sont dynamiques ont affaire à des systèmes mobiles. Le partage de ressources dans de tels systèmes est une tâche complexe qui nécessite une certaine forme d'organisation et des mécanismes et protocoles ad hoc.

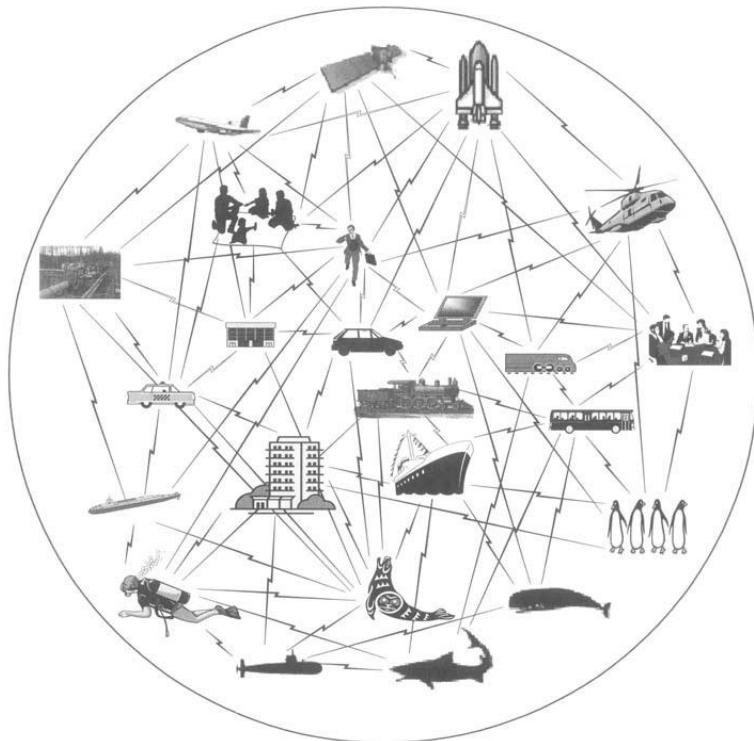


Figure 1.3. Espace d'information totalement connecté

La figure 1.4 donne un exemple d'organisation où un utilisateur visitant une organisation quelconque, accède à son intranet (qualifié de Host) via un réseau sans fil et utilise les ressources et services disponibles (utilisation de l'imprimante locale, accès à des bases de données, ...). Ce même utilisateur, a aussi la possibilité d'utiliser, à distance, l'intranet de son domicile via un téléphone portable. Cet exemple illustre une partie de la diversité des possibilités qu'offrent les systèmes mobiles.

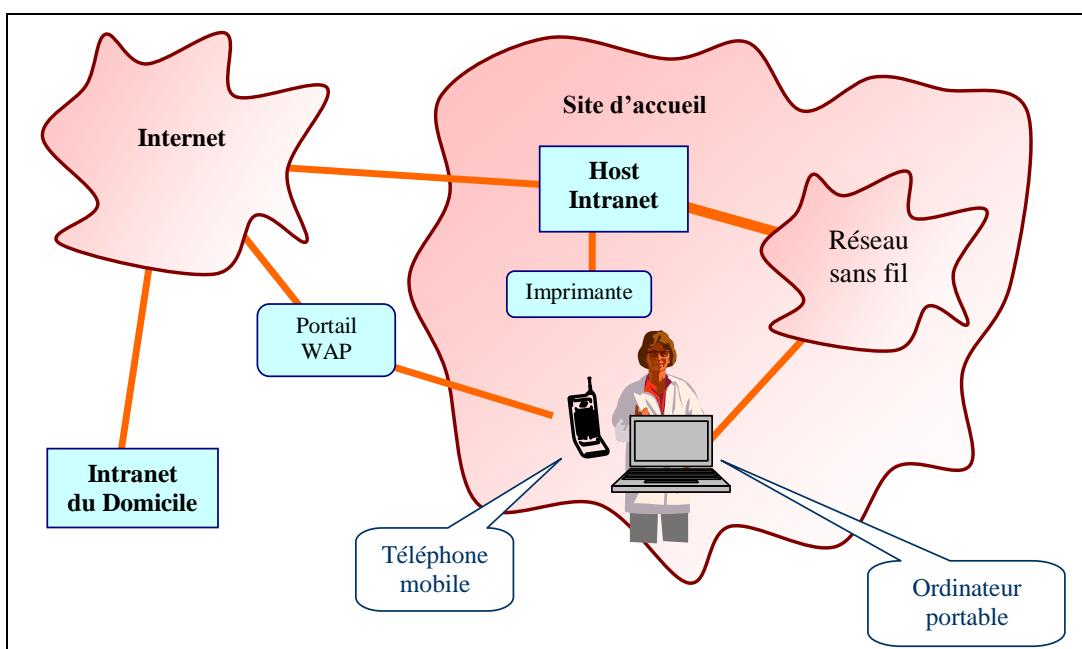


Figure 1.4. Exemple d'organisation pour systèmes mobiles

1.3 PROBLEMES

Actuellement, les applications distribuées apparaissent comme une dernière couche au dessous de laquelle il y a plusieurs autres. On en distingue trois telles qu'illustrées par la figure 1.5.

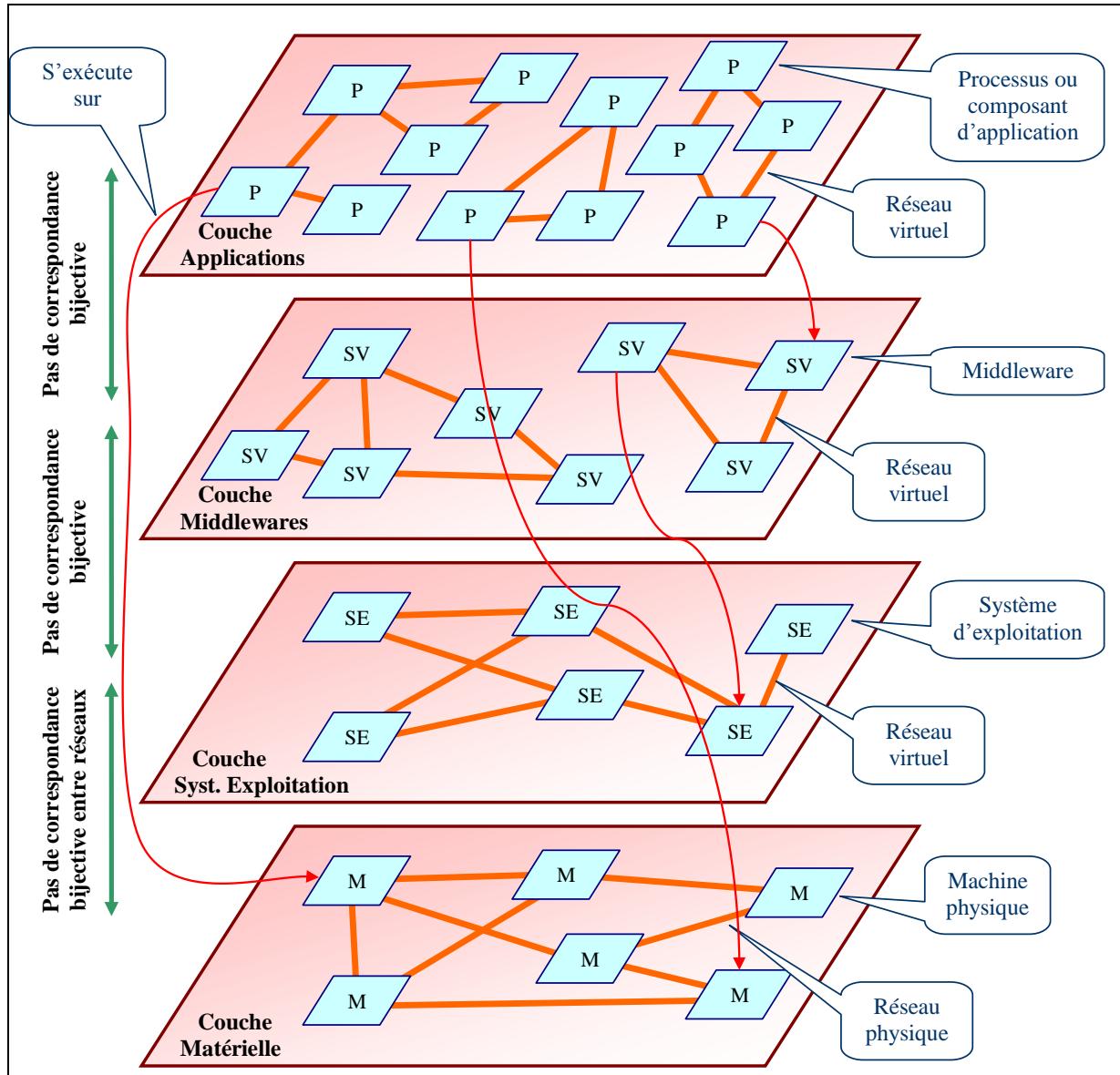


Figure 1.5. Structuration en couches des systèmes distribués

Dans chaque couche, il est possible d'avoir diverses entités reliées par un réseau qui est physique dans la couche matérielle et virtuel dans les autres couches. Chaque couche matérialise, à elle seule, un système distribué. Cependant, comme les entités de chaque couche s'exécutent en utilisant les services disponibles dans les couches inférieures, les applications distribuées sont donc mises en œuvre moyennant plusieurs systèmes distribués. La couche middleware est une couche spécifique pour les systèmes distribués qui est censée cacher les subtilités des couches inférieures et offrir aux

concepteurs des applications distribuées des services plus convenables de nature à réduire leur complexité.

Le schéma de la figure 1.5 laisse apparaître une complexité aussi bien au niveau de chaque couche mais également une complexité dans les relations qu'a une couche avec les autres. Cette situation engendre plusieurs problèmes auxquels la conception des systèmes distribués doit faire face, il s'agit de l'hétérogénéité, la concurrence, la sécurité, les pannes et l'absence d'informations globales. Nous discutons ces problèmes dans ce qui suit.

1.3.1 L'hétérogénéité (Heterogeneity)

Elle désigne les différences qui existent entre les différents composants d'un système distribué. Ces différences peuvent être au niveau de la couche matérielle (réseaux physiques et plateformes matérielles), de la couche systèmes d'exploitation (divers systèmes d'exploitation), de la couche middleware et de la couche application (utilisation de langages de programmation différents, etc.).

Dès lors en se retrouve devant une problématique : les composants d'un système distribué sont appelés à coopérer pour partager les ressources mais chaque composant peut avoir ses spécificités qui limitent ce partage ou le rendent difficile.

L'hétérogénéité au niveau matériel vient de la différence qui existe entre les ordinateurs aussi bien qu'entre les réseaux. Dans un même système distribué, il est possible de trouver un ordinateur personnel, une station de travail sophistiquée ou même un ordinateur multiprocesseur puissant. Les réseaux physiques présentent, eux aussi, beaucoup de différences : réseau de fibre optique, réseau sans fil, taille et structure des trames, etc.

La couche systèmes d'exploitation présente naturellement une hétérogénéité relativement importante malgré le fait qu'elle cache les subtilités de la couche matérielle. Ainsi, certains ordinateurs dans le système distribué peuvent être dotés d'un système d'exploitation Unix, d'autres utiliseront Windows, Mac OS ou Solaris, etc.

Au niveau middleware, il existe actuellement plusieurs possibilités. En effet, Microsoft propose sa plateforme .Net qui est censée faciliter la coopération entre les objets de diverses applications réalisées avec des langages de programmation différents. Corba est une spécification ouverte du consortium OMG qui a plusieurs réalisations pratiques qui offrent des moyens de partage et de coordination entre les composants des applications réparties. Les différents middleware sont appelés à coopérer malgré leur hétérogénéité pour faciliter la réalisation des applications distribuées.

Dans la couche application, l'hétérogénéité provient principalement de l'utilisation de langages de programmation différents, ce qui engendre des problèmes lors de la

manipulation des données, l'appel de procédures, etc. Par exemple, il peut y exister une différence dans le format des données où sur un ordinateur on utilise le stockage *Little Endian*, et sur un autre, un stockage *Big Endian*. De même pour la représentation des chaînes de caractères, les tableaux, les structures de données complexes (fichiers), etc.

1.3.2 Problème de la concurrence

Le problème de la concurrence provient de la manipulation simultanée des ressources par plusieurs programmes (processus). En effet, certaines ressources ne sont pas partageables et leur manipulation ne peut se faire que par un processus à la fois. C'est le cas des ressources physiques telles que l'imprimante mais aussi des ressources logiques telles que les fichiers, les tables des bases de données, etc.

Le problème de la concurrence se pose pour les systèmes distribués comme pour les systèmes centralisés (i.e. multiprocesseurs). La gestion de la concurrence dans les systèmes distribués nécessite des mécanismes et des outils tout comme les systèmes centralisés (moniteurs, sémaphore, etc.). Cependant, dans le cas des systèmes distribués, la gestion de la concurrence se base en partie sur la communication par messages et cette dernière introduit d'autres sources de difficultés, telles que la perte de messages ou leur corruption, d'en-t-il faut tenir compte.

1.3.3 Problème de la sécurité

Les problèmes de sécurité se posent pour tous les systèmes informatiques. Cependant, dans les systèmes distribués, la vulnérabilité se trouve accentuée par la répartition même. Ainsi, il est naturel dans le commerce électronique qu'une partie de l'application (un client) envoie des messages (par exemple numéro de la carte de crédit) à un autre (un serveur). Malheureusement le message peut être intercepté, lors de sa transmission, et son contenu divulgué, ce qui peut entraîner de conséquences graves aussi bien pour le client que pour le serveur.

D'autres problèmes se posent aussi, on cite :

- ★ L'authentification : Comment le client (respectivement le serveur) peut-il être sûr de l'identité du serveur (respectivement le client) ? Par exemple, le client d'une banque a besoin de connaître l'identité de celle-ci avant d'ouvrir un compte et y déposer une somme d'argent, de même, le serveur doit connaître l'identité du client avant d'entreprendre une transaction quelconque sur son compte.
- ★ L'attaque de services : Souvent les applications distribuées sont organisées selon le modèle client-serveur où le client invoque les services du serveur. Ce dernier dispose de ressources en nombre suffisant pour répondre à un nombre qui peut être important mais limité. L'attaque de service consiste à submerger le serveur par un nombre de requêtes qui excède ces capacités et ralentit considérablement sa cadence.

- ❖ Sécurité du code mobile : Le code mobile n'est autre qu'un programme qui migre d'une machine à l'autre dans un réseau dans le but d'accomplir une certaine tâche. A son arrivée, le code mobile utilise les ressources d'une machine pour accomplir sa tâche. Il peut donc présenter un risque s'il véhicule un virus ou un code mal intentionné.

1.3.4 Problème des pannes

Les systèmes peuvent échouer dans l'accomplissement de leurs tâches suite à une faille matérielle ou logiciel qu'on appelle couramment *panne*. Les pannes peuvent conduire à des résultats erronés comme elles peuvent engendrer l'arrêt de toute ou partie de l'application distribuée.

Les pannes peuvent avoir lieu à différents niveaux de la structure en couches présentée dans la figure 1.5. Une panne au niveau matériel peut inclure la perte d'un message par le réseau de communication, des erreurs d'entrée/sortie, la corruption d'une donnée ou d'un message, etc. Ces pannes, au niveau matériel, se propagent aux niveaux supérieurs engendant d'autres pannes de niveau supérieur, par exemple arrêt prématué d'un processus.

Les pannes peuvent apparaître aussi dans les différentes couches et se propager éventuellement aux autres couches. L'origine des pannes peut être une raison matérielle ou une raison logique liée à la conception des applications, des middlewares et des systèmes d'exploitation. Certaines peuvent avoir comme origine une faille dans la gestion de la concurrence ou la coordination en général. Les pannes d'ordre logiques sont innombrables du fait que les applications distribuées sont, par nature même, complexes. Par exemple, l'omission, dans le programme d'un processus, de la libération d'une ressource peut conduire à une situation d'interblocage. L'absence de l'exclusion mutuelle pour l'accès à une ressource critique, par exemple un tampon, peut avoir des répercussions importantes sur l'ensemble de l'application distribuée.

1.3.5 Absence des informations globales

Dans un système centralisé, une application se compose de plusieurs processus qui coopèrent pour réaliser un objectif commun. A tout moment, il est possible d'avoir des informations sur l'état de chaque processus et sur l'état des différentes ressources. Ainsi, chacun des processus peut avoir une vision globale de l'application et de son environnement. La présence de cette vision globale facilite grandement la mise au point des mécanismes de coordination des processus dont, par exemple, la gestion des ressources.

La vision globale dans les applications centralisées est due au fait que les processus partagent une mémoire commune qui renferme les informations sur les ressources et les processus.

Dans le cas des systèmes distribués, la mémoire commune est inexistante. Chaque processus se trouve dans un environnement différent (système d'exploitation différent) et son état est une information interne à ce dernier. De même, les informations sur les ressources partagées ne sont pas accessibles en temps réel. De ces faits, la coordination d'un ensemble de processus distribués nécessite le recours à la communication par échange de messages. Malheureusement, un message parvient toujours à sa destination avec un certain retard et par conséquent l'information qu'il véhicule ne reflète pas toujours l'état d'un processus ou d'une ressource, car ces derniers peuvent changer d'état durant la transmission du message.

Il est impossible d'avoir une vision globale instantanée de l'état d'un système distribué et cette situation rend la coordination dans ces derniers un véritable challenge.

1.4 DEFIS ET OBJECTIFS

Comme souligné dans la définition des systèmes distribués, l'objectif essentiel est le partage des ressources. Le concepteur d'une application répartie doit, en plus du développement des programmes assurant les fonctionnalités de celle-ci, tenir compte des différents problèmes cités précédemment et prévoir des mécanismes pour y faire face de telle sorte à faire apparaître le système distribué comme un tout cohérent.

Ainsi, à la complexité inhérente aux applications réparties, s'ajoute une complexité due à la distribution, ce qui constitue un défi considérable pour les concepteurs.

Pour faciliter le travail des concepteurs, les couches middlewares ont vu le jour. Ces dernières offrent des services plus convenables à utiliser. L'idée de simplifier le développement des applications en ajoutant des couches qui prennent en charge certains aspects liés à la répartition n'est pas nouvelle. En effet, les systèmes d'exploitation eux mêmes comprennent des services spécifiques à la répartition. Cependant, les disparités entre les divers systèmes d'exploitations et l'impossibilité d'arriver à un consensus ont conduit à la création des couches middlewares.

Faire face aux problèmes des systèmes distribués signifie apporter une solution à chacun soit au niveau du middleware (voire au niveau des couches inférieures) soit au niveau de l'application elle-même. Résoudre ces problèmes au niveau des middlewares vise en premier lieu à les masquer pour le concepteur de l'application et par conséquent pour l'utilisateur aussi. Si par contre, les problèmes sont résolus au niveau des applications, on les masque pour les utilisateurs mais ce sont les concepteurs de ces applications qui doivent les prendre en charge.

Quelque soit le niveau considéré, les solutions apportées doivent garantir des propriétés fondamentales suivantes : l'interopérabilité, l'ouverture, l'invariance à l'échelle (scalability), la gestion de la sécurité, l'efficacité et la disponibilité, le maintien de la

consistance des ressources, la transparence (flexibilité), la gestion des transactions réparties, la tolérance aux fautes et la gestion des situations d'exception (détection des pannes et reprise) etc.

Ces propriétés constituent de véritables défis pour les concepteurs de systèmes distribués. Nous les discutons dans les paragraphes suivants.

1.4.1 L'interopérabilité

L'interopérabilité est une caractéristique importante qui désigne la capacité à rendre compatibles deux systèmes quelconques. A son tour, la compatibilité est la capacité qu'ont deux systèmes à communiquer sans ambiguïté. L'interopérabilité nécessite que les informations relatives aux parties de communication des systèmes considérés soient disponibles sous la forme de standards ouverts.

L'interopérabilité vise à réduire le problème de l'hétérogénéité. Le masquage de celle-ci passe en premier lieu par l'utilisation d'un protocole unique de communication (e.g. cas de TCP/IP pour Internet). Pour établir un échange de message, il faut utiliser des standards qui cachent les différences entre les différentes plateformes.

Il existe actuellement deux approches principales de standardisation pour masquer l'hétérogénéité : les middlewares et les machines virtuelles.

Un middleware est une couche logicielle qui masque l'hétérogénéité en offrant aux programmeurs d'une application un modèle de programmation plus convenable. Concrètement un middleware est représenté par des processus et des ressources d'un ensemble d'ordinateurs, qui interagissent les uns avec les autres (communiquent et partagent des ressources).

Un middleware, tel que CORBA ou DCOM, offre des services qui peuvent être utilisés pour construire des composants logiciels pouvant fonctionner ensemble dans un système distribué.

Les middlewares améliorent la communication en offrant des abstractions telles que

- ★ Le RMI (Remote Method Invocation) : C'est la possibilité, pour un objet, d'invoquer la méthode d'un autre objet situé sur une plateforme distante.
- ★ La notification d'événement : Les événements représentent un moyen efficace de propagation d'information d'une plateforme vers une autre ou plusieurs autres plateformes ou composants d'une application répartie.
- ★ Communication entre groupes de processus
- ★ Gestion de la duplication des données partagées
- ★ Transmission de données multimédia en temps réel

Les machines virtuelles permettent de supporter le code mobile. Ce dernier désigne la possibilité de transfert de code d'une machine source à une machine destination et son exécution sur cette dernière. En effet, si les plateformes sont différentes, le code produit pour l'une ne peut fonctionner sur l'autre. Pour éviter ce problème, le code mobile est généré d'un langage source pour une machine virtuelle donnée (e.g. Machine virtuelle de Java :JVM).

Chaque plateforme doit disposer d'une couche logicielle qui implémente la machine virtuelle. Les machines virtuelles représentent une généralisation des middlewares dans le sens où elles n'offrent pas seulement quelques services communs mais offrent les mêmes services.

1.4.2 L'ouverture (*Openness*)

Elle désigne la possibilité d'ajout, de suppression ou de modification des ressources et services dans un système distribué.

L'ouverture nécessite que les interfaces logicielles soient documentées et accessibles aux développeurs d'applications (i.e. publiées publiquement).

Le défi que pose l'ouverture vient du fait que les composants d'un système réparti ont des origines diverses. Dans ce cadre, on qualifie un système d'*ouvert* s'il supporte, sans difficultés :

- ★ L'ajout d'ordinateurs au niveau de la couche matérielle
- ★ L'ajout de nouveaux services au niveau application, middleware et système d'exploitation
- ★ La réimplémentation des services anciens

Les systèmes ouverts présentent implicitement une indépendance vis-à-vis des constructeurs des plateformes et fournisseurs de produits logiciels.

1.4.3 L'invariance à l'échelle (*Scalability*), efficacité et disponibilité

Un système est dit invariant à l'échelle (scalable system) s'il reste performant et d'un coût raisonnable lors d'une augmentation importante du nombre des utilisateurs et de ressources gérées. Par exemple, Internet a connu une évolution exponentielle qui a fait passer le nombre d'ordinateurs de 188 en 1979 (début d'Internet) à 56 millions en 1999. De ce fait, il peut être considéré comme un système invariant au changement de l'échelle des utilisateurs et des ressources.

Avoir un système invariant à l'échelle signifie :

- ★ L'ajout d'utilisateurs et de ressources est toujours possible

- ★ Le coût d'ajout est proportionnel au nombre d'utilisateurs ou ressources ajoutés
- ★ Maintien de bonnes performances ou baisse raisonnable suite aux ajouts

1.4.4 Gestion de la sécurité

Résoudre le problème de sécurité revient à assurer :

- ★ La confidentialité des informations vis-à-vis des personnes non autorisées à la connaître
- ★ L'intégrité vis-à-vis des altérations ou corruptions (i.e. modifications mal intentionnées)
- ★ La disponibilité du système (i.e. pas d'interférence entre composants qui engendre un blocage du système ou détérioration des performances)

Dans un système réparti, l'échange de message nécessite leur protection et l'authentification de leur émetteur (e.g. cas du commerce électronique). La cryptographie et les mots de passe sont très utilisés actuellement. Cependant, malgré leur efficacité, ils ne permettent pas d'éradiquer complètement les problèmes épineux tels que l'attaque des services et la sécurité du code mobile.

1.4.5 Gestion de la concurrence

Dans les systèmes répartis, une ressource critique peut être utilisée simultanément par plusieurs processus physiquement répartis. Une manière simple de gérer cette concurrence consiste à créer un processus dit *allocateur de la ressource* qui accepte les demandes des processus dits clients (i.e. qui souhaitent utiliser la ressource) puis autorise un processus à la fois à utiliser la ressource. Cette solution est contraignante car le processus allocateur réduit les performances et peut tomber en panne.

Les applications réparties actuelles autorisent l'exécution de plusieurs services en concurrence (e.g. l'accès à une base de données). Chaque demande est prise en compte par un processus simple, dit thread, et la gestion de la concurrence fait appel aux mécanismes de synchronisation classiques tels que les sémaphores.

D'autres approches plus complexes existent, il s'agit d'établir une coordination entre les processus clients (dits alors processus pairs), à chaque demande, de telle sorte que l'un d'entre eux puisse accéder à la ressource au bout d'un temps fini.

Notons que la gestion de la concurrence dans un système distribué implique sa gestion aussi dans le cas d'un système centralisé car les parties impliquées dans un système distribué peuvent être des ordinateurs multitâches.

1.4.6 La transparence

En général, la transparence d'un mécanisme ou d'un concept signifie son existence et son utilisation implicite sans que l'utilisateur d'une application (qui peut être le concepteur de l'application) ne s'en rende compte. En d'autres termes, un concept est transparent dans un système si ce dernier se comporte, vu de l'utilisateur, comme si le concept n'y existait pas. Dans les systèmes distribués, la transparence signifie que la répartition des composants reste cachée pour l'utilisateur. Ce dernier perçoit le système distribué comme un tout et non comme une collection de composants indépendants. Dans ce contexte, on distingue huit formes de transparence :

- ★ Transparency d'accès : Il s'agit d'utiliser les mêmes opérations pour l'accès aux ressources distantes que pour les ressources locales.
- ★ Transparency à la localisation : Il s'agit de permettre une transparence d'accès aux ressources sans une connaissance préalable de leur localisation.
- ★ Transparency à la concurrence : Il s'agit de cacher à l'utilisateur la gestion de la concurrence en évitant principalement les interférences.
- ★ Transparency à la duplication : Dans le but d'accroître la fiabilité (implémenter d'autres formes de transparence) et améliorer les performances, on a souvent recours à la duplication de certaines ressources (e.g. fichiers de base de données). La gestion de cette duplication ne doit pas être perceptible par l'utilisateur.
- ★ Transparency aux pannes : Il s'agit de permettre aux applications des utilisateurs d'achever leurs exécutions malgré les pannes qui peuvent affecter les composants d'un système (composants physiques ou logiques).
- ★ Transparency à la mobilité : Il s'agit de permettre la migration des ressources et des clients à l'intérieur d'un système sans influencer le déroulement des applications.
- ★ Transparency à la reconfiguration : Dans le but d'améliorer les performances en fonction de la charge (i.e. les demandes de services des clients), il devrait être possible de reconfigurer un système sans que cela ne soit perceptible par l'utilisateur. Par exemple, les sites dits miroirs permettent d'améliorer les performances de l'accès à travers Internet, les déroutements vers ces sites doivent être non perceptible pour l'utilisateur.
- ★ Transparency à la modification de l'échelle (Scaling transparency) : C'est la possibilité d'une extension importante d'un système sans influence notable sur les performances des applications.

Remarque : Dans certains cas la transparence n'est pas souhaitable. Par exemple la duplication d'une imprimante améliore les performances, cependant l'utilisateur doit toujours avoir la possibilité de spécifier explicitement sur quelle imprimante il souhaite

imprimer ses documents. Cet aspect est particulièrement important si les imprimantes n'ont pas les mêmes caractéristiques (qualités) ou ne sont pas situées dans un même endroit.

1.4.7 Gestion des situations d'exception

Assurer la gestion des situations d'exception ou pannes fait intervenir plusieurs notions :

- ★ Détection : Certaines pannes ou erreurs peuvent être détectées facilement (cas des erreurs de transmission), par contre, d'autres pannes ne peuvent que être suspectées. Par exemple, la panne d'un processus serveur ou allocateur d'une ressource ne peut être détectée avec certitude. On ne peut conclure à une panne alors que le serveur n'est peut être que surchargé ou que les messages sont perdus. D'un autre côté, on peut attendre indéfiniment alors que le serveur est en panne. C'est un dilemme difficile à résoudre.
- ★ Masquage : Il s'agit de rendre les pannes autant que possible transparentes. Pour cela, différentes solutions peuvent être imaginées. Par exemple :
 - Retransmission des messages qui ne parviennent pas à destination
 - Duplication des ressources importantes pour les maintenir accessibles en cas de problèmes (cas de corruption d'un fichier)
- ★ Tolérance : Certains composants doivent être programmés (voire conçus) pour tolérer des pannes ou transmettre la détection des pannes aux utilisateurs (e.g. panne non masquable). Un composant est dit tolérant, s'il profite de/gère la redondance de ressource.
- ★ Reprise : La reprise après pannes désigne la possibilité de remise du système (principalement les données qu'il manipule) dans un état cohérent après la détection d'une panne.

1.4.8 Gestion des transactions réparties

ARCHITECTURE DES SYSTEMES DISTRIBUES

CONTENU

2.1 TAXONOMIE DES SYSTEMES DISTRIBUES

2.2 TAXONOMIE AU NIVEAU MATERIEL

2.3 TAXONOMIE AU NIVEAU SYSTEMES D'EXPLOITATION

2.3.1 Les systemes d'exploitation distribués

2.3.2 Les systemes d'exploitation réseaux

2.4 ARCHITECTURE DES APPLICATIONS DISTRIBUEES

2.4.1 Architecture Client/serveur et multitiers

Principe, rôle des processus, répartition des responsabilités, les variantes du modèle client/serveur (agent mobile, code mobile, proxy, ...). Les applications WEB et les bases de données, utilité des modèles deux tiers et trois tiers, scénario de mise en oeuvre

2.4.2 Le modèle Mandataire/Cache

2.4.3 Les architectures Pair-à-Pair

Inconvénients du modèle client/serveur, notions et avantages de processus pairs, problèmes de coordination

2.1 TAXONOMIE DES SYSTEMES DISTRIBUES

On vise par la taxonomie le partage des différents systèmes existants en catégories ou classes de telle sorte à faciliter leur compréhension et à parvenir aux concepts fondamentaux communs à chaque classe. La recherche d'une taxonomie n'est pas une tâche facile car les systèmes distribués sont complexes et variés. Nous proposons dans ce chapitre une taxonomie à trois niveaux qui reflète la structuration en couche présentée précédemment (figure 1.5). On répartie, dans le premier niveau, les systèmes distribués en considérant leur caractéristiques matérielles (couche matérielle). Dans le second niveau, les systèmes distribués sont vus sous l'angle des systèmes d'exploitation qui les supportent. Quant au troisième niveau, il est question de processus d'application et d'architecture. Il s'agit alors d'étudier les diverses approches de structuration (architecture) des applications distribuées en termes de composants (ou processus) et répartition des rôles.

2.2 TAXONOMIE AU NIVEAU MATERIEL

D'un point de vu abstrait, un ordinateur, se compose de deux types d'entités : les mémoires et les processeurs. On peut envisager un système distribué physique comme

une collection de mémoires et de processeurs interconnectés de telle sorte à pouvoir communiquer. L'interconnexion peut être faite de diverses façons en utilisant des technologies variées ce qui donne lieu au schéma de taxonomie de la figure 2.1. Quelque soit le système distribué considéré, il est possible de le classer dans l'une des quatre catégories de la figure 2.1.

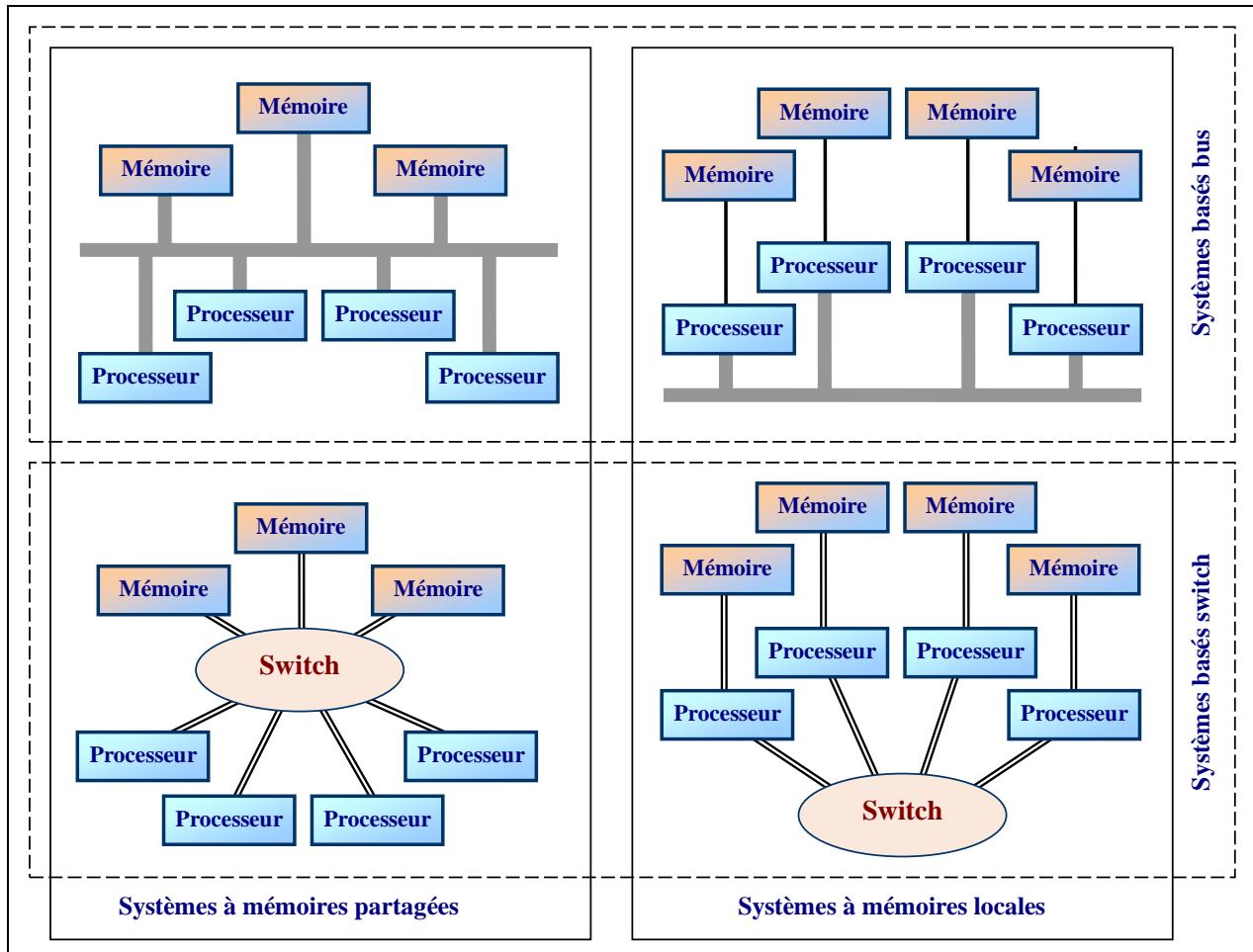


Figure 2.1. Interconnexion des processeurs et des modules de mémoire

Les systèmes à mémoires partagées sont souvent appelés *systèmes multiprocesseurs*. Les systèmes à mémoires locales, dits *systèmes multiordinateurs*, ne partagent pas de mémoires communes mais chaque processeur dispose de sa propre mémoire dont il est le seul à pouvoir y accéder.

Dans les *systèmes multiprocesseurs basés bus*, les processeurs et les modules de mémoire (un ou plusieurs) sont connectés à un bus commun de telle sorte que tous les modules de mémoire soient accessibles à n'importe quel processeur. Dans ce type de configuration, le bus constitue un goulet d'étranglement qui baisse considérablement les performances. Ce problème peut être réduit par l'utilisation de mémoires caches locales à chaque processeur. Mais, à leur tour, les caches doivent être maintenus cohérents, ce qui n'est pas une tâche facile.

Dans les *systèmes multiprocesseurs basés switch*, les processeurs et les modules de mémoire sont reliés par un dispositif de communication (réseau) utilisant des switchs (réseau de commutateurs). Lorsque le commutateur entre un processeur Pr₁ et un module mémoire M₁ est ouvert, Pr₁ peut accéder à M₁. Le nombre de switch peut être important engendrant un coût prohibitif. Pour réduire ce coût, il est possible d'envisager des réseaux avec des configurations diverses. Voir figure 2.2.

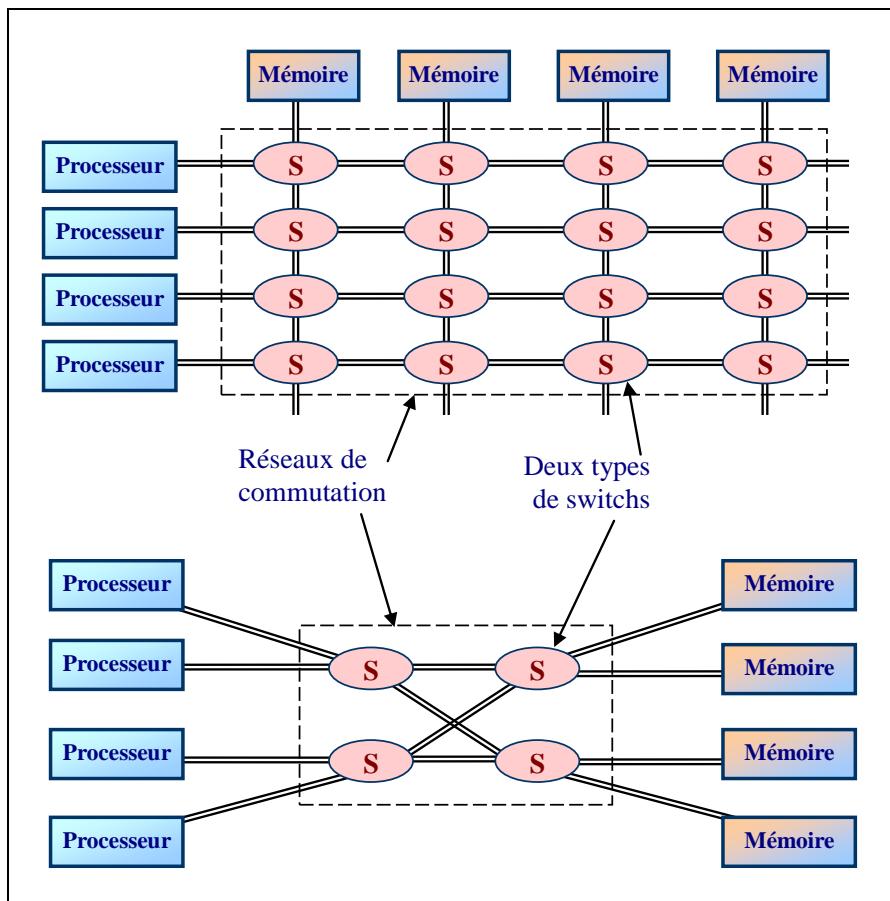


Figure 2.2. Exemples de réseau de commutation

La construction des systèmes à mémoires locales (multiordinateurs) est plus simple que celle des systèmes multiprocesseurs. Les *systèmes à mémoires locales basés bus* sont construits à partir d'ordinateurs (processeur + mémoire) identiques (on les qualifie de systèmes homogènes). Les processeurs sont reliés par un réseau multiaccès partagé (Tel que Fast Ethernet) et communiquent par diffusion de messages. Bien que la bande passante du réseau soit importante (de l'ordre de 100 Mbps), les systèmes ainsi construits ont une invariance à l'échelle limitée (25 à 100 noeuds).

Les systèmes basés switchs échangent des messages par routage à travers un réseau d'interconnexion pouvant avoir plusieurs topologies, allant des grilles simples aux hypercubes (voir figure 2.3). Les architectures en grilles, souvent présentées sur un seul circuit imprimé, conviennent pour les problèmes à deux dimensions (traitement d'images, théorie des graphes, etc.).

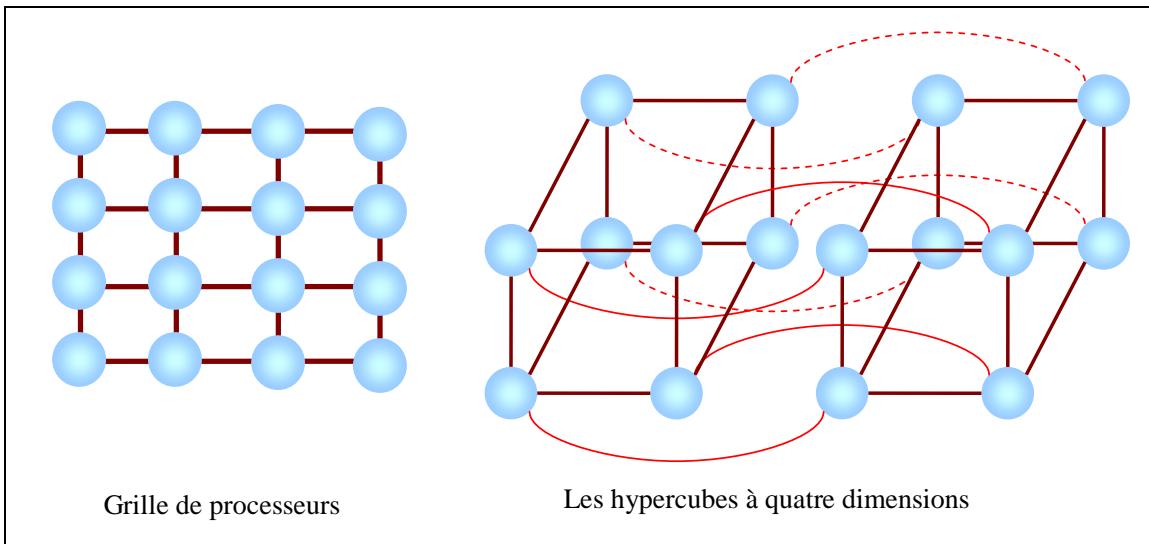


Figure 2.3. Exemples de systèmes basés switchs

Un hypercube est un cube à n-dimensions où chaque nœud (un processeur) est relié à n autres nœuds processeurs. Les hypercubes conviennent à la résolution de problèmes spécifiques tel que le calcul matriciel. Les systèmes basés switchs peuvent avoir des configurations très variées allant jusqu'à des superordinateurs massivement parallèles (MPP : Massively Parallel Processors) contenant des milliers de processeurs et dont le coût se chiffre en millions de dollars.

Les systèmes multiordinateurs hétérogènes sont les plus utilisés actuellement. Les ordinateurs peuvent avoir des structures et des performances nettement différentes et sont reliés par des réseaux hétérogènes.

2.3 TAXONOMIE AU NIVEAU SYSTEMES D'EXPLOITATION

Il existe une relation étroite entre les applications distribuées et les systèmes d'exploitation. Dans un premier lieu, la mise en œuvre des applications distribuées dépend des systèmes d'exploitation qui gèrent les différentes plateformes matérielles (i.e. les services qu'ils offrent). Dans un second lieu, les systèmes d'exploitation, eux-mêmes, peuvent être distribués (cas du système d'exploitation Chorus de Sun)

On peut diviser les systèmes d'exploitation en deux catégories : Les systèmes fortement couplés et les systèmes faiblement couplés. Dans le premier cas, le système d'exploitation essaye de maintenir une vue globale unique des ressources qu'il gère. Ce type de système a tendance à rendre la répartition physique transparente au niveau des applications. Dans le deuxième cas, on a affaire à une collection de plateformes où chacune dispose de son propre système d'exploitation mais ces derniers coopèrent pour rendre leurs services et leurs ressources disponibles les uns aux autres, il s'agit des systèmes d'exploitation réseau.

Notons qu'il est important de signaler que les systèmes d'exploitation constituent, eux aussi, des applications distribuées. Cependant, contrairement à ces dernières, ils sont implantés directement sur les plateformes matérielles et en dépendent fortement.

2.3.1 Les systèmes d'exploitation distribués

On en distingue deux types :

- ★ Les systèmes d'exploitation des plateformes multiprocesseurs (MPOS) (considérés comme des systèmes répartis particuliers)
- ★ Les systèmes d'exploitation multiordinateurs (MCOS).

Les MPOS sont conçus pour supporter les hautes performances en utilisant des processeurs multiples. Un des buts des MPOS est de rendre le nombre des processeurs transparent pour les applications. Dans ces systèmes, les processus communiquent via l'utilisation de données situées dans des emplacements de mémoire partagés. La protection de ces emplacements est faite par des sémaphores ou des moniteurs.

Les MCOS ont une structure totalement différente et complexe par rapport aux MPOS. Ceci est du au fait que les structures de données communes ne peuvent être simplement placées dans une mémoire physique partagée. L'unique moyen de communication et l'envoi de messages (voir figure 2.4).

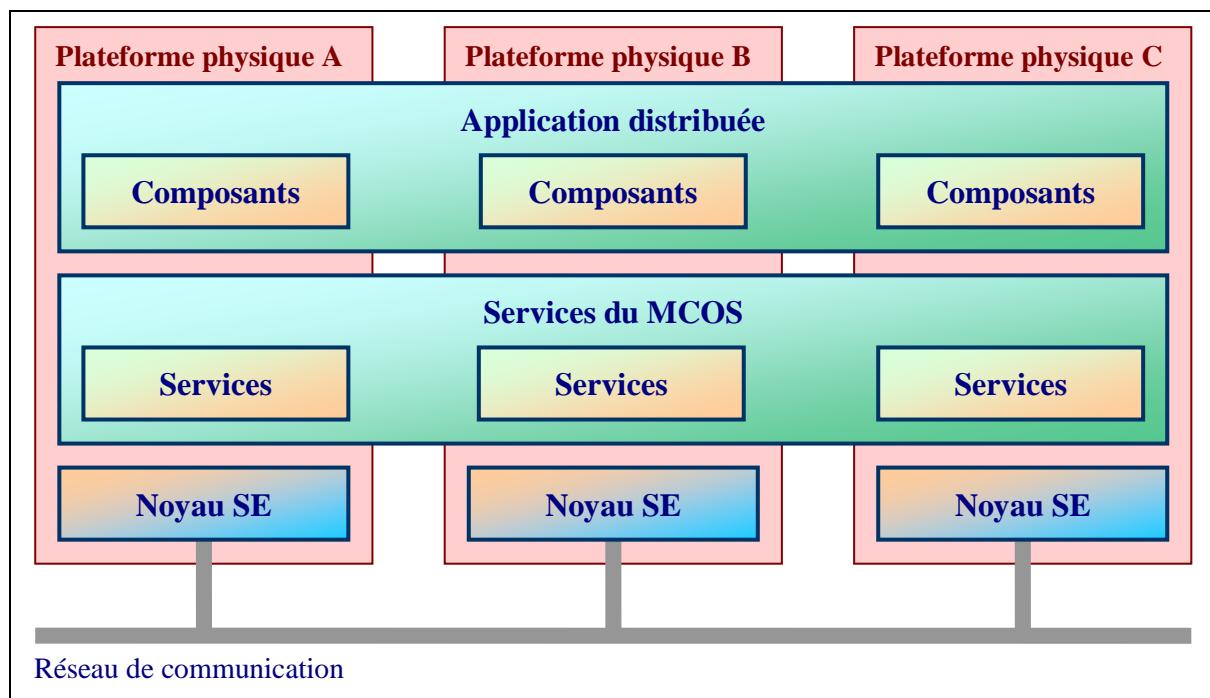


Figure 2.4. Positions des MCOS

La figure 2.4 peut être interprétée comme suit. Chaque nœud du réseau possède un système d'exploitation qui permet de gérer les ressources locales : mémoire, processeur, disque, De même, chaque nœud possède un module chargé de la communication entre plateformes (envoi et réception de messages). Sur chaque noyau

on greffe une couche commune qui implémente une machine virtuelle capable d'exécuter des tâches parallèles et concurrentes. Cette couche peut faire apparaître tout le système d'ordinateurs comme une machine multiprocesseur en implantant une mémoire partagée. D'autres services sont assignés à cette couche : affecter une tâche à un processeur, masquer les pannes matérielles, assurer la transparence à la localisation et la communication interprocessus.

La programmation des systèmes MCOS est beaucoup plus difficile que la programmation des systèmes MPOS. La raison est que l'utilisation d'une mémoire partagée avec une protection par sémaphores ou moniteurs est plus simple que la manipulation des messages. Cette constatation est derrière la solution qui consiste à créer des MCOS en modifiant les MPOS par l'émulation d'une mémoire partagée virtuelle à partir de la mémoire virtuelle de chaque nœud.

Par exemple, on peut utiliser la pagination et avoir une mémoire répartie partagée basée sur la pagination. Les pages sont alors réparties sur tous les nœuds et on maintient une table globale des pages qui indique l'emplacement des pages sur les nœuds. C'est essentiellement la pagination classique excepté qu'au lieu d'utiliser le disque local, on utilise la mémoire virtuelle distante. Lorsqu'un processeur génère une référence d'une page qui n'est pas présente localement, un déroutement à lieu, le MCOS cherche la page et la ramène dans la mémoire locale au processeur ayant généré la référence, ce dernier pourra alors continuer son exécution. La figure 2.5 donne un exemple d'état d'une mémoire virtuelle de 16 pages.

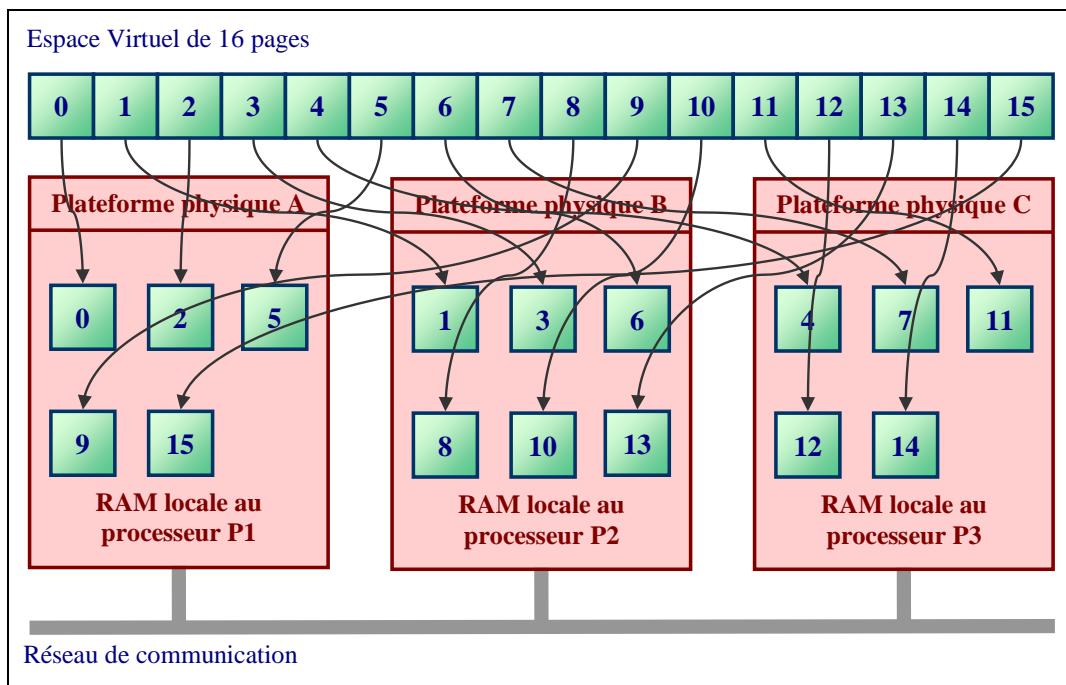


Figure 2.5. Une mémoire virtuelle partagée

Des améliorations peuvent être apportées à cette solution si on considère que certaines pages sont accédées uniquement en lecture et peuvent de ce fait être dupliquées.

2.3.2 Les systèmes d'exploitation réseau

Dans ces systèmes, on ne suppose pas que les plateformes matérielles sont homogènes et on ne cherche pas à faire apparaître l'ensemble comme un seul système. Il s'agit de réseaux où chaque nœud est une plateforme différente sur laquelle s'exécute un système d'exploitation différent (voir figure 2.6).

Les systèmes d'exploitation réseau offre divers services :

- ★ Connexion avec une machine distante. Il s'agit de faire apparaître une machine comme un simple terminal d'une autre (c à d uniquement envoyer et afficher des caractères).
- ★ Copie de fichiers d'un nœud à un autre (sans transparence).

Une amélioration courante de ces systèmes consiste à créer des serveurs qui cachent la répartition des fichiers sur les différentes machines et offrent un service de recherche et de transfert de fichiers aux machines qui sont alors considérées comme des clients.

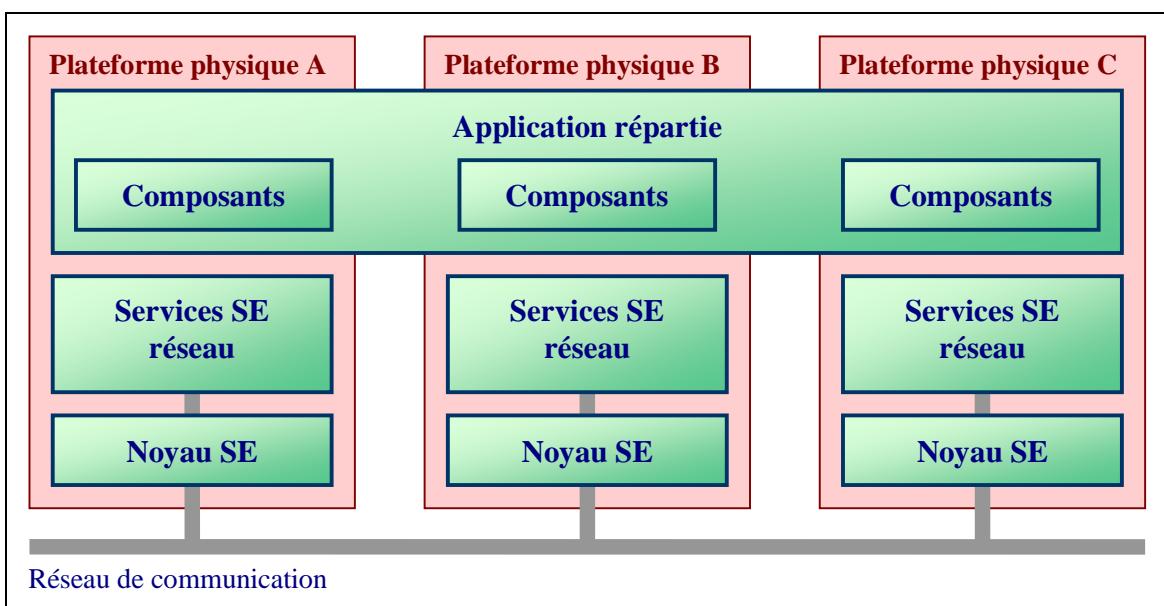


Figure 2.6. Les systèmes d'exploitation réseau

Un système d'exploitation réseau présente plusieurs inconvénients :

- ★ Manque de transparence (connexion explicite, copie de fichiers d'une machine à l'autre explicitement désignées).
- ★ Les machines étant indépendantes, elles sont gérées séparément (l'utilisateur d'une machine A à partir d'une machine B doit avoir un compte sur A et inversement, il faut alors gérer les mots de passes, les comptes et les droits accordés).

En contre partie, l'ajout d'une machine peut se faire librement et tout moment.

2.4 ARCHITECTURE DES APPLICATIONS DISTRIBUEES

Naturellement, les applications réparties ont plus d'indépendance vis-à-vis des plateformes physiques et peuvent de ce fait être organisées d'une multitude de façons. L'architecture client/serveur et ses variantes constituent actuellement les modèles le plus utilisés dans l'organisation des applications distribuées. Cependant d'autres modèles existent et leur utilisation augmente de jour en jour ; c'est le cas du modèle poste à poste (processus pairs) et ses variantes. Il n'est pas rare dans les applications distribuées que plusieurs modèles soient combinés à la fois pour tirer profit des avantages des uns et atténuer les inconvénients des autres. Dans la suite, nous présentons les différents modèles.

2.4.1 Architecture Client/serveur

C'est le modèle le plus utilisé et le plus important. Les processus représentant le système réparti, jouent les rôles de *client* pour un service donné et de *serveur* pour un autre. Par exemple, un navigateur Internet se comporte comme client lorsqu'il s'agit de récupérer une page Web. Un moteur de recherche est un serveur mais devient un client s'il déclenche d'autres moteurs de recherches sur d'autres sites Web. Actuellement, les moteurs de recherche typiques comportent plusieurs threads, certains servent les clients et d'autres se comportent comme client vis-à-vis des autres moteurs de recherche.

Dans le modèle Client/Serveur, on distingue deux sous modèles selon que le service est effectué par un ou plusieurs serveurs (voir figure 2.7). Dans ce dernier cas, plusieurs serveurs coopèrent pour exécuter une requête d'un client donné (e.g. Réservation de places d'avions pour une tournée impliquant plusieurs systèmes de réservation).

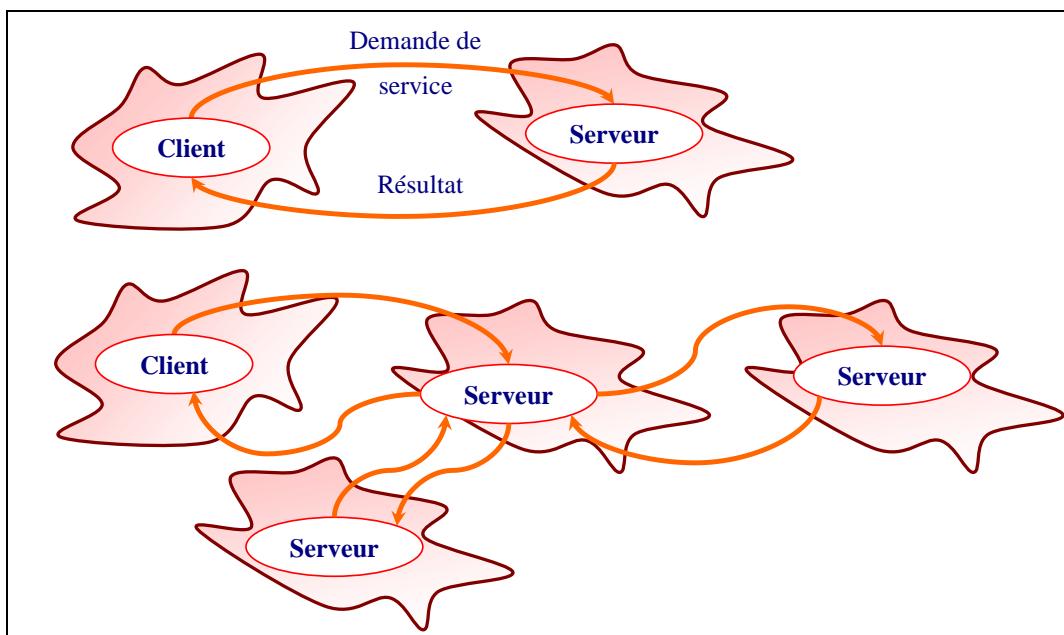


Figure 2.7. Un client/Un serveur Vs Un client/Plusieurs serveurs

2.4.1.1 Variantes de l'architecture Client/Serveur

Il existe diverses variantes du modèle Client/Serveur, nous les décrivons ci-après.

A. Le modèle Client/Serveur et le code mobile (le serveur vient chez le client) : Lorsqu'il s'agit de code mobile, le modèle Client/Serveur présente une particularité. En effet, au lieu que le serveur exécute un code et renvoi un résultat, il renvoi au client le code exécutable lui même (e.g. cas des applets Java). Ce modèle présente un avantage important, celui d'un temps de réponse meilleur, et un inconvénient important celui des risques que le code mobile engendre pour la sécurité du client. En effet, le code mobile peut être dangereux pour les ressources locales du client. Dans les cas des applets Java, les navigateurs ne les autorisent pas à accéder aux ressources locales du client.

B. Le modèle Client/Serveur et les agents mobiles (le client va chez le serveur) : Un agent mobile est un programme composé de code et de données, qui passent d'une plateforme à l'autre dans un réseau, dont le but de réaliser une tâche donnée. A l'arrivée, sur une plateforme donnée, l'agent invoque les services disponibles et utilise les ressources locales du serveur. Cette approche réduit le volume d'informations échangées sur un réseau et le temps de réponse. Les agents mobiles peuvent être utilisés pour :

- ★ La réalisation de tâches communes telles que comparaison d'offre de prix en visitant les sites des opérateurs économiques.
- ★ L'installation et la maintenance de logiciels
- ★ La répartition de charge où l'agent mobile migre vers les plateformes sous utilisées pour s'exécuter plus rapidement (solution testée au centre de recherche PARC de Xerox)

Comme pour le code mobile, les agents mobiles peuvent présenter un danger pour celui qui les accueille. Pour cela, il faut prévoir un accès restreint aux ressources et l'authentification de celui qui mandate l'agent (l'agent doit préserver secrètement l'information d'authentification). D'un autre coté, les agents mobiles sont vulnérables et ne peuvent continuer leurs tâches (survivre), si l'accès aux ressources locales leur est interdit.

C. Le modèle Client/Serveur avec clients limités : En général, le client dispose de données locales, d'un système d'exploitation et d'autres logiciels qui sont parfois difficiles à gérer et nécessite des utilisateurs qualifiés. Pour remédier à cela, on peut alléger le client de deux façons :

- ★ Client sans logiciels. La machine cliente ne dispose au départ que d'une application limité qui doit télécharger un système d'exploitation et les logiciels ou composants nécessaires à partir d'un serveur de fichier distant.

Les parties téléchargées s'exécutent sur la machine cliente mais les fichiers sont gérés par le serveur de fichiers distant.

- ★ Client interface : Dans cette approche, le client dispose d'une couche logicielle qui se contente de jouer le rôle d'une interface d'affichage. Les applications sont exécutées sur le serveur qui doit être une machine multiprocesseur puissante.

Les approches qui visent à limiter les capacités des clients ont l'inconvénient de produire un système dont le temps de réponse est long spécialement lorsqu'il s'agit des applications de conception assistée par ordinateur. Notons que dans la pratique, il existe une panoplie de modèles où le côté client d'une application peut avoir plus ou moins de responsabilité. Ceci est discuté dans les paragraphes suivants.

2.4.1.2 Architecture Client/serveur et répartition des rôles

Dans leur grande majorité, les applications visent le support de l'accès des utilisateurs à une base de données. De ce fait, on distingue trois niveaux différents dans une application Client/Serveur :

- ★ Niveau Interface utilisateur. La partie cliente dans une application implémente, souvent, l'interface utilisateur. Ce niveau permet de gérer l'interaction de l'utilisateur avec l'application. L'interface utilisateur peut être très simple comme elle peut être très sophistiquée. Dans le cas où l'interface ne fait que gérer l'affichage de caractères et leur saisie par un clavier, l'appellation Client/Serveur n'est pas très appropriée. Actuellement, la majorité des applications assignent au clients au moins l'affichage graphique et diverses fonctionnalités utilisant la souris.
- ★ Niveau Données. Le niveau Données dans une application Client/Serveur contient les programmes qui maintiennent les données de l'application. Les données à ce niveau sont persistantes. Dans le cas le plus simple il s'agit d'un système de fichiers, mais, souvent, c'est des bases de données complètes qui matérialisent le niveau Données. Dans le cas le plus général du modèle client/serveur, les données sont du côté du serveur.
- ★ Niveau Traitement de l'application dit aussi logique métier. Le niveau traitement dépend des applications. Il consiste en un ensemble de fonctionnalités qui se situent entre le niveau Interface et le niveau Données.

La figure 2.8 illustre ces niveaux à travers un exemple. Il s'agit de l'accès à un moteur de recherche à travers le WEB. Le client n'est autre qu'un navigateur Internet qui affiche les pages Web et transmet les requêtes de l'utilisateur en utilisant le protocole HTTP.

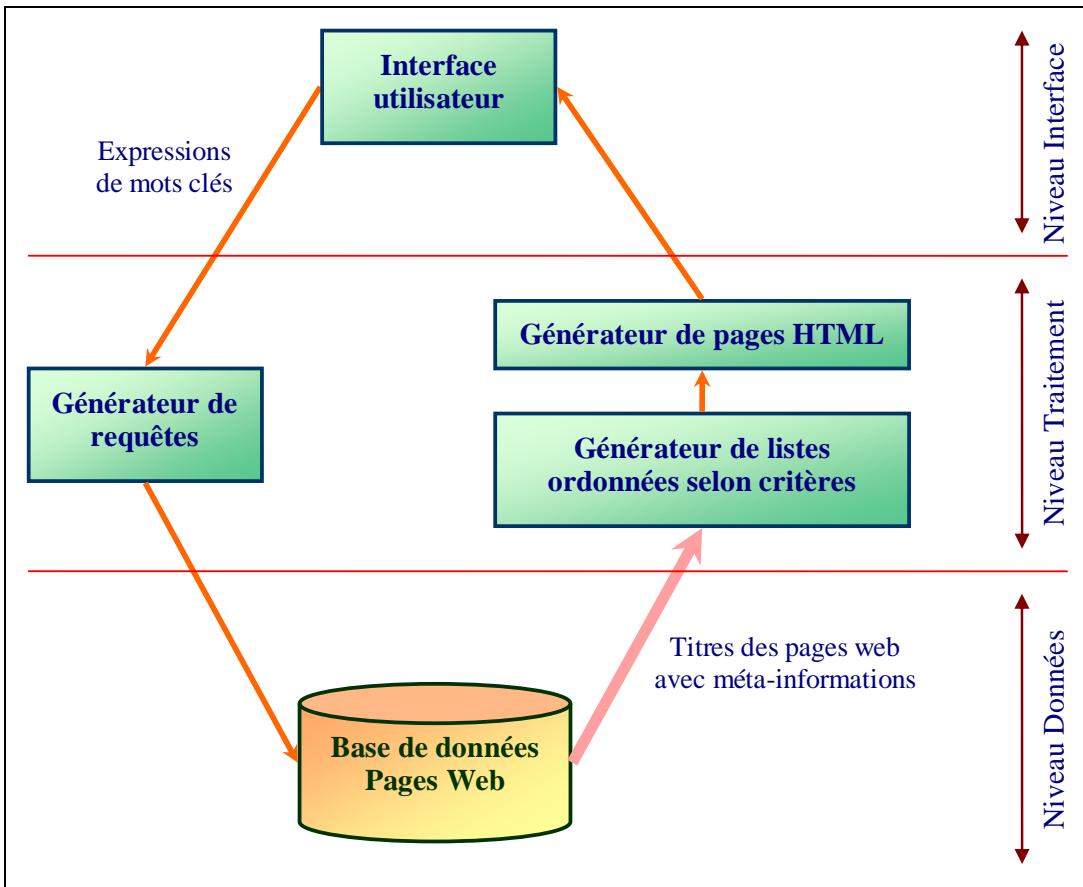


Figure 2.8. Un moteur de recherche sur le Web

2.4.1.3 Architectures multitiers

La répartition des rôles discutés précédemment, suggère différentes possibilités de répartition d'une application qualifiées de répartition multitiers (Multitiered en anglais).

Le terme Client/Serveur a été traditionnellement associé à la configuration matérielle qui consistait en un microordinateur connecté à un serveur SQL de base de données. Le terme désigne, donc, un modèle de partage où les tâches sont réparties entre des couches clientes et serveurs dites tiers.

L'architecture *un tier* correspond aux applications classiques des ordinateurs centraux (mainframe) où des terminaux permettent à des utilisateurs d'interagir avec une application monolithique qui effectue des traitement (logique métier), gère des bases de données et communique avec l'utilisateur.

Dans les architectures *deux tiers*, l'application est partagée en deux parties qui, naturellement, résident sur deux plateformes distinctes. Le client accède directement au serveur de la base de données et les traitement peuvent être du coté client comme du coté serveur de la base de données sous forme de procédures.

Lorsque les traitements ont lieu du côté client, on a affaire à des clients lourds (fat client). Le serveur gère la base de données et reçoit des requêtes SQL qu'il exécute et renvoi un ensemble de données résultats au client.

Plus l'application est importante, plus le client est lourd ce qui nécessite une plateforme support performante ce qui est un inconvénient important.

Lorsque les traitements résident du côté du serveur, on parle de serveur lourd. Les clients ne font qu'invoquer les procédures stockées dans le serveur de bases de données. Du point de vue performances, la configuration où le serveur est lourd est meilleure que la précédente car la communication entre client et serveur est moins importante.

Entre client lourd/léger et serveur lourd/léger, il existe une multitude de variantes que nous illustrons par le schéma de la figure 2.9.

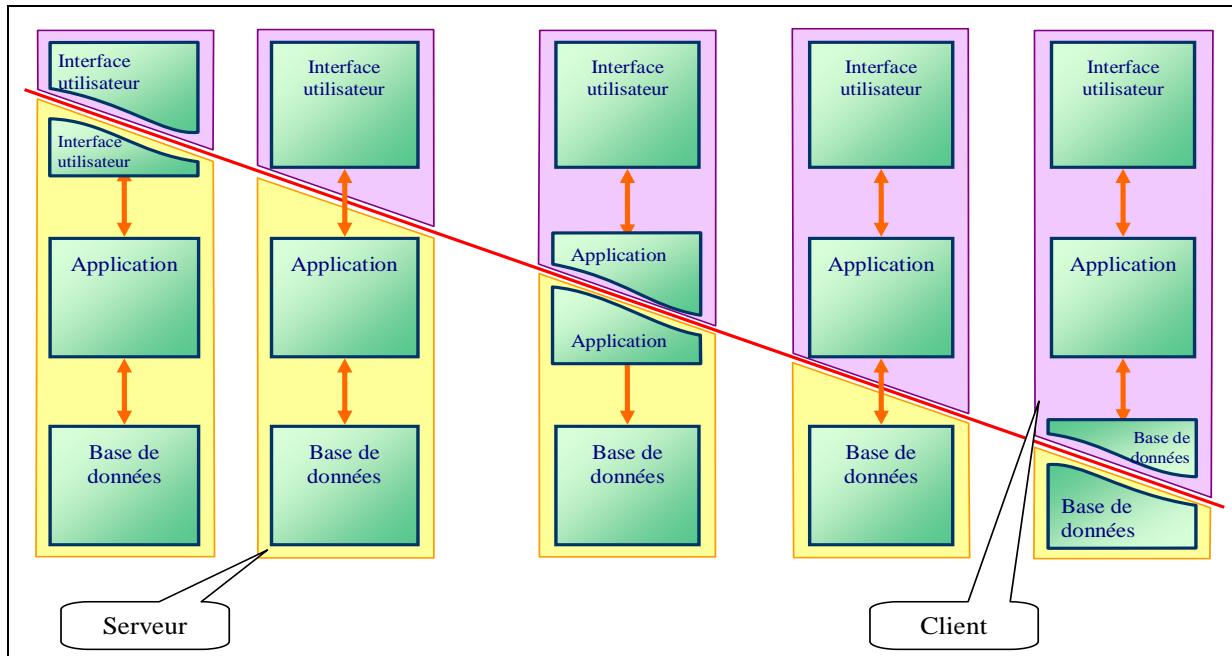


Figure 2.9. Architecture deux tiers

Dans cette figure, on constate l'existence de cinq cas :

- 1- La plateforme du côté client joue le rôle de terminal
- 2- L'application contient un minimum concernant l'interface. L'interface est une couche graphique qui communique avec l'application
- 3- Une partie de l'application est dans la plateforme cliente. Par exemple vérification des formulaires (consistance) ou édition de textes sur la plateforme cliente et des fonctions avancées sur le serveur.

4- Configuration très courante. Le client est un microordinateur ou une station connecté à travers un réseau à une base de données ou un système de fichiers. Toute l'application s'exécute sur la plateforme cliente mais les opérations sur la base se font sur le serveur.

5- Ce cas correspond à un maintien local au client d'une partie des données.

Généralement l'architecture deux tiers n'est pas invariante à l'échelle. Au-delà d'une centaine de clients, les performances chutent considérablement.

Dans les configurations plus récentes, un *tiers du milieu* est ajouté entre le client et le serveur de la base de données. On a alors affaire à des architectures *trois tiers* où le premier tiers n'est autre qu'un client léger qui implémente une logique de présentation, le second tiers dit *serveur d'application*, implémente la logique métier et le troisième est un serveur de bases de données. Dans un environnement donné, il est possible de trouver plusieurs serveurs d'application et plusieurs serveurs de bases de données.

Le tier du milieu implémente, en plus de la logique métier, des opérations de translation qui convertissent les demandes des clients en requêtes de base de données et convertissent les résultats de ces requêtes selon le format utilisé par le client.

Souvent, le tiers client interagit avec le serveur de l'application moyennant un protocole standard tel que le RPC (Remote Procedure Call) ou HTTP. A son tour, le tiers du milieu interagit avec le serveur de bases de données en utilisant un protocole standard tel que SQL, ODBC JDBC.

Cette configuration, permet une meilleure invariance à l'échelle et l'utilisation de protocoles standards permet de créer une indépendance entre les tiers ce qui permet à chacun d'évoluer plus facilement : évolution de la logique métier, changement de la base de données en choisissant un autre fournisseur sans altérer les autres tiers, etc.

L'architecture multitiers utilise plusieurs tiers du milieu au lieu d'un seul tiers milieu lourd. Ces architectures sont actuellement très utilisées et permettent de supporter des applications volumineuses en les décomposant en couches plus simples à développer et à maintenir. N'importe quelle application client/serveur peut être implémentée avec une architecture multitiers où la logique métier est décomposée en plusieurs serveurs.

Comme avantages de l'architecture multitiers, on cite :

- ❖ Chaque tiers est plus ou moins indépendant des autres ce qui lui permet d'évoluer sans trop de contraintes. En effet, la concentration de la logique métier dans les tiers du milieu permet la modification de celle-ci sans

nécessiter le changement d'une multitude de tiers clients qui peuvent être physiquement éloignés.

- ★ Réduction importante du volume d'informations échangé sur le réseau (on échange uniquement les informations nécessaires à la réalisation d'un service), ce qui améliore les performances.
- ★ La base de données peut être partagée par plusieurs utilisateurs ayant chacun sa logique métier. Ceci est possible par l'utilisation de plusieurs serveurs d'applications différents.
- ★ Possibilité de répartir la charge du tiers milieu sur plusieurs plateformes physiques.
- ★ Possibilité de dupliquer les serveurs de l'application et les serveurs des bases de données.
- ★ Développement et maintenance plus aisée des applications distribuées.

L'utilisation d'une architecture trois/multitiers n'exclu pas les architectures un/deux tiers. Si pour une application réduite, un ou deux tiers sont convenables, pour une application importante l'architecture multitiers conviendrait davantage.

2.4.2 Le modèle du Mandataire/Cache

Un cache est un espace mémoire qui maintient une copie des objets récemment utilisés proches, vis-à-vis du client, que les objets originaux. Un objet reçu est ajouté au cache remplaçant éventuellement un objet existant. Lorsqu'un client demande un objet, le gestionnaire du cache essaye d'abord de le trouver dans le cache et le transmettre au client. Si l'objet n'est pas dans le cache, le gestionnaire transmet la demande au serveur qui détient l'objet.

Le cache peut être géré par le client lui-même comme il peut être géré par un gestionnaire indépendant dit Serveur Mandataire (appelé aussi Proxy). Dans ce dernier cas, le cache peut être utilisé par plusieurs clients (voir figure 2.12).

Par exemple, dans le Web, les mandataires maintiennent des caches des pages récemment visitées mais avant de livrer une page à un client, le mandataire vérifie au moyen d'une requête spéciale, du protocole HTTP, si la page qu'il a est conforme à l'originale.

Les mandataires permettent d'augmenter les performances en diminuant le temps de réponse. Les mandataires peuvent implémenter un protocole de sécurité tel que les pare-feu (Firewall). Le modèle du mandataire est souvent utilisé comme modèle complémentaire avec d'autres modèles (i.e. combiné avec les autres).

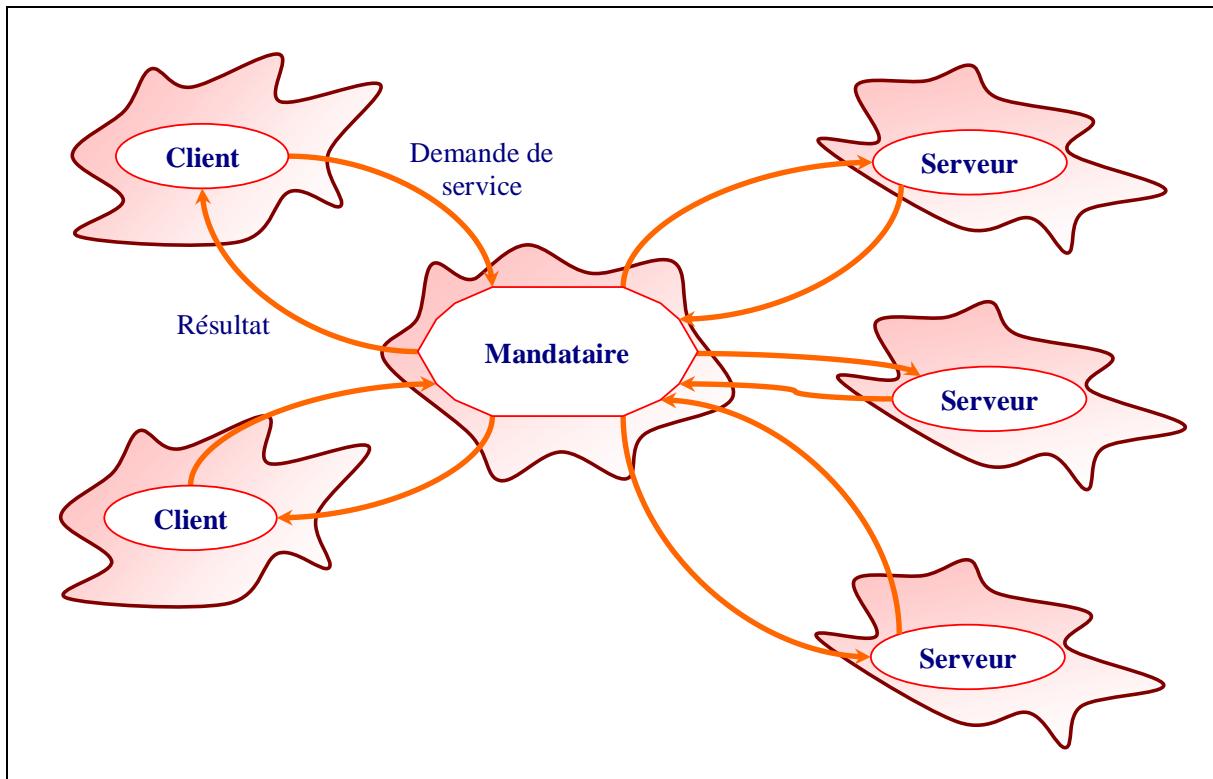


Figure 2.12. Le mandataire comme intermédiaire entre clients et serveurs

2.4.3 Architecture pair-à-pair (Peer To Peer)

Dans ce type d'architecture, il n'existe pas de distinction, en terme de clients et de serveurs, entre les composants (processus) d'un système distribué. Les processus jouent des rôles similaires et coopèrent d'égal à égal pour réaliser une activité répartie (Voir figure 2.13).

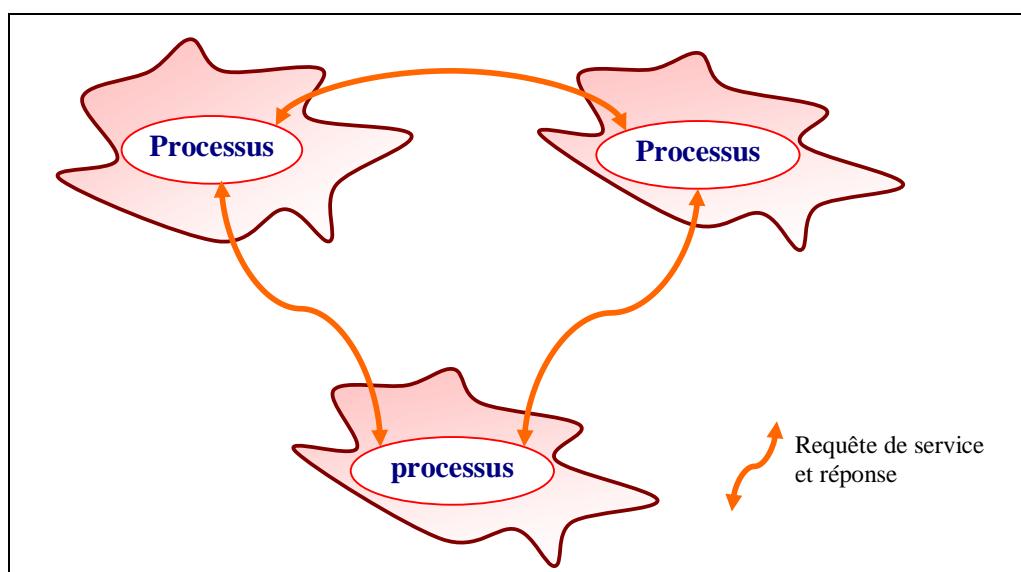


Figure 2.13. La coopération des processus pairs

Le terme *Peer-to-peer* (abrégé en *P2P*), qu'on peut traduire par *pair-à-pair* (*poste à poste ou encore égal à égal*) désigne tout système où les participants (les pairs) mettent en partage des ressources locales (qui peuvent être des capacités de traitement, des fichiers, des espaces de stockage, des moyens de communication, etc.) sans utilisation de serveurs spécifiques.

Les participants partagent les ressources locales en établissant des communications directes entre eux moyennant les protocoles TCP/IP. Ainsi, chaque participant est à la fois un client et un serveur. Il est *serveur* de ce qu'il possède et souhaite partager et *client* de ce que les autres mettent à sa disposition. Le P2P désigne donc une classe d'applications qui tirent profit des ressources matérielles ou humaines qui sont disponibles sur le réseau Internet.

L'infrastructure support des applications peer-to-peer comprend principalement des réseaux utilisant les protocoles TCP/IP, protocoles symétriques qui ne font aucune distinction entre client et serveur, et des composants logiciels particuliers qui remplissent, à la fois, les fonctions de client et de serveur. Ces derniers sont parfois appelés *servents* (de la contraction de serveur et de client, due à Gnutella), ou, plus communément mais de façon réductrice, *clients*. Les servents peuvent matérialiser toute l'application, et sont alors en interaction directe avec l'utilisateur, comme ils peuvent ne constituer qu'une couche offrant des services à diverses applications. La figure 2.14 donne un aperçu sur l'infrastructure décrite.

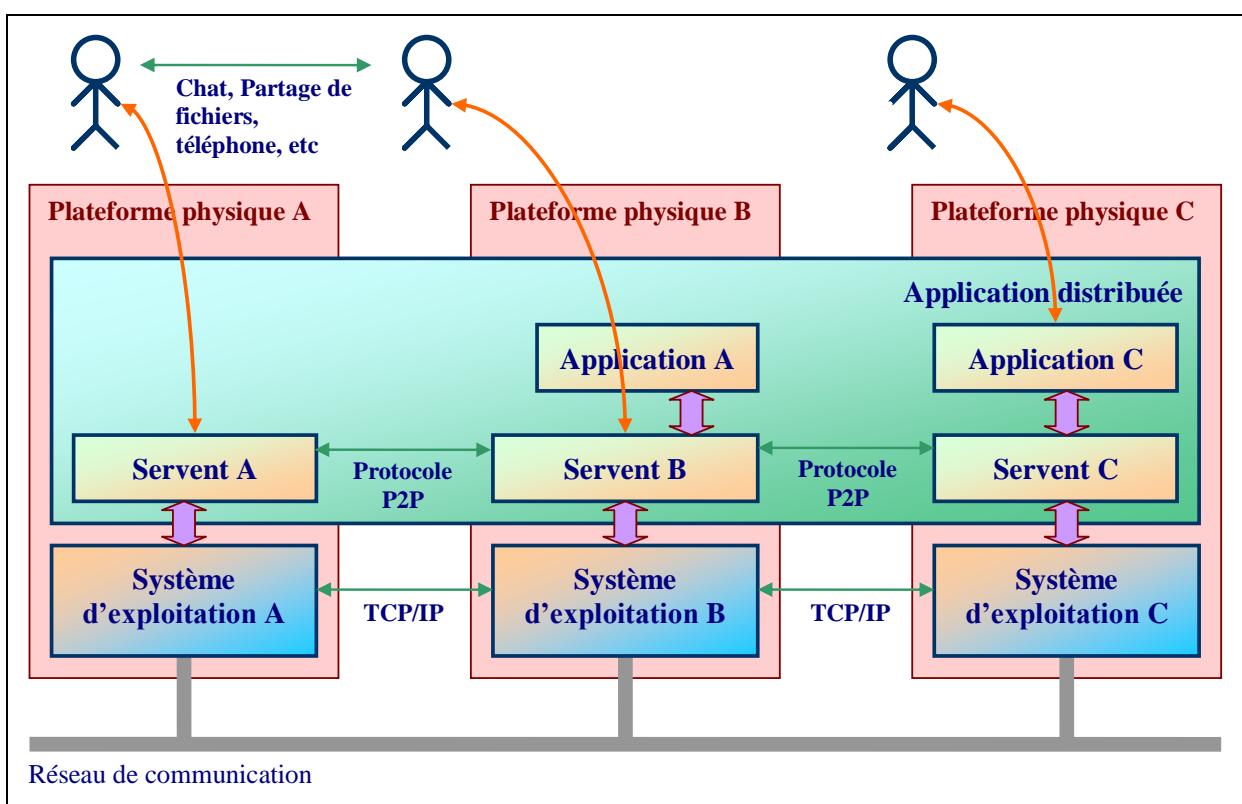


Figure 2.14. Infrastructure des systèmes P2P

2.4.3.1 Avantages des systèmes Pair-à-pair

Bien que l'utilisation dominante, des systèmes P2P, soit, actuellement, le partage de fichiers, le modèle pair-à-pair va bien plus loin que ce simple partage. En effet, il est possible de décentraliser des services et de mettre à la disposition des autres participants des ressources. Vu que chaque utilisateur d'un réseau pair-à-pair peut proposer des ressources et en obtenir, les systèmes pair-à-pair permettent de faciliter le partage d'informations et rendent la censure ou les attaques des services plus difficiles.

En général, les systèmes P2P présentent certains avantages par rapport aux systèmes client-serveur :

- ★ Réduction des coûts. Au lieu d'acquérir un nouveau matériel pour construire une infrastructure client-serveur, il est possible d'utiliser les plateformes existantes avec une approche P2P ce qui évite des dépenses supplémentaires et réduit le coût d'administration. A titre d'exemple, un réseau de stockage P2P évite le recours aux serveurs de stockages centraux tout en offrant des performances meilleures.
- ★ Invariance à l'échelle (passage à l'échelle): La distribution/duplication des ressources et services permet de répartir l'information sur l'ensemble du réseau P2P et évite l'apparition de goulots d'étranglements. Ce qui permet une bonne invariance à l'échelle. Par exemple, certains systèmes d'échange de fichiers tel que Napster (échange de fichiers MP3) pouvaient servir plus de six millions d'utilisateurs simultanément.
- ★ Réseaux ad hoc : L'approche P2P est convenable pour les réseaux ad hoc où la connexion est intermittente. L'utilisateur est libre de se connecter ou se déconnecter au réseau pour des durées quelconques. L'approche P2P est ainsi convenable pour des applications où un contrôle centralisé n'est pas envisageable. Cas de l'informatique diffuse, l'informatique mobile, etc.
- ★ La fiabilité : Vu que le bon fonctionnement d'un système P2P ne dépend pas d'un participant spécifique mais donne une importance égale à chaque chacun des participants, la fiabilité se trouve améliorée. En effet, la panne d'un nœud n'a pas d'influence sur le système dans son ensemble, ce qui n'est pas le cas lors de la panne d'un serveur dans une architecture client-serveur.

Ces avantages font des systèmes pair-à-pair des outils privilégiés pour décentraliser des services devant avoir une haute disponibilité et des coûts de maintenance faibles. Il s'agit des services telsque : la diffusion de données multimédia, la distribution des logiciels et leurs mises à jour, la communication téléphonique et messagerie, la gestion des noms des domaines (DNS), etc.

2.4.3.2 Les inconvénients des systèmes P2P

L'approche P2P présente des inconvénients importants aussi. En effet, les problèmes de sécurité ou de comptabilité sont plus simples à résoudre dans un système doté d'un serveur central. De même, la disponibilité des ressources n'est pas toujours garantie lorsque les participants aux systèmes P2P sont peu nombreux. La rupture de la connexion par un participant peut rendre une ressource non accessible si elle n'est pas disponible chez d'autres participants. C'est typiquement le cas dans l'échange de fichiers.

2.4.3.3 Les variantes du modèle P2P pour le partage de fichiers

Les variantes des architectures P2P peuvent être classées en quatre catégories : architecture centralisée, architecture décentralisée, architecture hiérarchique et architecture en anneau. Dans la pratique, ces architectures sont souvent combinées pour avoir des systèmes distribués P2P hybrides.

A- Architecture centralisée : le peer-to-peer assisté

Dans une architecture centralisée, il doit y exister un serveur qui se charge de mettre en relation directe tous les participants connectés. Le serveur maintient une base de donnée centrale qui consiste en un index de tous noms des fichiers, que chaque participant met à la disposition des autres, couplés avec les adresses IP des participants qui les possèdent. La base ne contient à aucun moment les fichiers eux-mêmes mais seulement leurs intitulés. La mise à jour de la base se fait à chaque fois dès qu'un participant se connecte ou se retire du réseau. La figure 2.15 illustre cette architecture.

Pour télécharger un fichier, l'utilisateur soumet à l'aide du servent une requête comportant les mots clés pouvant apparaître dans le nom du fichier. Le serveur répond avec une liste de noms accompagnés des adresses IP correspondantes. L'utilisateur dispose alors pour chaque fichier d'une ou plusieurs adresses IP. Il ne lui reste qu'à établir (en utilisant son servent) une connexion directe avec le servent possédant le fichier recherché et le télécharger.

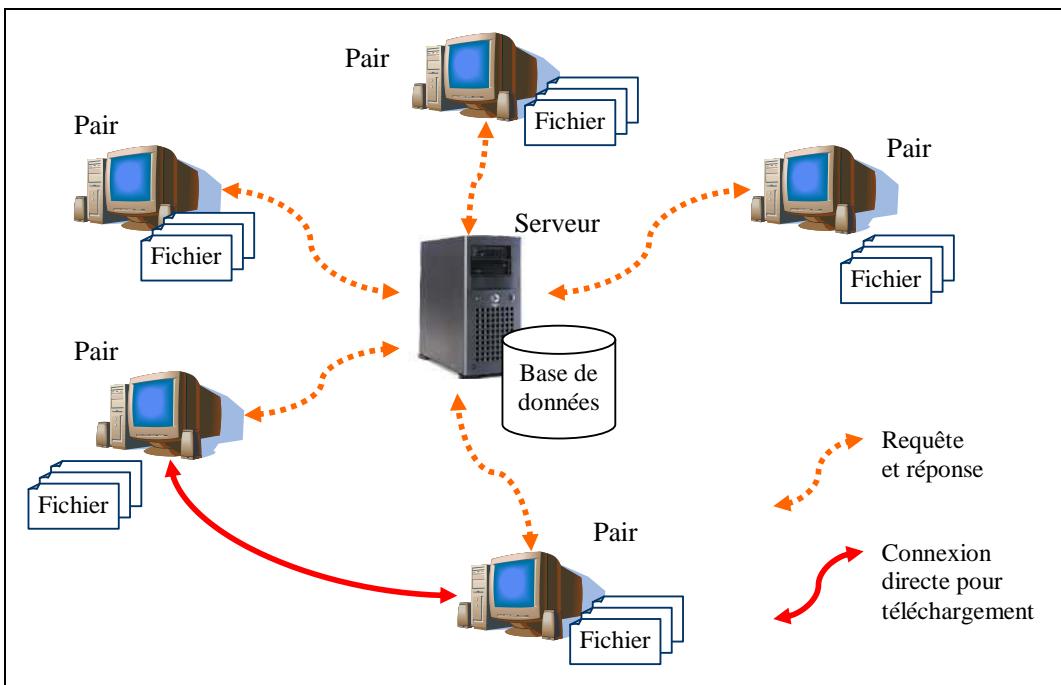


Figure 2.15. Architecture centralisée

Comme pour l'architecture client-serveur, le point faible de l'architecture P2P centralisée réside au niveau du serveur. En effet, en cas de panne de ce dernier aucun échange ne peut avoir lieu. De même, le serveur peut être souvent sollicité ce qui réduit les performances (bande passante du serveur). La présence des adresses IP sur le serveur ne permet pas un échange totalement anonyme des ressources.

B - Architecture en anneau

Dans le but d'accroître la fiabilité de l'architecture centralisée, le serveur central est remplacé par un anneau virtuel de serveurs. La figure 2.16 donne un aperçu sur cette architecture.

Chaque serveur de l'anneau maintient des informations sur les participants qui lui sont connectés et échange ces informations avec les autres serveurs. Ainsi, en cas de panne d'un serveur le système reste opérationnel. Avec une telle approche, on constate une meilleure disponibilité des serveurs et une répartition de la charge qui préserve les performances (répartitions des demandes de connexions et requêtes qui préserve la bande passante. L'architecture en anneau est souvent utilisée lorsque les machines sont relativement proches (appartenant souvent à la même organisation).

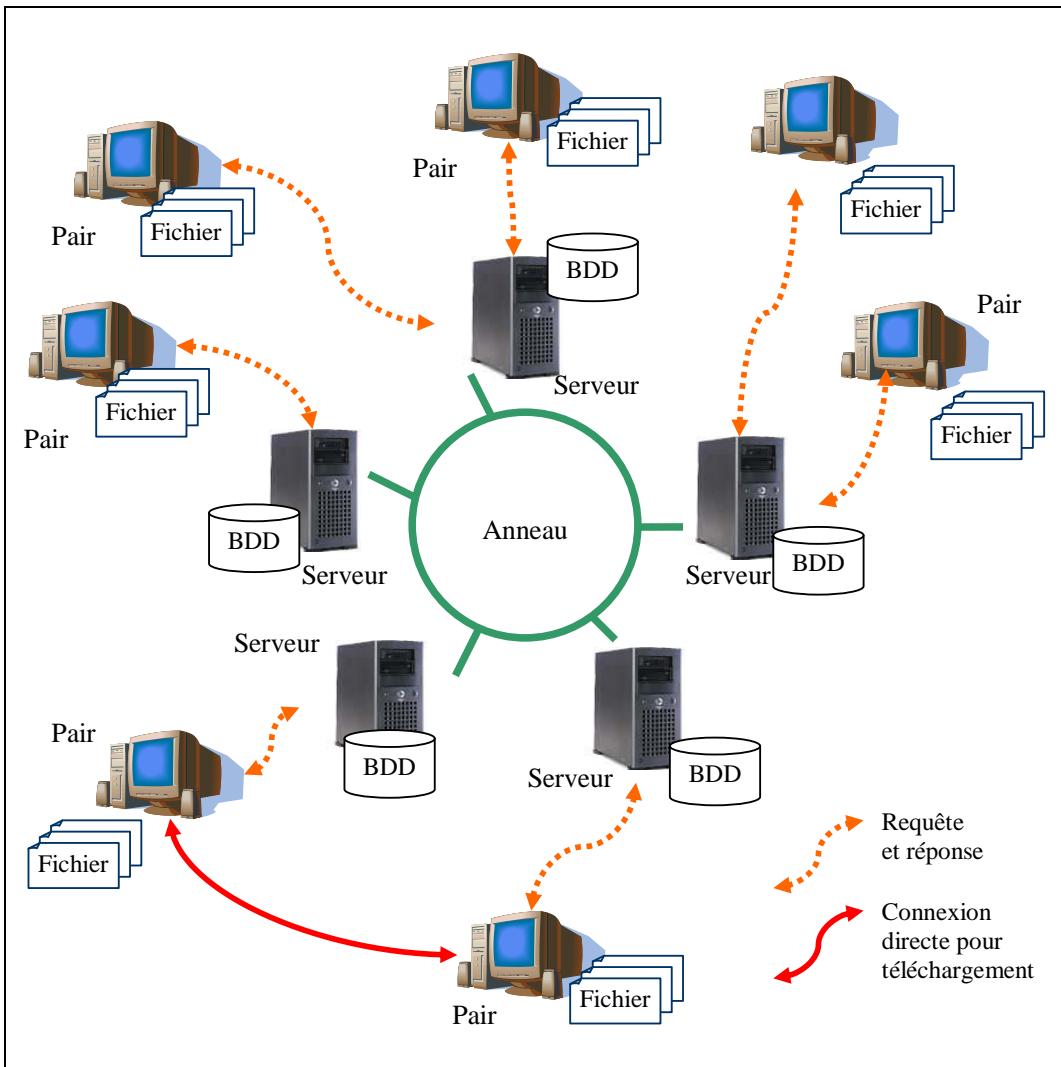


Figure 2.16. Architecture P2P en anneau

C – Architecture hiérarchique : Au lieu d’organiser les serveurs en anneau, l’architecture hiérarchique élabore plutôt une hiérarchie qui réduit le volume d’informations échangées entre les serveurs, ce qui préserve la bande passante. La figure 2.17 illustre ce principe.

D – Architecture décentralisée (Le P2P pur) : Dans ce type d’architecture, il n’existe pas de serveurs. Tous les participants ont le même statut. Pour rejoindre le réseau, un participant P doit d’abord diffuser un message spécial sur le réseau physique. Les pairs directement proches de P et qui reçoivent ce message renvoient leurs adresses IP. Vu qu’il n’y a pas de serveurs, les requêtes de P seront diffusées à ces voisins et ses derniers les diffusent à leurs voisins immédiats et ainsi de suite. Ce qui engendre un trafic important sur le réseau physique et réduit ainsi les performances. Les architectures décentralisées présentent l’avantage d’être insensibles aux pannes des pairs. La figure 2.17 illustre l’approche.

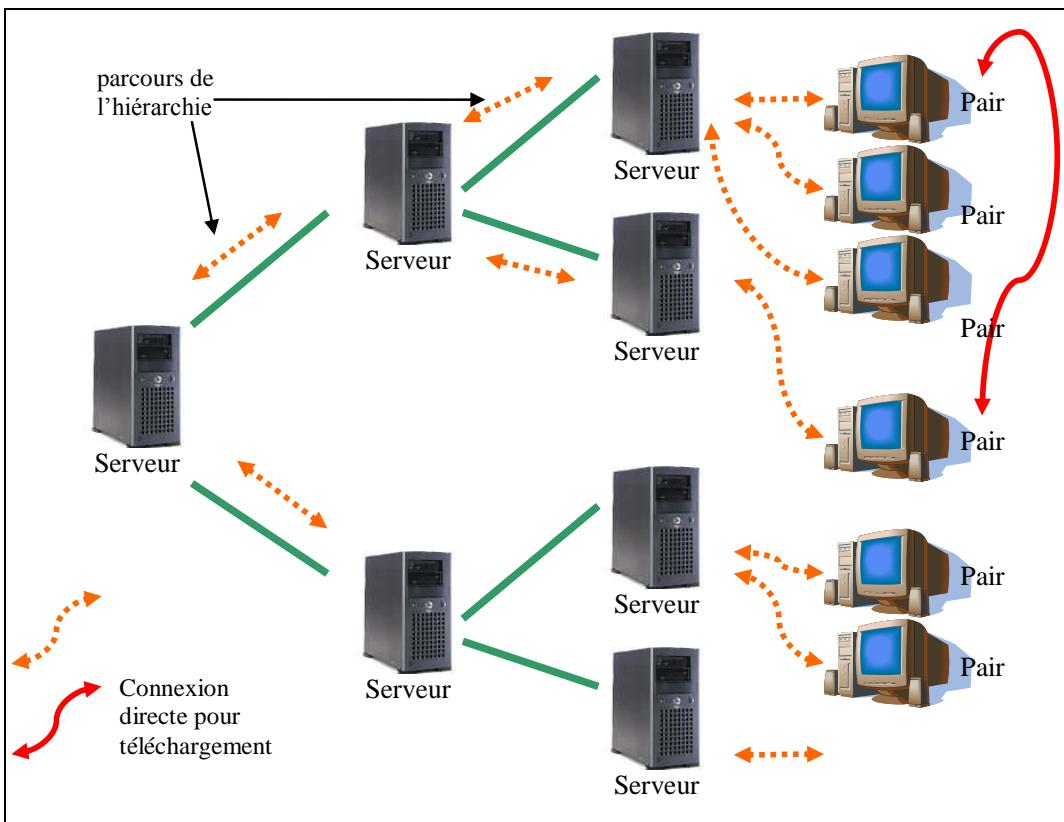


Figure 2.16. Architecture P2P hiérarchique

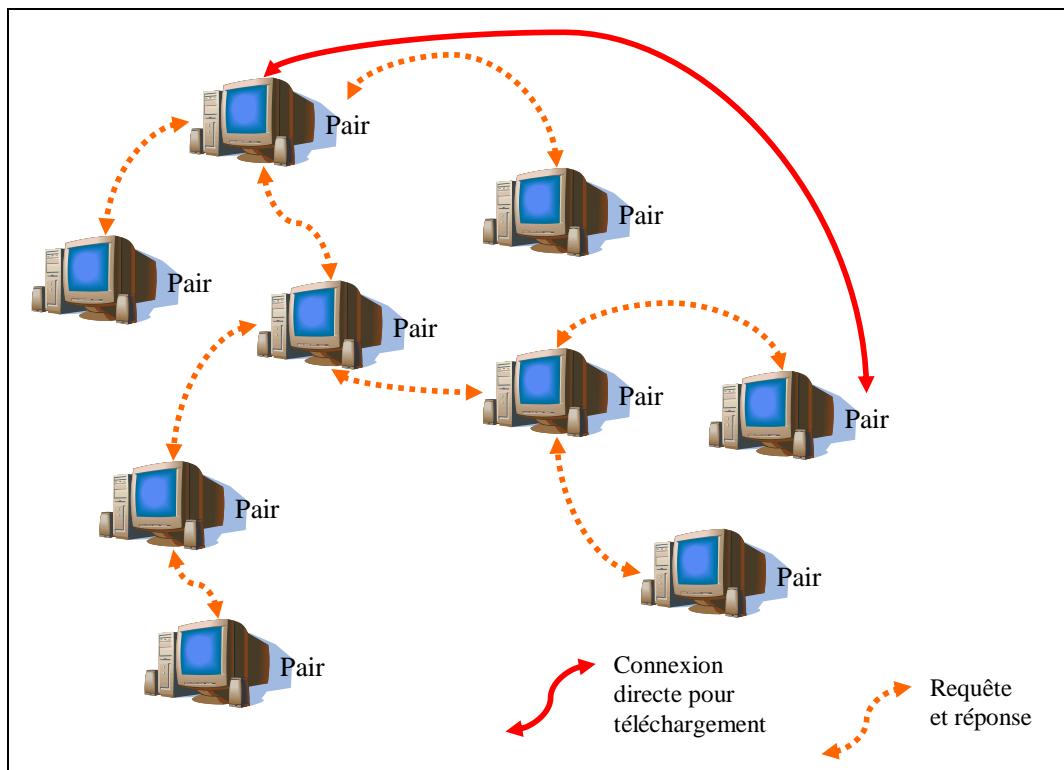


Figure 2.17. Architecture P2P décentralisée

2.4.3.4 Quelques exemples de systèmes

e-Mule : e-Mule est la version actuelle de e-Donkey qui est né en septembre 2000. Il adopte une architecture centralisée avec une multitude de serveurs et permet le transfert de tous types de fichiers en utilisant des servents pour les plateformes windows et Unix.

Chaque utilisateur peut créer son propre serveur et les serveurs sont reliés entre eux. Lors de sa connexion avec l'un des serveurs, le pair fournit la liste des fichiers qu'il souhaite mettre à la disposition des autres. Après connexion, le pair a la possibilité de soumettre une requête de recherche au serveur auquel il est connecté ou à tous les serveurs dont les adresses lui sont fournies par les autres pairs. Les fichiers partagés ne transitent pas par les serveurs, ces derniers se contentent de maintenir des index composés de noms de fichiers et d'adresses IP.

e-Mule utilise au niveau bas le protocole UDP et au niveau des servents le protocole MFTP (Multisource File Transfert Protocol) qui est une variante du FTP où les fichiers sont décomposés en blocs et les téléchargements se font par blocs. Il est ainsi possible de télécharger les blocs d'un fichier de différentes sources, ce qui améliore grandement les performances.

Gnutella : Il s'agit d'un réseau P2P décentralisé qui a évolué au fil du temps d'une architecture semblable à celle de e-Donkey vers une architecture P2P pure qui se caractérise par l'absence de serveurs. Pour se connecter au réseau, le servent diffuse un message particulier sur Internet (dit PING) pour récupérer les adresses IP et quelques informations sur les données partagées des servents les plus proches (Ces derniers renvoient des messages dits PONG). Une fois la connexion établie (Récupération des adresses IP), le servent envoie des trames de recherche aux différents pairs et récupère, par des trames réponses, les noms des fichiers partagés. Il ne reste plus qu'à télécharger directement le fichier sélectionné du pair qui le détient. Comme pour e-Mule, Gnutella utilise le protocole MFTP.

Divers servents Gnutella existent pour les plateformes Windows, Unix et Macintosh :

- ★ **Pour Windows :** Gnutella Clients, BearShare, Gnucleus, Morpheus, Shareaza, Swapper, XoloX, LimeWire, Phex
- ★ **Pour Unix et Linux :** Gnutella Clients, Gnewtellium, Gtk-Gnutella, Mutella, Qtella, LimeWire, Phex
- ★ **Pour Macintosh :** Gnutella Clients, LimeWire, Phex

LES PARADIGMES DE COMMUNICATION

CONTENU

3.1 Le passage de messages

Qu'est ce q'un Message?, les primitives de communications, primitives bloquantes et non bloquantes, les primitives de passage de messages bufférissés

3.2 Le RPC (Remote Procedure Call)

Principe, mécanismes et concepts utilisés, paramètres et résultats dans le RPC, l'édition des liens, exemple

3.3 Le RMI (Remote Method Invocation)

Principe, mécanismes et concepts (interfaces, classes, stub, ...), Java RMI

3.4 Communication par évènements et notifications

Principe, émetteur/récepteur, abonnement, notification

3.5 Communication de groupe

Principe, structure d'un groupe, opérations sur les membres (ajout, suppression, recherche, ...), élaboration de la communication (diffusion, ordonnancement des messages, ...)

3.6 Communication par mémoire partagée

Principe, mécanisme d'utilisation, maintien de la mémoire (création, protection, ...)

3.7 Communication par flux

L'objet d'une communication est de faire parvenir une information transmise par un processus à un autre processus. Cette information peut être utilisée à différentes fins : synchronisation, demande de service, renvoi de résultats, communication d'un état, etc.

On distingue principalement deux types de communication : La communication par message et la communication par mémoire commune. Cette dernière étant utilisée le plus souvent dans le cas des systèmes multiprocesseurs et sa mise en œuvre est plutôt triviale.

Il existe d'autres outils de communication plus évolués qui font appel à la communication par message dans leur mise en œuvre. Il s'agit de l'appel de procédure à distance (Remote Procedure Call ou simplement RPC), de l'invocation de méthode à distance (Remote Method Invocation ou simplement RMI) et de la notification d'événements. Dans ce chapitre, nous décrivons ces outils.

3.1 LE PASSAGE DE MESSAGES

La communication par message est supportée par deux primitives de communication qu'on peut nommer *Send* et *Receive*. Un processus P communique avec un processus

Q en lui envoyant une séquence d'octet au moyen de la primitive Send. Q reçoit cette séquence en utilisant la primitive Receive.

<pre>Process P ; Begin ... Send (M, Q) ; // M est la séquence d'octet ... End ;</pre>	<pre>Process Q ; Begin ... Receive (Mes, P) ; ... End ;</pre>
---	---

L'implémentation de Send et Receive soulève deux problèmes particuliers : la synchronisation et l'établissement d'un canal de communication. Nous les discutons dans ce qui suit.

3.1.1 Problème de la synchronisation lors de la communication

Avant de pouvoir échanger une information, les processus doivent, souvent, se synchroniser d'abord (i.e. l'un attend l'autre). On distingue plusieurs possibilités selon que les primitives Send et Receive soient bloquantes ou non pour le processus qui les exécute. Lorsque Send et Receive sont toutes les deux bloquantes, on parle de communication synchrone. Dans ce cas, si un processus exécute Send avant que l'autre n'exécute le Receive, le processus émetteur reste bloqué en attente et inversement. Le déblocage aussi bien de l'émetteur que du récepteur, intervient lorsque le processus receveur reçoit le message et son exécution est relancée.

Dans la forme asynchrone de communication, le Send n'est pas bloquant pour l'émetteur. Il consiste à mettre le message dans un buffer et le système d'exploitation se chargera de le faire parvenir à destination. Le processus émetteur continue alors son exécution sans attendre que le message parvienne à sa destination. Si le buffer dédié à la communication est plein le Send devient une opération bloquante jusqu'à ce que une place se libère. Autrement dit, le processus ne continue son exécution que lorsque le système d'exploitation prend en charge l'envoi du message.

Le Receive peut être bloquant ou non. Dans le cas où le Receive est non bloquant, le receveur continue son exécution après l'opération Receive qui consiste, alors, à spécifier seulement un buffer qui accueillera le message. Une fois le buffer rempli, le Receveur sera informé par interruption ou par scrutination. Le Receive non bloquant est difficile à mettre en œuvre et est peu utilisé car il rend la logique de programmation complexe.

3.1.2 Problème du canal de communication

Un canal de communication est défini par la paire (Expéditeur, Destinataire). Une solution simple serait la spécification d'un canal de communication en utilisant les noms des processus. Cette approche présente l'inconvénient de ne pas convenir aux systèmes répartis car, on ne connaît pas préalablement les noms des processus, de plus, ces processus peuvent provenir de programmes qui ont été développés indépendamment

les uns des autres. Autrement dit chaque processus peut avoir un espace de nommage différent non connu au moment de la compilation des programmes des autres processus. Même si ces noms sont connus au moment de la compilation, il faut aussi connaître leur équivalent après compilation pour pouvoir mettre les processus en correspondance. Ce qui complique encore les choses, c'est le fait qu'un processus serveur est souvent développé pour servir plusieurs clients préalablement inconnus.

Dans les systèmes actuels, le canal de communication consiste en une paire de *Socket* (qu'on peut traduire littéralement par Prise), un pour l'émetteur et un pour le récepteur. La communication consiste, alors, à transmettre des messages entre les deux sockets. C'est le cas dans les systèmes BSD Unix, Windows et MacOS.

Un socket n'est autre qu'un port et une adresse Internet. L'adresse Internet spécifie une machine et le port spécifie un point (i.e. numéro) d'émission ou de réception dans celle-ci. Les ports sont identifiés par des numéros entiers sur 16 bits et leur nombre peut atteindre 2^{16} pour une machine donnée. Les ports sont, simplement, des numéros qui correspondent à des buffers de communication au niveau du système d'exploitation. Un port peut être associé, au plus, à un processus receveur mais peut avoir plusieurs émetteurs.

Pour qu'un processus reçoive des messages, son socket doit spécifier un port local et une des adresses Internet de la machine sur laquelle il s'exécute. Les processus utilisent les mêmes sockets pour l'émission et la réception (voir figure 3.1).

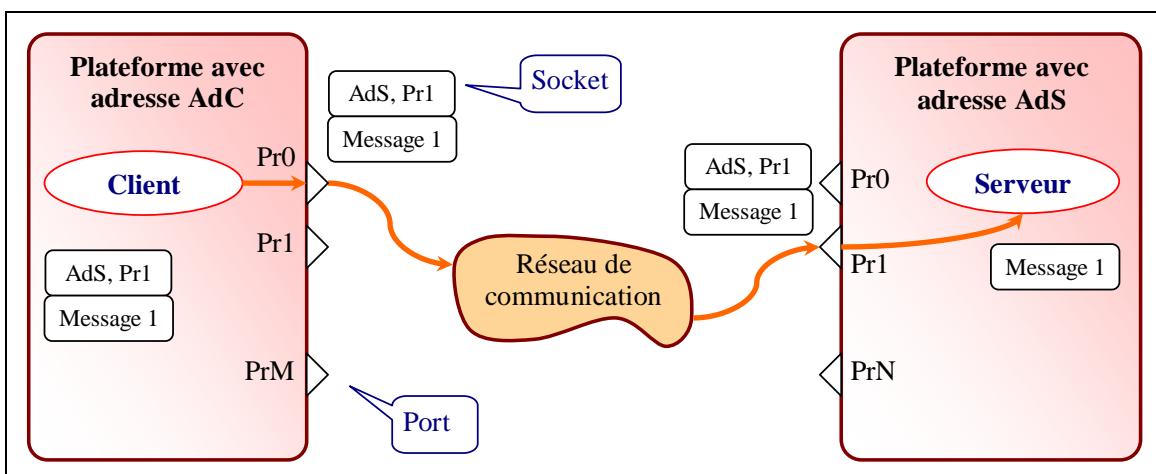


Figure 3.1. Emission et réception par sockets

3.2 LE RPC (REMOTE PROCEDURE CALL)

Une des formes de communication dans un système centralisé et l'appel de procédure. Un processus appelle une procédure en lui transmettant des paramètres celle-ci s'exécute et met à jour éventuellement un espace mémoire. Un autre processus peut, ensuite, appeler cette même procédure ou une autre pour récupérer/utiliser le contenu

de l'espace mémoire en question. L'idée du RPC est simple, elle consiste, pour un processus P, à appeler une procédure se trouvant sur une autre plateforme, au lieu qu'elle soit sur la même plateforme que P (i.e. dans son espace accessible de nommage).

Le RPC peut être énoncé comme suit : Un processus P sur un site S1 donné appelle une procédure M sur un autre site S2 en lui transmettant éventuellement des paramètres. Suite à cet appel, P est suspendu et M est exécutée sur le site distant S2. Le transfert d'information est concrètement effectué par les paramètres de S1 vers S2 et par le résultat que la procédure retourne du site S2 vers S1.

Malgré la simplicité de l'idée, le mécanisme RPC est difficile à mettre en œuvre car :

- ★ L'émetteur et receveur s'exécutent sur deux sites différents, donc deux espaces d'adressage différents et il faut transmettre les paramètres et le résultat.
- ★ Les deux sites peuvent tomber en panne.

Dans l'implémentation du RPC, l'objectif est toujours d'offrir un mécanisme qui rende l'appel distant semblable à l'appel local (i.e. on cherche à garantir la transparence du mécanisme pour le programmeur).

Pour bien cerner le mécanisme du RPC, considérons un exemple d'un appel local de procédure :

```
Process P ;
Begin
  ...
  Cpt := Lire (F, Tampon, NbrOctet); /* F est un fichier
local, Tampon est destiné à contenir les informations lues et
NbrOctet est la taille de l'information à lire */
  ...
End ;
```

Considérons maintenant l'état de la pile d'exécution avant et pendant l'appel (figure 3.2).

On remarque que F, NbrOctet sont passés par valeur (c'est leurs valeurs qui sont transmises à la procédure et occupent un espace dans l'espace pile de celle-ci). A l'opposé, Tampon est passé par référence, en d'autres termes, c'est juste une adresse qui est transmise à la procédure. Cette adresse indique l'espace pile du programme appelant ou éventuellement un espace mémoire quelconque contenant la valeur de Tampon.

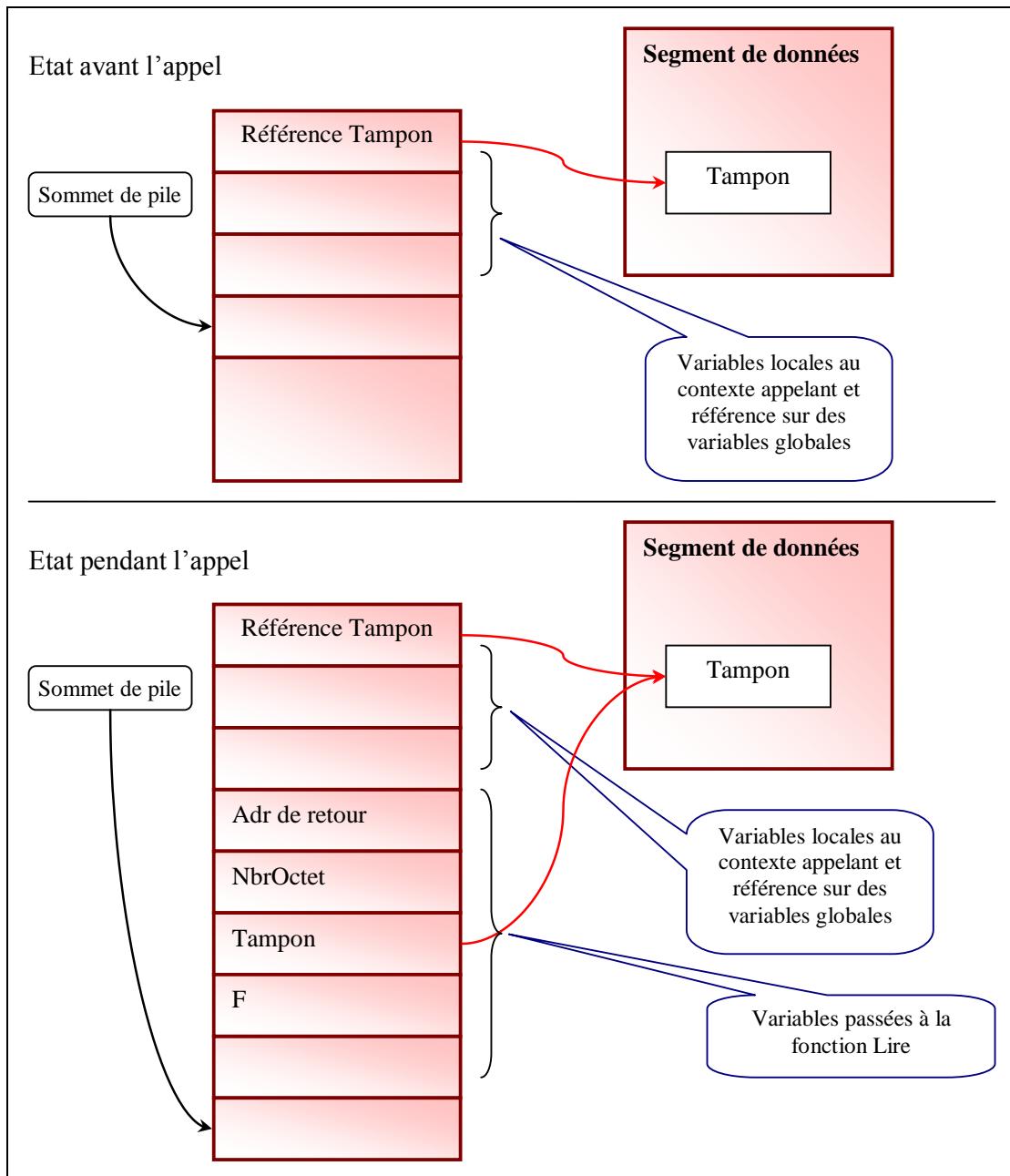


Figure 3.2. Evolution de la pile d'exécution lors d'un appel de procédures

Un appel local nécessite l'intervention du programmeur, du compilateur et de la plateforme d'exécution :

- ★ Le programmeur place une instruction d'appel à une procédure dans son code.
- ★ Le code de la procédure est extrait de la librairie puis inséré dans le programme objet par le compilateur (c'est l'édition des liens).
- ★ L'exécution de l'appel par le processus appelant bloque ce dernier jusqu'à la fin de la procédure.

Pour réaliser une implémentation transparente du RPC, le compilateur insère dans le code objet du client non pas le code de la procédure appelée mais sa *souche* (on parle de *souche du client ou Client Stub*). Celle-ci se trouve normalement dans la librairie qui accompagne le programme.

Contrairement à la procédure originale, la souche ne vas pas s'exécuter et produire le résultat mais elle se contente de construire un message à partir des paramètres et l'envoi au serveur puis exécute un *Receive* bloquant qui bloque la souche, donc le processus appelant, pour attendre la réception du résultat du serveur (voir l'illustration à la figure 3.3).

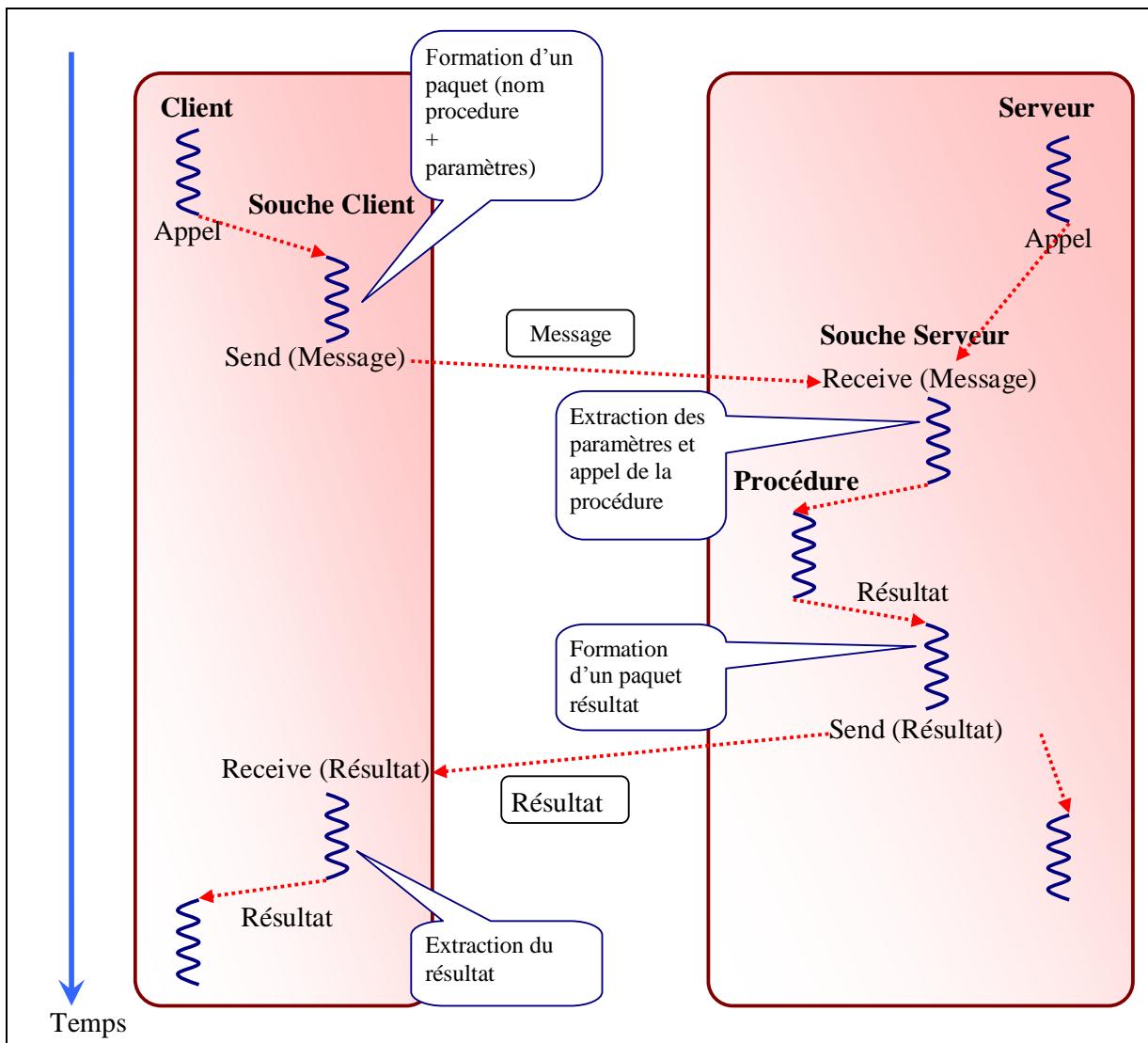


Figure 3.3. Communication entre client et serveur par RPC

A l'arrivée du message, le système d'exploitation ne le passe pas directement à la procédure appelée mais à sa souche du coté serveur (on parle de souche serveur ou Server stub). La souche du serveur a, préalablement, exécuté un *Receive* et s'est bloquée pour attendre le message de la souche client. Une fois ce message arrivé, la souche serveur extrait les paramètres puis appelle la procédure du serveur.

Si le serveur supporte plusieurs procédures accessibles à distance, la souche du serveur doit comporter une instruction de choix multiple. L'appel effectué par la souche serveur de la procédure en question est un appel local dont les paramètres sont des variables initialisées par les valeurs reçues dans le message.

Il demeure un problème difficile qui est celui des paramètres passé par références. La solution consiste à transformer l'appel par référence en un appel par copie/restauration. Ce dernier consiste à transmettre le contenu du paramètre passé par référence à la souche du serveur qui le place à son tour dans un emplacement accessible à la procédure appelée. Cette dernière le modifie puis la souche du serveur transmettra une copie à la souche client qui lui permet de restaurer la valeur du paramètre passé par référence.

Une optimisation peut être envisagée selon que le serveur ne fait que modifier la copie ou ne fait que consulter la copie. On peut alors éliminer soit l'envoi de la copie soit l'opération de restauration.

Remarques :

- ★ Certaines structures telles que les arbres sont difficiles à transmettre. Il est possible de faire un va et vient entre les souches pour acquérir les éléments des structures complexes selon le besoin.
- ★ La génération des souches se fait suite à une description d'interface (contenant les entêtes des procédures accessibles à distance) dans un langage spécifique souvent appelé IDL (Interface Definition Language). Le programmeur du côté client n'a besoin que de cette interface pour spécifier des appels aux procédures dans ces programmes (i.e. processus). Du côté serveur, le programmeur doit implémenter les différentes procédures définies dans l'interface. D'un côté comme de l'autre, les souches sont générées automatique.
- ★ Dans le cas où une procédure ne retourne aucun résultat, le serveur peut retourner avant l'exécution de la procédure un accusé de réception de la demande qui libère la souche client et par la suite le processus client. Dans ce dernier cas, on parle de RPC asynchrone.

3.3 LE RMI (REMOTE METHOD INVOCATION)

Le RMI est une approche spécifique au paradigme orienté objet où l'application est vue comme une multitude d'objets et où chacun communique avec les autres en appelant leurs procédures ou méthodes (on parle d'invocation de méthode et souvent de passage de message entre objet). L'invocation de méthode est souvent une opération bloquante.

Vu que les applications basées objets sont logiquement partitionnées en objets, la répartition de ces objets sur différents processus et plateformes (ce qui donne une application répartie) est considérée comme une extension naturelle de l'approche basée objets.

Dans une architecture Client/Serveur, les objets sont répartis en deux catégories vis-à-vis d'un service donné, les objets clients et les objets serveurs. Les clients peuvent invoquer les méthodes d'autres objets (dits alors serveurs) sur le même site, on parle dans ce cas d'invocation locale, ou sur des sites distants, il s'agit alors du RMI (voir figure 3.4).

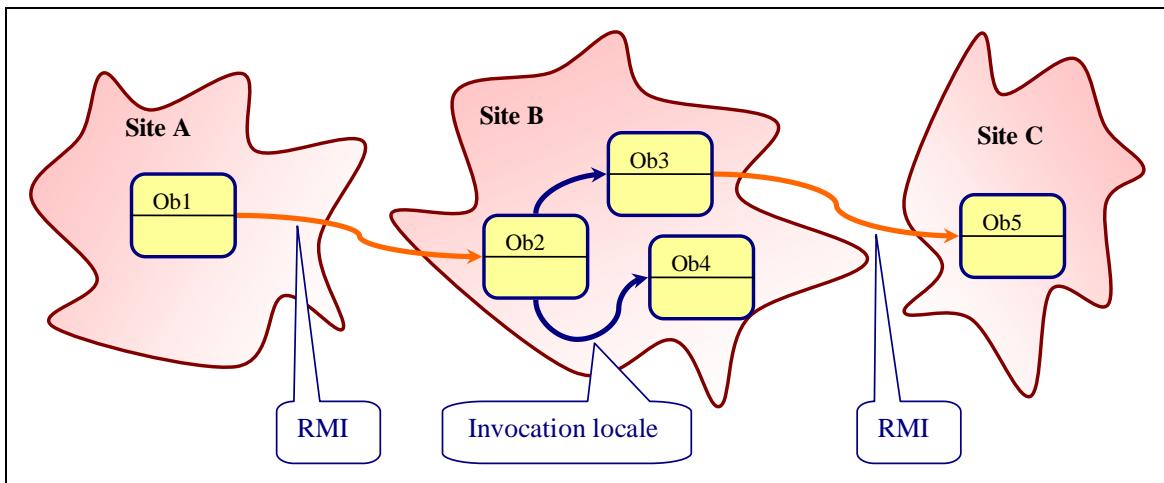


Figure 3.4. Invocation locale et invocation à distance

Dans un site donné, l'ensemble des objets qui peuvent recevoir un RMI est géré par un module serveur. Il intercepte les messages envoyés par les clients et dispatche chacun à l'objet serveur correspondant. Ce dernier exécute la méthode correspondant au message puis renvoie le résultat au module serveur qui le transmet au client. À un moment donné, plusieurs exécutions de la même méthode peuvent être invoquées. Dans certains cas, il est nécessaire de faire une synchronisation.

Dans une approche basée objets, certains aspects sont à noter :

- ★ **Référence des objets distants** : Chaque objet susceptible de recevoir un RMI doit avoir une 'référence distante' qui consiste en un identificateur unique à travers tout le système réparti. La référence distante peut être construite par une concaténation comme suit :

Adresse Internet	N°du port du processus ayant créé l'objet	Moment de création	Numéro local de l'objet
------------------	---	--------------------	-------------------------

Les références distantes peuvent être passées comme paramètres et rendus comme résultat.

- ★ **L'interface distante** : Chaque objet distant (qui peut recevoir un RMI) dispose d'une interface consistant en un nombre de méthodes que sa classe

implémente. Les objets dans d'autres processus ne peuvent invoquer ces méthodes que s'ils disposent des formes exactes de leurs signatures (nom, paramètre et type du résultat retourné). C'est pour cela que la définition des interfaces est nécessaire. Elle consiste, dans le cas du RMI, en une série de signature de méthodes. Les objets locaux peuvent invoquer les méthodes définies dans l'interface en plus de l'invocation d'autres méthodes implémentées localement. Dans le mécanisme RMI de Java, les interfaces distantes sont définies de la même façon que les autres interfaces, cependant elles héritent de l'interface particulière nommée *Remote*. Dans CORBA, le langage spécifique *IDL* permet de spécifier les interfaces distantes.

- ★ **Collecteurs de miettes** : La récupération des espaces alloués aux objets requiert la coopération des différents collecteurs de miettes distants. En effet, le collecteur de miettes local ne peut disposer de l'espace alloué à un objet que lorsqu'il ne subsiste plus aucun objet distant maintenant une référence sur le premier
- ★ **Les exceptions** : L'implémentation du RMI doit être capable de gérer des exceptions liées à la répartition aussi bien que celles liées à l'exécution classique des méthodes.

L'implémentation du RMI fait intervenir plusieurs composants dont chacun joue un rôle spécifique. La figure 3.5 donne un aperçu sur les intervenants. Nous décrivons juste après le rôle de chacun.

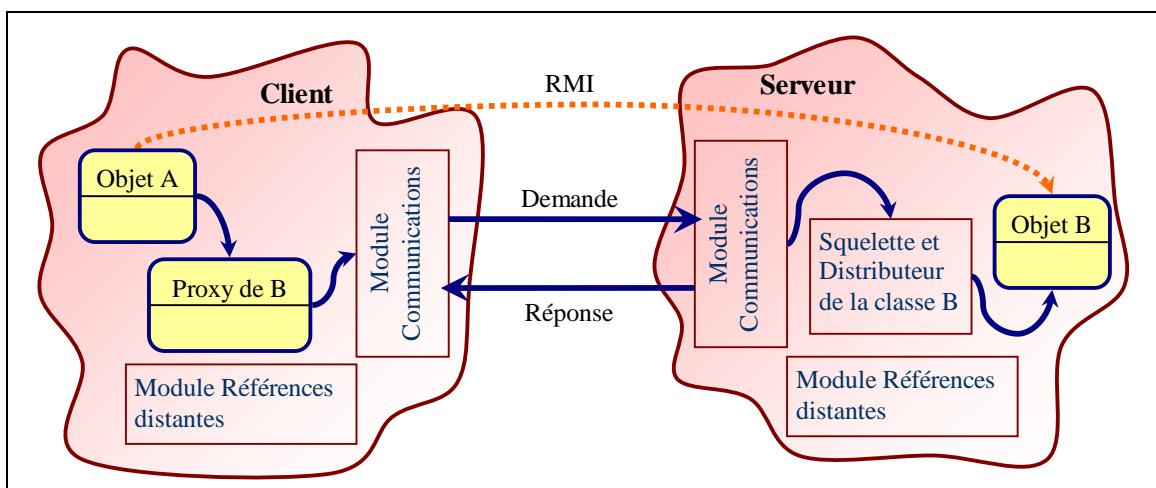


Figure 3.5. Composants intervenant dans l'implémentation du RMI

Rôle du module de communication : Les deux modules de communication se chargent de la transmission de la demande du client vers le serveur et la transmission de la réponse du serveur au client. La demande et la réponse sont des messages composés comme suit :

Type message (0 pour demande et 1 pour réponse)
Identificateur de demande (un entier qui sera retourné avec la réponse)
Référence d'objet (spécifiée précédemment)
Identificateur de méthode (un entier ou un nom si on utilise le même langage avec possibilité de réflexion)
Arguments (tableau d'octet)

Les modules de communication utilisent uniquement les trois premiers champs (ils les consultent lors de la transmission).

Au niveau du serveur, le module Communications sélectionne le distributeur de la classe de l'objet invoqué et lui passe la référence locale de l'objet obtenu du module des références distantes. La couche logicielle qui implémente le RMI est constituée essentiellement du proxy, du squelette et du distributeur (nommés parfois objets middleware). Elle est construite sur les modules Communications et Références Distantes.

Rôle du module Références Distantes : Il est responsable de la translation entre références locales et références distantes et de la création des références distantes pour des objets. Pour cela, le module Références Distantes dans chaque processus maintient une table d'objets distants qui donne la correspondance entre une référence distante (globale) et la référence locale d'un objet dans le processus. La table doit contenir une entrée pour chaque objet distant manipulé par les processus et une entrée pour chaque Proxy local.

Les principales opérations des modules Références Distantes sont :

- ★ Lorsqu'un objet distant est passé comme paramètre ou résultat pour la première fois, le module Références Distantes distant est chargé de créer une entrée qui lui corresponde dans sa table.
- ★ Lorsqu'une référence distante arrive dans une demande ou réponse, le module doit fournir la référence locale correspondante. Celle-ci indique soit un Proxy soit un objet distant. Dans le cas où celle-ci n'est pas dans la table, un Proxy sera créé et le module ajoutera sa référence dans sa table.

Rôle du Proxy : Le Proxy offre une transparence à l'objet appelant en se comportant comme l'objet appelé. Cependant, au lieu d'exécuter une méthode, il constitue un message et l'envoie à l'objet distant puis récupère le résultat et le transmet à l'objet appelant. Le Proxy cache tout le détail de la gestion de l'invocation de l'objet distant.

Il existe un objet Proxy pour chaque objet distant. La classe du Proxy implémente la même interface que la classe de l'objet distant. Mais cette implémentation diffère de celle de la classe de l'objet distant. Chaque méthode dans le Proxy forme un message de demande, l'envoi à l'objet distant puis récupère le résultat et le transmet à l'objet appelant.

Rôle du distributeur : Le serveur a un distributeur est un squelette pour chaque classe d'un objet distant. Le distributeur reçoit la demande du module Communication, utilise l'Identificateur de méthode pour sélectionner, dans la table, la méthode appropriée du squelette et lui transmet la demande. Le distributeur et le Proxy utilisent le même identificateur de méthode pour une méthode de l'interface distante.

Rôle du squelette : Il implémente les méthodes de l'interface distante, en extrayant les arguments et en invoquant la méthode de l'objet distant. Ensuite il attend le résultat puis forme un message résultat (avec éventuellement des exceptions) et le renvoi au Proxy (i.e. sa méthode invocatrice).

Remarques :

- ★ Au départ, les programmes clients ont besoin d'avoir une référence distante pour au moins un objet distant. L'éditeur de liens pour un système réparti est un service séparé qui maintient une table contenant la correspondance entre les noms textuels et les références distantes. Cette table sera utilisée par les serveurs pour enregistrer les objets distants et par les clients pour la recherche.
- ★ Pour éviter qu'une exécution d'une méthode invoquée ne retarde une autre, les serveurs lancent un thread séparé pour chaque invocation distante.
- ★ Dans la suite nous donnons un scénario du RMI du côté client

<pre> Objet a ... b.mb (i,j,k) ; // Va au proxy /*b est le proxy de l'objet distant qui exécutera la demande */ </pre> <pre> Proxy de l'objet b //Il maintient une référence distante de b mb1(){} mb2(){} mb(...i, ...j, ...k) {...} </pre>					
Etape 1 : Former un paquet requête + plus création des références distantes de i, j et k si elles n'existent pas déjà <table border="1" style="margin-top: 5px; border-collapse: collapse; width: 100%;"> <tr><td>0 (Signifiant Requête)</td></tr> <tr><td>15 (Numéro de la requête)</td></tr> <tr><td>b (Référence distante de l'objet b)</td></tr> <tr><td>3 (numéro du service, désigne la méthode mb())</td></tr> <tr><td>01AF8531C ... (octets représentant les paramètres)</td></tr> </table>	0 (Signifiant Requête)	15 (Numéro de la requête)	b (Référence distante de l'objet b)	3 (numéro du service, désigne la méthode mb())	01AF8531C ... (octets représentant les paramètres)
0 (Signifiant Requête)					
15 (Numéro de la requête)					
b (Référence distante de l'objet b)					
3 (numéro du service, désigne la méthode mb())					
01AF8531C ... (octets représentant les paramètres)					
Etape 2 : Envoyer le paquet en invoquant un service du module Communications					
Etape 3 : Attendre puis recevoir le paquet réponse <table border="1" style="margin-top: 5px; border-collapse: collapse; width: 100%;"> <tr><td>1 (Signifiant Réponse)</td></tr> <tr><td>15 (Numéro de la requête)</td></tr> <tr><td>... (non utilisé)</td></tr> <tr><td>... (nom utilisé)</td></tr> <tr><td>AB0018F ... (octets représentant le résultat)</td></tr> </table>	1 (Signifiant Réponse)	15 (Numéro de la requête)	... (non utilisé)	... (nom utilisé)	AB0018F ... (octets représentant le résultat)
1 (Signifiant Réponse)					
15 (Numéro de la requête)					
... (non utilisé)					
... (nom utilisé)					
AB0018F ... (octets représentant le résultat)					

Etape 4 : Extraire le résultat et le retourner à l'objet a (si la valeur retournée est une référence d'objet, on cherche sa référence locale et on la transmet. S'il n'y a pas de référence locale on crée un proxy local pour l'objet référencé et on transmet sa référence)

...

}

3.4 COMMUNICATION PAR EVENEMENTS ET NOTIFICATIONS

Dans un système, un objet peut réagir à un changement qui touche un autre objet. Ce changement est considéré comme un événement. Par exemple, un clic sur le bouton d'une souris ou la saisie d'un texte dans une fenêtre, constituent des événements qui sont notifiés aux objets responsables de l'affichage.

Dans un système réparti, la notification ne se fait pas seulement pour les objets locaux mais pour les objets résidants sur d'autres sites ; ce qui est une extension naturelle.

Les systèmes répartis basés sur les événements utilisent un mécanisme dit de *publication/abonnement*. Un objet qui génère des événements publie le type d'événement qu'il rend observable aux autres objets. Les objets qui souhaitent recevoir des événements, souscrivent un abonnement aux types qui les intéressent. Les événements peuvent correspondre à l'exécution de certaines méthodes par un objet.

Les objets qui représentent les événements sont appelés notifications. Les notifications peuvent être sauvegardées, envoyées dans des messages ou utilisé de diverses façons.

Les systèmes répartis basés sur les événements ont deux caractéristiques :

- ★ L'hétérogénéité : les différents composants sont hétérogènes mais pour garantir la communication, il faut utiliser le mécanisme de publication/abonnement.
- ★ L'asynchronisme : les objets générateurs d'événements sont totalement découplés des objets abonnés. Aucune synchronisation n'a lieu. Cependant, les abonnés peuvent rester en attente d'un objet notification.

Dans un système réparti basé sur les événements, on trouve un ensemble d'objets et des participants :

- ★ Objets centre d'intérêt : Ce sont les objets qui changent d'états suite à des opérations quelconques.
- ★ Événements : Correspondent souvent à la fin de l'exécution d'une opération sur un objet centre d'intérêt.
- ★ Notification : Objet contenant une information sur l'événement. Il contient généralement le type de l'événement, les attributs qui indiquent l'état de

l'objet centre d'intérêt, la méthode invoquée, le temps d'occurrence et un numéro de séquence.

- ★ Abonnés : Des objets intéressés par un type d'événement et qui reçoivent des notifications.
- ★ Objets observateurs : Ils permettent de découpler les abonnées et les objets centre d'intérêt. Chaque objet centre d'intérêt peut avoir plusieurs abonnés avec des intérêts différents (valeurs différentes pour les attributs) et les objets observateurs permettent de maintenir des objets centre d'intérêt simples.
- ★ Objets éditeurs : Un objet éditeur est un objet centre d'intérêt ou un observateur qui déclare son intention de générer des notification d'un type donné.

La relation entre les différents objets peut se faire de diverses façons (voir figure 3.6)

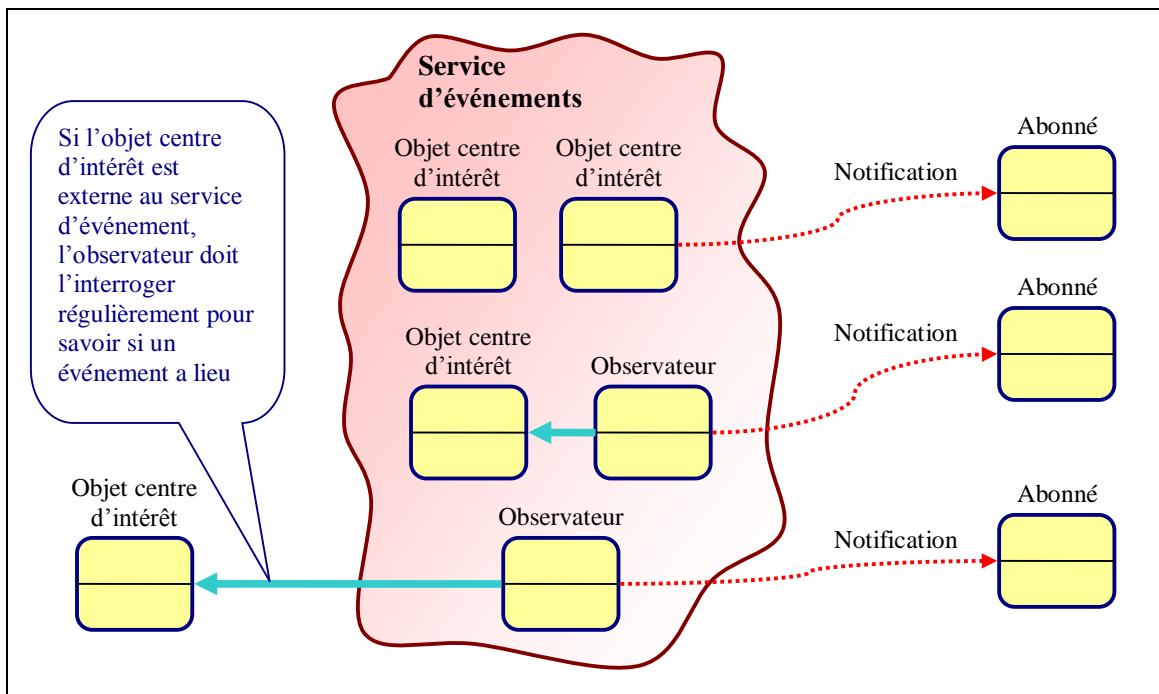


Figure 3.6. Relation entre objets

Les observateurs ont pour rôle de :

- ★ Transmettre des notifications
- ★ Filtrer des notifications
- ★ Opérer des compositions d'événements
- ★ Servir comme boîte aux lettres : Un observateur peut maintenir l'information en attendant que l'abonné la réclame (synchronisation).

3.5 COMMUNICATION DE GROUPE

Lorsqu'il s'agit de la communication d'un processus avec un groupe d'autres processus, les modes de communication présentés précédemment ne sont pas convenables. La communication « multicast » (diffusion) est préférable ; elle consiste à envoyer un message vers un groupe et non plus un seul processus.

L'opération multicast peut être utilisée à différentes fins :

- ★ Tolérance aux fautes : Il s'agit de dupliquer le processus serveur (i.e. former un groupe de serveurs) de tel sorte à ce qu'un processus client adresse sa requête au groupe plutôt qu'à un processus spécifique. Chaque serveur du groupe exécute la même requête et ainsi le client aura plus de chance d'être servi même si certains processus du groupe échouent.
- ★ Découverte des services dans un réseau ad hoc : Il s'agit de rechercher dynamiquement des interfaces pour construire par la suite des requêtes invoquant des services particuliers de celles-ci. Un client peut diffuser un message non pas pour demander l'exécution d'un service donné mais pour récupérer une interface. Les serveurs, à leur tour, utilisent des opérations multicast pour enregistrer leurs interfaces dans divers sites.
- ★ Mises à jour des données dupliquées : Lorsqu'on souhaite améliorer les performances par la duplication des données, l'opération multicast est utilisée pour diffuser les nouvelles valeurs des données à chaque modification.
- ★ Notification des événements : La notification des événements fait naturellement usage de l'opération multicast pour notifier les abonnés.

L'avantage d'une opération multicast réside dans le fait que le processus qui l'utilise génère une seule opération pour envoyer un message à tout le groupe ce qui est convenable du point de vue programmation mais aussi du point de vue efficacité et fiabilité.

Sur le plan efficacité, l'implémentation du multicast peut se limiter à envoyer un message une seule fois sur chaque lien. Cela est possible en donnant au groupe la forme d'un arbre et en utilisant le support hardware pour le multicast à chaque fois que cela est possible. Par exemple, si un message doit être transmis deux fois à deux destinataires qui se trouvent dans le même réseau Ethernet, un multicast-IP permet d'acheminer un seul message jusqu'au dernier routeur qui utilise le multicast d'Ethernet pour transmettre une copie du message à chaque destinataire.

Sur le plan fiabilité, le multicast qui est implémenté par une suite d'opération Send (nommé Multicast-basic) n'est pas fiable car si on suppose que chaque opération Send est prise en charge par un thread, il y aura une explosion d'acquittements dès que le nombre de processus d'un groupe devient important. En effet, les buffers du processus

émetteur débordent entraînant des pertes d'acquittements qui entraînent à leur tour des retransmissions ce qui réduit la bande passante du réseau.

Le multicast fiable se caractérise par trois propriétés :

- ★ **Intégrité** : Chaque message reçu est identique au message transmis et aucun message n'est délivré deux fois.
- ★ **Validité** : N'importe qu'elle message parvient éventuellement à sa destination s'il est correct (ne comporte pas d'information fausse).
- ★ **Accord (Agreement)** : Tous les processus corrects du groupe doivent recevoir un message si l'un d'eux le reçoit.

Il existe divers algorithmes qui permettent une implémentation fiable du multicast. Parmi ceux là, l'algorithme qui se base sur le Multicast-IP utilise des acquittements négatifs (détection de l'absence d'un message attendu) et les acquittements indirects (Piggy backed acknowledgement: acquittement attaché à d'autres messages). Cet algorithme part de la constatation que le protocole Multicast-IP réussit souvent sans les d'acquittements.

Dans la communication de groupe, la création et la gestion des groupes constituent un problème fondamental. En effet, si dans le cas le plus simple, les groupes sont statiques, dans le cas général pratique, les groupes évoluent dynamiquement par ajout ou retrait des processus et chaque processus peut appartenir à plusieurs groupes.

Les systèmes qui utilisent la notion de groupe doivent contenir un service de gestion de l'appartenance aux groupes dont le rôle est :

- ★ Offrir une interface de gestion des membres : création/destruction de groupes et création/destruction de membres.
- ★ Implémenter une détection de pannes : Le service détecte la panne des membres mais aussi celle des communications. Il exclut un membre s'il devient non atteignable.
- ★ Notification des membres lors d'un changement : Les membres d'un groupe sont maintenus informés de l'introduction/retrait d'un nouveau membre.
- ★ Expansion des adresses de groupe : Les processus qui effectuent une opération multicast fournissent un identificateur de groupe. Le service peut coordonner l'acheminement d'un message en présence des changements qui affectent le groupe (ajout/retrait de membres) grâce au contrôle de l'expansion des adresses.

3.6 COMMUNICATION PAR MEMOIRE PARTAGEE DISTRIBUEE

Lorsque les processus d'une application distribuée sont physiquement répartis sur des machines distinctes et ne disposant d'aucune mémoire physique partagée, il est possible de créer une mémoire commune virtuelle qui sera utilisée pour échanger des données diverses.

La mémoire partagé distribuée constitue un paradigme de communication qui tente de rapprocher les systèmes distribués d'une configuration centralisée du type des machines multiprocesseurs.

Le paradigme de mémoire partagée distribuée n'est généralement pas convenable pour les applications client/serveur où généralement le serveur encapsule les ressources dont les données de l'application (pour des raisons de modularité et de protection). Ces dernières peuvent être limitées comme elles peuvent être des bases de données entière, ce qui nécessite l'utilisation d'un serveur de données.

A l'opposé, les processus pairs peuvent profiter pleinement des possibilités de communication par mémoire partagée, ce qui réduit la tâche du programmeur et lui évite l'utilisation des messages.

Comparée à la communication par message, la communication par mémoire partagée présente plusieurs particularités. Nous citons :

- ★ Le codage des données : Lors de l'échange de messages, le processus émetteur doit coder les données à transmettre sous forme d'une suite d'octets ce qui nécessite la reconstitution de ces données selon leur type par le processus récepteur. A l'opposé, les processus accèdent aux données de la mémoire partagée de la même façon qu'ils accèdent à leurs mémoires locales.
- ★ Protection des espaces d'adressage : Les processus qui échangent des messages préservent l'indépendance de leurs espaces d'adresses. A l'opposé, un processus peut altérer la mémoire partagée par erreur et induire des problèmes pour les autres processus.
- ★ La synchronisation entre processus : Les processus qui échangent des messages se synchronisent à l'aide des primitives d'envoi et de réception. Lorsqu'on dispose d'une mémoire partagée, il est possible d'utiliser les mécanismes de synchronisation classiques entre processus, moyennant une certaine adaptation. Par exemple, il est envisageable d'utiliser des sémaphores.
- ★ Exécution libre des processus : La programmation des processus en se basant sur l'échange de messages suppose implicitement l'exécution simultanée des processus. Un certain chevauchement est nécessaire. A

l'opposé, l'utilisation d'une mémoire partagée permet à un processus de déposer des données qu'un autre processus pourra lire immédiatement ou après la terminaison du premier. L'exécution des processus est ainsi découpée.

- ★ Communication indirecte : Dans la communication par messages, le processus émetteur doit spécifier le ou les processus récepteurs (problème du canal de communication). A l'opposé, la communication par mémoire commune élimine cette exigence en permettant aux processus de communiquer sans se connaître.

La mise en œuvre du paradigme de communication par mémoire partagée se base sur la communication par messages. Cette mise en œuvre n'est pas aisée. Elle doit faire face au problème de manque de performances lorsque le nombre d'ordinateurs concernés devient important (donc nombre de messages important) et doit aussi maintenir la consistance des données stockées qui peuvent exister en plusieurs exemplaires.

Il existe diverses approches de mise en œuvre des mémoires partagées qu'on peut partager en deux groupes : Les approches matérielles et les approches logicielles. Les approches matérielles concernent les machines multiprocesseurs considérées comme un cas particulier de systèmes distribués.

Les approches logicielles quant à elles se font soit au niveau des systèmes d'exploitation, soit au niveau des middleware (voire des applications).

Dans la suite, nous considérons les aspects importants qui caractérisent toute implémentation de la communication par mémoire partagée. Notons que aspects diffèrent d'une l'implémentation à l'autre.

3.6.1 Structure

La structure de la mémoire partagée peut consister en un ensemble d'octet (ou mots) auquel les processus accèdent par un mécanisme d'adressage. Ils utilisent pour cela les opérations de lecture et d'écriture classiques. La structure peut aussi être une collection d'objets qui encapsulent des variables. L'accès aux variables ne peut se faire qu'en invoquant les opérations associées aux objets qui détiennent ces variables. Comme dernière possibilité, la mémoire partagée peut avoir une structure composée d'éléments de même type (par exemple des tuples identiques avec des valeurs différentes). Dans ce cas, la mémoire est plutôt vue comme une boîte aux lettres. Certains processus y déposent des informations sous un format bien précis et les autres récupèrent ces informations et les utilisent.

3.6.2 Synchronisation

L'utilisation correcte d'une mémoire partagée nécessite l'utilisation d'un service de synchronisation distribué (verrous, sémaphores, etc.). Même lorsque la mémoire est sous forme d'objets, il est nécessaire d'empêcher l'exécution simultanée des opérations d'un objet.

3.6.3 Consistance

Pour améliorer les performances, les différentes données de la mémoire partagée peuvent être lues et enregistrées dans des mémoires locales qui jouent le rôle de mémoires caches. Dans certains cas, la mémoire partagée est physiquement matérialisée par l'ensemble des caches. Les processus accèdent alors aux caches pour les opérations de lecture. Lorsqu'un processus met à jour une données dans un cache, toutes les autres copies disponibles sur les autres caches doivent être mise à jour aussi. On parle ici de propagation des mises à jour. Il se pose alors le problème de consistance concernant les valeurs des données que le service de maintien de la consistance doit garantir. En effet, lorsqu'un processus lis le contenu d'une variable dans sans cache, cette valeur peut ne pas correspondre son état réel actuel. Le service de maintien de la consistance doit implémenter un protocole qui garantit la consistance en fonction du type d'application. Dans certains cas, on ne tolère aucune inconsistance, le processus qui effectue une lecture doit réellement lire la dernière valeur.

3.6.4 Options de mise à jour

3.6.5 Granularité

3.6.6 Trashing

3.7 COMMUNICATION PAR FLUX (STREAMING)

La communication vue jusqu'à présent s'intéressait à l'échange d'unités d'information indépendantes et complètes. Le temps n'a pas d'importance et il importe peu de savoir à quel moment la communication aura lieu.

Il existe des formes de communication dans lesquelles le temps joue un rôle crucial. Par exemple, considérons un flux audio consistant en une séquence d'échantillons de 16 bits qui représentent l'amplitude d'une onde sonore (cas de Pulse Code Modulation (PCM)) échantilloné à une fréquence de 44100Hz. Dans une telle situation, la reproduction du signal original nécessite que :

- 1- Les échantillons doivent être lus dans l'ordre qui correspond à leur apparition dans le flux et
- 2- Qu'ils doivent être lus à des intervalles de $1/44100$ seconde exactement. Si le flux est lu à une cadence différente, nous obtenons un son différent du son original.

Lorsqu'on parle de prise en compte de l'information dépendante du temps, il s'agit de codage de l'information, de transmission de l'information et de reconstitution de l'information d'origine en tenant compte du temps. L'information dépendante du temps concerne les flux audio et les flux vidéo.

On distingue ainsi deux façons de coder ou représenter l'information. Une représentation continue et une représentation discrète. Dans la **représentation continue**, les éléments d'information ont une relation temporelle qui est importante pour interpréter correctement le sens de l'information. Par exemple, dans une animation, une série d'images doivent être présentées à des intervalles fixes de l'ordre de 30 à 40 ms par image et dans un ordre bien défini.

Dans la **représentation discrète**, la relation temporelle entre les éléments d'information n'a aucune influence sur l'interprétation correcte des données.

Pour échanger des informations dépendantes du temps, les systèmes distribués offrent généralement le concept de **flux de données** (Data streams) qui ne sont autres que des séquences d'unités de données. Les flux de données peuvent être appliqués aussi bien au cas de représentation continue qu'au cas de représentation discrète. Par exemple, la lecture d'un fichier son nécessite l'établissement d'un flux de données continu entre le fichier et le périphérique audio qui produit le son.

Le temps est crucial pour les flux de données continus. Pour capturer l'aspect temporel, une distinction est faite entre les modes de transmission. Dans le **mode de transmission asynchrone**, les éléments de données sont transmis l'un après l'autre sans aucune contrainte temporelle sur le moment où le transfert aura lieu. C'est le cas des flux de données discrets (par exemple un transfert de fichiers ordinaires).

Dans le **mode de transmission synchrone**, il y a un délai maximal pour la transmission de chaque élément d'information dans le flux de données. Un élément peut être transmis plus rapidement que ce qui est exigé, mais cela n'a pas d'importance. Par exemple, un capteur peut prendre des échantillons de température avec un certain taux et les transmettre à un opérateur via un réseau. Dans ce cas, il est important de garantir que la propagation de bout en bout à travers le réseau soit inférieure au temps de prise de deux échantillons successifs.

Dans le **mode de transmission isochrone**, il est important que les unités de données soient transmises à temps. Cela implique qu'il y a un délai maximal et un délai minimal

pour le transfert de bout en bout. Le mode isochrone est intéressant pour les systèmes multimedia distribués où il joue un rôle important dans la transmission audio et vidéo.

Dans la suite nous considérons uniquement le mode de transmission isochrone.

Les flux peuvent être simple ou complexes. Un **flux simple** consiste en une seule séquence de données alors qu'un **flux complexe** consiste en plusieurs flux simples en relation. Cette relation est souvent temporelle. Par exemple, un son audio stéréo peut être transmis par un flux complexe composé de deux flux simples où chacun correspond à un canal audio. Il est important que ces deux flux simple restent synchronisés continuellement pour garantir un effet stéréo. Un autre exemple, est celui de la transmission d'un film. Dans ce cas, il est nécessaire d'avoir un flux video et deux flux audio représentant le son stéréo du film. Un quatrième flux peut contenir un sous-titrage pour sourds/muets ou une traduction dans une langue différente de celle du signal audio. La synchronisation de tous les flux du flux complexe est importante. En effet, en cas d'échec, la reconstitution du film échoue.

Un flux peut être considéré comme une connexion virtuelle entre une **source** et un **puit** (Source and Sink). Les deux pouvant être des processus ou des périphériques. Par exemple, un processus lit un fichier audio à partir d'un disque et le transmet octet par octet sur un réseau. Le puit peut être un processus qui reçoit les octets et les transmet à un périphérique audio local. Dans les systèmes multimedia, il peut être fourni un support pour une connexion directe entre la source et le puit. La figure 3.7 illustre les deux situations.

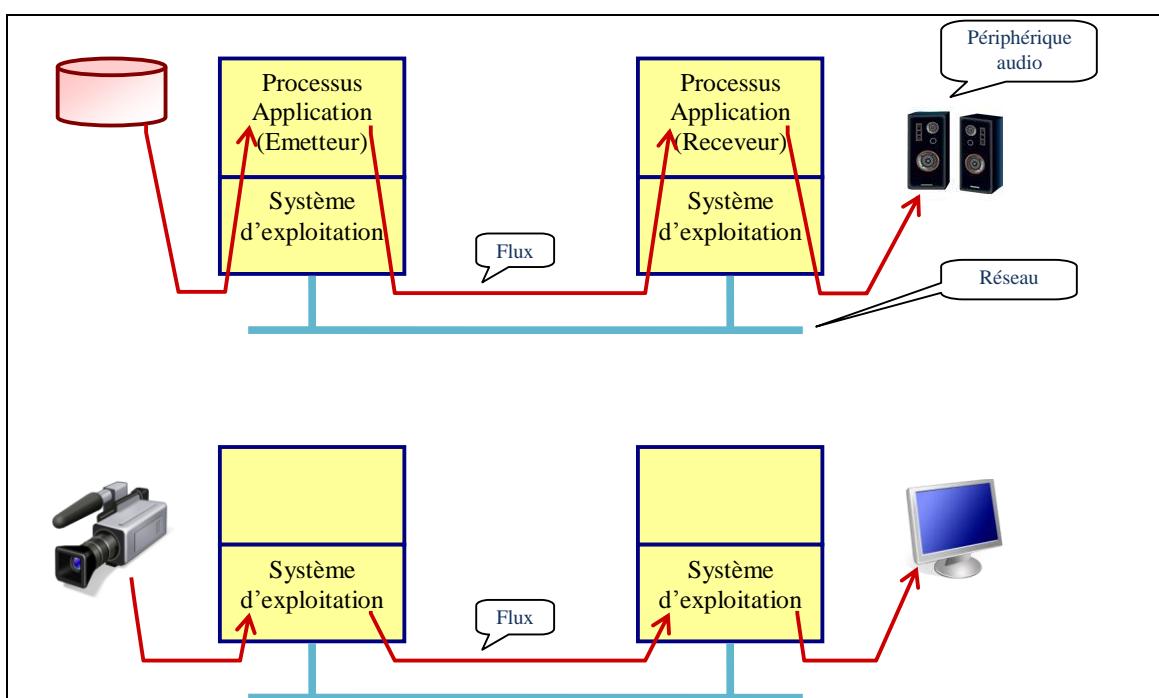


Figure 3.7. Connexion virtuelle entre Source et Puit

Un autre aspect important est celui de l'existence de plusieurs sources et plusieurs puits. Il s'agit de communication dite **multiparties**. Dans la pratique, la situation de communication multipartie la plus commune est celle de rattacher plusieurs puits à un flux ce qui permet de diffuser l'information à plusieurs receveurs.

Le problème majeur de la diffusion de flux vient de la situation où les receveurs ont différents exigences sur la qualité du flux. Par exemple, lorsqu'il s'agit d'un flux complexe représentant un film à 50 images par seconde et deux sous flux audio. Même en utilisant des techniques sophistiquées de compression la bande passante nécessaire avoisine 30 million de bits par seconde. Les receveurs ne peuvent tous manipuler autant de données ce qui nous pousse à utiliser des filtres qui s'accomodent de bande passantes réduites (figure 3.8).

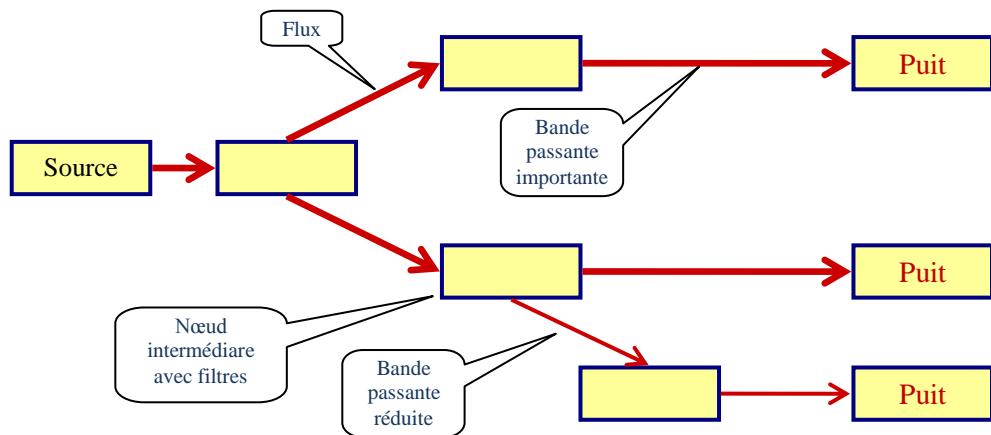


Figure 3.8. Diffusion de flux à plusieurs receveurs

3.7.1 Notion de qualités de service

Certains besoins non fonctionnels tel que la dépendance vis-à-vis du temps d'un flux ou la fiabilité de communication représente ce qui est communément nommé **qualité de service (QoS ou Quality of Service en anglais)**. Exprimer la qualité des services peut être fait de plusieurs façons. L'une serait de donner une spécification du flux précise qui indique :

1- ce qu'une application attend du réseau de communication. Par exemple :

- **Sensibilité à la perte** (en octet) et **intervalle de perte** (en microseconde) : Elles expriment conjointement un maximum acceptable de taux de perte (par exemple : 1 octet par minute).
- **Sensibilité à la rupture** (en unité de données) : Elle indique combien des données consécutives peuvent être perdues au maximum.
- **Le délai minimal détectable** (en microseconde) : Il spécifie de combien le réseau peut retarder la transmission des données sans que le receveur ne s'en rende compte.

- **La variation maximal du delay** (en microseconde) : C'est une mesure importante pour la video et l'audio. Elle spécifie un intervalle de tolérance lors de la transmission de deux unités de données consécutives d'un flux.
- **Qualité de la garantie** : C'est nombre qui indique l'importance des spécifications de la qualité de service. Un nombre petit indique une grande tolérance de l'application, alors qu'un nombre élevé signifie que si les qualités de service ne sont pas garanties il sera inutile d'établir un flux.

2- Ce que l'application délivre en entrée pour le réseau, par exemple :

- Taille maximale des unités de données (en octet)
- Taux de transmission maximal (en octet/seconde)
- Etc

Dans certains cas ces spécifications ne sont pas faciles à déterminer. Une autre alternative serait de laisser l'utilisateur spécifier s'il s'agit d'audio ou de video et dans l'un ou l'autre cas, indiquer la qualité souhaitée du flux par des termes qui reflète une classification préalable (par exemple : haut, moyen, bas) des différents paramètres.

3.7.2 Configuration d'un flux

Une fois les spécifications décrites, le système distribué est en mesure d'allouer les ressources nécessaires pour que la communication ait lieu en garantissant les qualités de service requises. Les ressources nécessaires concernent :

- la bande passante : en ordonnançant la transmission des unités de données avec une certaine priorité.
- les buffers : où les unités peuvent être mis en file en attendant leur transmission (buffers au niveau des routeurs et des systèmes d'exploitation).
- les capacités de calcul : qui nécessite l'allocation, selon un ordonnancement convenable, du processeur aux ordonnanceurs, filtres, codeurs et décodeurs.

Un des problèmes clés concernant la configuration du flux et celui de la traduction des spécifications de qualité de service en paramètres concernant les ressources impliquées. Par exemple pour une sensibilité à la rupture k , il serait nécessaire d'allouer statiquement des buffers (avec une certaine taille) tout au long du chemin entre la source et le puit.

La mise en pratique de la notion de qualité de service n'est pas une tâche facile. Cela est due à trois raisons : absence d'un modèle de référence unique pour la spécification de la QoS, absence de possibilité de description abstraite des ressources d'un réseau de communication et absence de consensus sur une approche unique de traduction des

QoS en demande pour les ressources. Notons enfin, que cette traduction nécessite l'utilisation d'un protocole spécifique qui réside dans la couche de transport tel que le RSVP (Resource reSerVation Protocol).

3.7.3 Synchronisation des flux

Lorsqu'il s'agit de flux complexes, il est souvent nécessaire d'établir une synchronisation entre les flux constituants de telle sorte à préserver une relation quelconque entre ces derniers. La synchronisation peut avoir lieu pour les flux de données discrets ou continus.

Par exemple, pour les flux de données discrets, on peut considérer l'association entre un affichage écran d'une présentation (par exemple une présentation Powerpoint) et un signal audio décrivant l'écran en question. Bien que la présentation et le signal audio peuvent être transmis sans contrainte temporelles sur un réseau, il est primordial que les l'affichage écran soit synchronisé avec la partie du signal audio qui lui correspond.

Pour les flux de données continus, tel que les films, le problème de synchronisation est plus complexe. Il est souvent nommé Lip Synchronisation (Synchronisation son-mouvement des lèvres).

La synchronisation de deux flux représentant un signal audio stereo nécessite une synchronisation au niveau de chaque unité de données (i.e. échantillon de 16 bits) est nécessaire. Autrement dit, pour une fréquence d'échantillonage de 44100 Hz, on doit lire un échantillon de chaque flux tous les 22,7 microsecondes ($1/44100$).

Pour la synchronisation audio/video, le standard NTSC utilise une fréquence vidéo de 30 Hz (soit 30 unité image par seconde), ce qui nous oblige à utiliser comme unité de donnée pour le son le nombre d'échantillons de 16 bits qu'il est possible de lire en 1/30 seconde (33 millisecondes) soit 1470 échantillons ou encore 11760 octet ($(44100/30)*16/2$).

Qu'en est-il du mécanisme de synchronisation lui-même ? La synchronisation s'opère sur les unités de données des flux simples. C'est un processus simple qui opère des lectures et des écritures sur plusieurs flux en s'assurant que les contraintes temporelles sont satisfaites. La synchronisation peut être à la charge de l'émetteur comme elle peut être à la charge du receveur. Lorsqu'elle est à la charge de l'émetteur, il constitue des unités de données en fonction de la synchronisation, le récepteur n'aura qu'à décomposé chaque unité pour en extraire l'unité qui correspond à chaque flux et la transmettre au périphérique adéquat. Lorsque la synchronisation est à la charge du récepteur, l'émetteur se contente de transmettre les flux simples.

MISE EN EVIDENCE DES PROBLEMES FONDAMENTAUX DES SYSTEMES DISTRIBUES

CONTENU

- 4.1 Introduction
- 4.2 Protocoles du Web
- 4.3 Les langages clients (HTML, Javascript)
- 4.4 Les langages serveurs (PHP, JSP)
- 4.5 Le langage XML (Morphologie et spécification des documents XML, DTD, Schémas, styles)
- 4.6 L'approche RIA (AJAX, Google Gears, JavaFx, CURL)

4.1 INTRODUCTION

2.4.1.4 Architecture des applications du World Wide WEB

L'idée initiale de lien associatifs entre des données et des documents d'origines diverses est attribué à douglas Engelbart, directeur de l'ARC (Argumentation Research Center of Stanford Research Institute), dans les années 50.

Le *lien associatif* est constitué d'une référence (la *cible*) et d'une ancre (la *source*). La figure 4.1 montre un exemple d'hypertexte. La référence et l'ancre résident normalement dans des nœuds informationnels différents (textes, image, voire sons). Il est ainsi possible de tisser une véritable toile représentant les liens sémantiques entre différentes informations. L'utilisateur peut alors se déplacer dans ce réseau, passant d'un nœud à l'autre selon ses objectifs.

Des outils tels que les systèmes de documentations et d'aides en ligne des logiciels utilisent cette approche de navigation où l'on doit cliquer pour passer d'une explication à une autre, suivant ses besoins.

Le contenu des nœuds est désormais étendu à d'autres types de données tels que : image, graphisme, vidéo, son, etc, faisant ainsi, évoluer le principe de l'hypertexte vers celui de l'*hypermédia*.

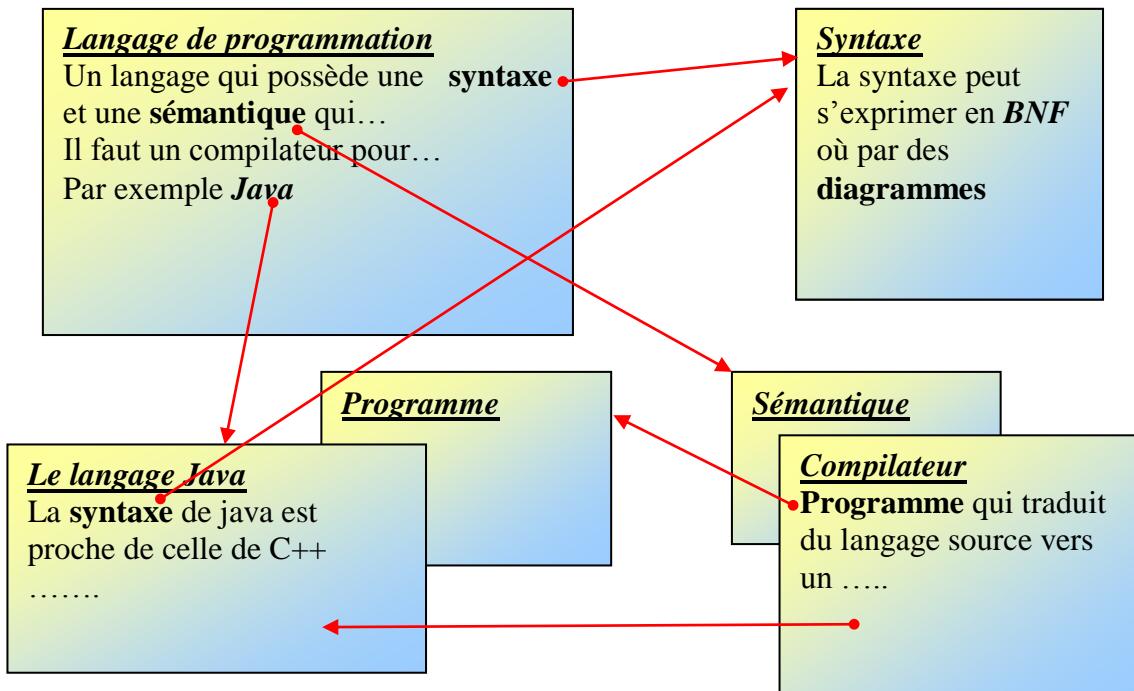


Figure 4.1. Exemple d'un hypertexte

Si on imagine un instant que nous avons à disposition un hypertexte et que les nœuds de celui-ci soient répartis sur les différents plateformes de l'internet, nous obtenons le *World Wide Web* (WWW). Le WWW n'est autre qu'une toile sémantique, composée de page web et de ressources de divers formats, supportées par une toile physique qu'est l'internet.

Le passage d'un nœud à l'autre (d'une page à l'autre) ou navigation est supporté par des outils et des protocoles standards. L'utilisateur dispose d'un navigateur qui lui permet, en cliquant sur un lien (dit URL : Uniform Resource Locator), de demander la ressource ou la page correspondant à ce lien au serveur qui la détient. Le serveur, dit serveur Web, communique avec les navigateurs en utilisant le protocole nommé HTTP.

Les navigateurs sont compatibles entre eux et ne diffèrent que par les fonctionnalités supplémentaires qu'ils peuvent offrir. Les navigateurs ont pour principale tâche l'affichage des pages Web, qu'ils reçoivent, aux utilisateurs. Pour cela, les pages utilisent un codage spécial qui obéit à la norme HTML.

Les serveurs Web sont des composants qui ont la capacité d'interpréter les requêtes invoquées par un client et lui renvoyer une réponse. Le serveur Web prend en charge le protocole HTTP. Il réceptionne les requêtes des clients et leur retourne le contenu des nœuds référencés. Il sert également de passerelle (gateway) vers des applications invoquées par les clients. De plus il gère les droits d'accès à certains nœuds et récolte des statistiques sur l'utilisation des services. Il existe divers serveurs HTTP (par exemple NCSA httpd développé par National Center for Supercomputing Applications) développés pour les principales plateformes : Unix, Windows, etc..

La figure 4.2 illustre un scénario de l'échange entre le navigateur et le serveur lorsqu'il s'agit d'un transfert de page Web.

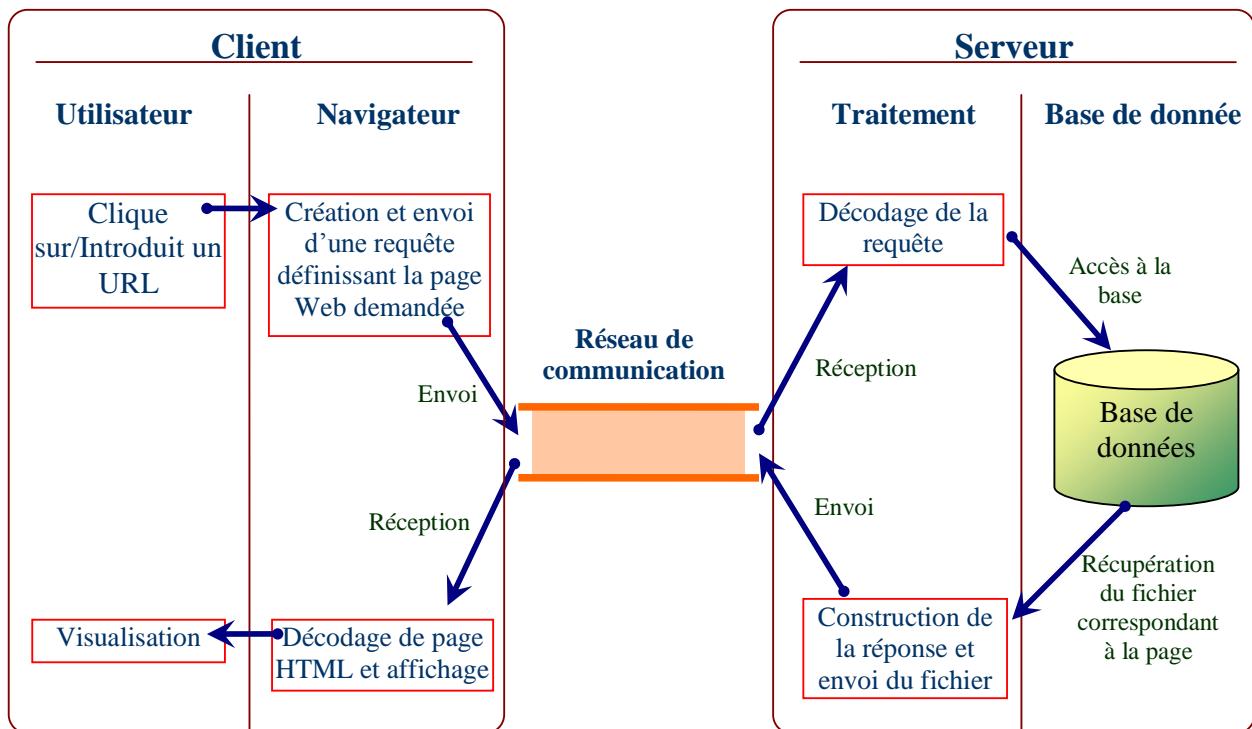


Figure 4.2. Fonctionnement de l'échange entre client et serveur sur le WWW

L'utilisateur clique sur un lien dans une page Web ou tape un URL dans la barre d'adresse d'un Navigateur. Le navigateur commence, alors, par l'établissement d'une connexion avec le serveur. Le navigateur utilise l'adresse de nom de domaine pour contacter le serveur. Le navigateur regarde l'adresse du nom de domaine, il envoie ensuite au domaine identifié les en-têtes de requête suivants :

- ★ Un en-tête de requête identifiant le fichier ou le service requis (URL)
- ★ Des champs d'en-tête de requête, identifiant le navigateur
- ★ Des informations spécialisées concernant la requête elle-même
- ★ Les données de la requête

On appelle ces données en-têtes de requête HTTP. Elles indiquent au serveur quelles sont les informations que le client demande, et quel type de réponse est accepté par le client.

Une fois la requête réceptionnée par le serveur, il l'analyse et extrait le premier en-tête entrant, appelé *en-tête de requête Method*, et essaie de localiser la ressource qui correspond à l'URL à son niveau. Pour se faire, il commence par regarder au plus haut niveau du répertoire de la racine serveur pour voir s'il contient un fichier qui correspond à cette URL. Le serveur regarde le chemin d'accès qui suit le nom de domaine, et regarde si le répertoire correspondant contient un nom de fichier convenable.

Une fois la vérification terminée, le serveur répond par des en-têtes de réponse valides. Le premier en-tête de réponse est une ligne d'état qui indique au navigateur client le résultat de la recherche de l'URL demandée. Cette réponse peut être Succès ou échec. Si l'état est succès le contenu de l'URL demandé est généralement envoyé au navigateur client et s'affiche sur l'écran de l'ordinateur du client. Sinon un message d'erreur est renvoyé.

L'une des caractéristiques les plus remarquables, mises à notre disposition par les serveurs Web, est la possibilité de développer des applications ou des scripts que l'utilisateur distant démarrent, en cliquant sur des URLs ou en remplissant et soumettant un formulaire HTML. Avec des langages de programmation on peut créer des applications ou scripts capables de communiquer avec l'utilisateur via des pages HTML construites dynamiquement.

Les applications du coté serveur doivent respecter certains standards d'interfaçage dont principalement la norme CGI (Common Gateway Interface) et la norme ISAPI (Internet Server Application Programming Interface). Ces applications respectant la norme ISAPI sont compilées en tant que DLL (Dynamic-Link Libraries) et chargées par le serveur Web à son démarrage.

La programmation des applications Web consiste à concevoir et à écrire des programmes résidants dans un répertoire, défini comme étant un répertoire de scripts chez le serveur, et recevant leurs commande d'activation d'une page Web. Généralement cette page WEB utilise un formulaire HTML ou des liens directs pour lancer le programme ou le script. Le formulaire HTML est devenu le moyen le plus utilisé pour l'envoi des données sur internet, car il permet de mettre en place des fenêtres de saisie, des cases à cocher, des boutons radios, etc. ce qui facilite la récupération des données.

D'une manière succincte, la démarche à suivre pour la conception d'une application sur le WWW comporte les étapes suivantes :

- ★ La création des pages Web contenant des références à des programmes exécutables, en utilisant la norme HTML.
- ★ Les références sont généralement des liens directs d'activation des programmes (scripts) ou des boutons de soumission de formulaires qui seront remplis et à envoyés aux programmes de l'application.
- ★ La création d'un ensemble de programmes qui reçoivent leur commande d'activation par des références dans des pages Web. Ces programmes ont généralement une interface standard du type CGI ou ISAPI, et résident dans un répertoire définie comme étant un répertoire de scripts dans l'arborescence des répertoires du serveur (généralement cgi-bin, scripts, etc.).

Les programmes reçoivent en entrée des arguments dits variables d'environnement qu'ils traitent, et génèrent en sortie des en-têtes de réponses compréhensibles par le serveur HTTP. Dans le cas où un programme ne fournit rien en sortie, la réponse sera vide.

La création des programmes (scripts) se fait par un langage permettant la définition des interfaces (ISAPI, CGI,...) des scripts (Java, Perl, Delphi ou autres).

La création d'une telle application nécessite un certain niveau de programmation ainsi qu'une certaine manipulation des serveurs (installation, configuration, etc).

En conclusion, les applications Web sont des applications multitiers qui se caractérise par :

- ★ Des clients identiques consistant en des navigateurs internet
- ★ Des serveurs qui se composent principalement d'un serveur Web à l'écoute sur un port de communication et de programmes d'application avec éventuellement une base de données
- ★ Des échanges entre les tiers qui respectent des normes et des protocoles standards