Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche

researchportal.unamur.be

University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Applications distribuées, description et principes de développement

Karemera, Hugues

Award date:
1993

Awarding institution:
Universite de Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 03. Jul. 2025

FACULTES
UNIVERSITAIRES
N.D. DE LA PAIX

NAMUR

INSTITUT D'INFORMATIQUE

Applications distribuées,
description et principes de développement

par Hugues KAREMERA

Promoteur:
Professeur F. Bodart

Mémoire présenté en vue
de l'obtention du titre
de Licencié et Maître
en Informatique.

Année académique 1992-1993

Facultés Universitaires Notre-Dame de la Paix

Institut d'Informatique

Rue Grandgagnage, 21B

B-5000 NAMUR

Applications distribuées,
description et principes de développement

Hugues KAREMERA

Résumé

Il existe plusieurs modalités de distribution des traitements d'une application de gestion. Ces notions de répartition recouvrant des aspects techniques, logiques et organisationnels. Notre travail consiste à structurer ces éléments, en détaillant les spécificités de toute infrastructure d'application distribuée, et en définissant différents modèles de distribution de leurs traitements (architectures dites de "client/serveur", de "cooperative processing" et de "remote presentation processing"). En l'absence de démarche systématique, cette étude propose également quelques critères utiles à l'analyste d'un projet de construction d'une architecture distribuée.

Abstract

In information technology, the processes of an application can be distributed in several different ways. Distribution concepts involve a mix of technical, logical and organizational aspects. The purpose of this work is to structure those elements by a detailed study of the aspects specific to any implementation of a distributed application and by defining the different distributed processing models ("client/server", "cooperative-processing" and "remote presentation processing" architectures). As a systematic and practical methodology is missing, an additional purpose of this work is to propose some criteria helpful to any analyst involved in the implementation of a distributed architecture.

Mémoire présenté en vue de l'obtention du titre
de Licencié et Maître en Informatique

Septembre 1993

Promoteur: Professeur F. Bodart

Je tiens à remercier Monsieur le Professeur Bodart, en qui, plus qu'un promoteur, j'ai trouvé un véritable soutien tout au long de ce mémoire.

Je souhaite également exprimer ma reconnaissance à Monsieur André Gonay, mon maître de stage, qui non seulement m'a guidé dans ce travail, mais a également fait preuve d'une patience remarquable à mon égard.

Derrière ce travail, se trouvent aussi de nombreuses personnes qui de, près ou de loin, m'ont aidé, encouragé; qu'elles trouvent ici tous mes remerciements.

A mes parents, sans qui je n'aurais pu réaliser ces études.

Table des matières

Table des matières

Introduction **1**

Chapitre 1: Infrastructure des applications distribuées **3**

1.1 Introduction 3

1.2 Composants matériels et logiciels 6

1.3. La couche Middleware 8
1.3.1. Fonctions de communication 10
Fonctions de l'Appelant 11
Fonctions d'Interface 12
Fonctions de l'Appelé 13

1.3.2 Techniques de middleware- Illustrations 14
1.3.2.1 DCE de OSF 14
1.3.2.2 SQLNetwork de Gupta Technologies 20
1.3.2.3 Access/kit de ARM-SIE 23

1.3.3 Les API de middleware 25

Chapitre 2: Architecture des applications distribuées **28**

2.1 Introduction 28

2.2 Proposition d'architecture des applications de gestion 30
2.2.1 Principes de définition 30
Principe de modularité 30
Principe d'indépendance 30

2.2.2 Description de l'architecture 32
2.2.2.1 Les modules 33
Presentation logic 33
Data logic 34
Business logic 35
2.2.2.2 Le dialogue interne 36
2.2.2.3. Contrôle de l'application 39

2.2.3 Comparaison avec d'autres architectures 41
Architectures hiérarchiques 41
Architecture du projet TRIDENT 42

2.3. Classes d'architectures distribuées 46
2.3.1. Systèmes composites et architectures distribuées 46
Système composite 46
Architecture distribuée 47
Conséquences de la distribution d'une application de gestion 48

2.3.2. Trois classes d'architectures distribuées 51
2.3.2.1 Client/server processing 52
2.3.2.2 Cooperative processing 58
2.3.2.3 Remote Presentation processing 61

Chapitre 3: Critères de distribution

3.1. Introduction 63

3.2. Le type d'application 65
3.2.1. Applications transactionnelles 66
3.2.2. Applications analytiques 71

3.3. Critère d'intégration 76
3.3.1. Information de type "donnée" 76
3.3.2. Information de type "événementiel" 79

3.4. Critères de rightsizing 83
3.4.1 Rapport coût/efficacité 85
3.4.2 Considérations politiques et organisationnelles 87
3.4.3 Conseils méthodologiques 88

Chapitre 4: Etude de cas

Chapitre 4: Etude de cas

4.1. Introduction 90

4.2 Présentation du cas 91

4.3. Critères de distribution 93
4.3.1. Le type d'application 93

4.3.2. Critère d'intégration 93
Solution idéale 94
Solution implémentée 95
Solution plus efficace 98

4.3.3. Critères de rightsizing 99

4.4. Infrastructure 99

Conclusion 100

Abréviations

Liste des abréviations

- A.D.: Application Distribuée
- APPC: Advanced Program to Program Communication
- API: Application Programming Interface
- b.d.: base de données
- B.L.: Business Logic
- C.I.: Contraintes d'Intégrité
- C/S: Client/serveur
- DCE: Distributed Computing Environment
- DDE: Dynamic Data Exchange
- D.L.: Data Logic
- DME: Distributed Managment Environment
- GPAO: Gestion de Production Assistée par Ordinateur
- GUI: Graphic User Interfaces
- ISO: International Satndards Organization
- OSF: Open Systems Interconnection
- P.C.: Personal Computer
- RPC: Remote Procedure Call
- P.L.: Presentation Logic
- S.I.A.D.: Système d'Information d'Aide à la Décision
- SQL: Structured Query Language
- TRIDENT: Tools foR Interactive Development Environment

Introduction

Introduction

La première idée de ce travail consistait à définir les applications de gestion qualifiées de "*client/serveur*", et à préciser leurs aspects techniques, logiques et organisationnels.

Une première lecture de la littérature nous a révélé combien ce terme s'utilise, souvent de manière confuse, pour recouvrir des domaines tels que, pour n'en citer que quelques uns, les bases de données distribuées, les traitements coopératifs ou encore les traitements distribués.

La définition la plus communément admise caractérise le paradigme client/serveur comme *la communication entre deux programmes où l'un, appelé "client", envoit une requête à un autre programme, nommé "serveur", qui exécute le traitement demandé et renvoit le résultat* à son client.

Même si la notion de répartition de traitements et de requête d'un service apparaît, dans le cadre de notre analyse des applications de gestion, cette définition est encore trop générale.
En fait, le seul principe certain sur lequel nous pouvons nous baser était la distribution du code d'une même application sur différents ordinateurs.

Dès lors, partant de ce postulat, nous avons préféré parler d' "applications distribuées", et en préciser la définition tout au long de notre étude, où comme le définit [Davies83]:

> "*Actually, a definition is useful if it helps one to infer principles and guidelines for the design of distributed systems*".

A cette fin, nous avons procédé à un examen de différentes applications "distribuées", développées par la firme Ciba-Geigy S.A. Nous avons commencé par une étude de l'infrastructure technique des systèmes distribués, pour ensuite détailler les architectures de traitements de ces applications.

De ces analyses, nous avons déduit quelques fondements théoriques devant guider le développeur d'une application distribuée dans la conception de son architecture et le choix de l'environnement matériel le plus adéquat.

Ce mémoire est composé de quatre chapitres.

Le premier est consacré à une **étude de l'infrastructure** des systèmes distribués, c'est-à-dire l'ensemble des moyens matériels et logiciels utilisés par les différents programmes d'une application pour communiquer. Nous y détaillerons particulièrement les éléments techniques distinguant les applications distribuées de tout autre système informatique.

Le deuxième chapitre propose une **architecture générale** des traitements de toute application de gestion, indépendamment de son infrastructure.
De cette proposition, nous déduisons **différents modèles de distribution** des traitements, correspondant à la répartition des composants de l'architecture générale.

Dans le troisième chapitre, nous dégageons quelques principes, appelés "**critères de distribution**", qui doivent guider l'informaticien dans la conception d'une application distribuée.

Nous terminons, dans le quatrième chapitre, par une **étude de cas** d'une application développée par la firme Ciba-Geigy S.A., illustrant les différents aspects des systèmes distribués décrits tout au long de ce travail.

Chapitre 1

Chapitre 1: Infrastructure des applications distribuées

1.1 Introduction

L'information et sa circulation dans une organisation constituent des éléments clés de la réussite de toute entreprise. Ces dernières années, le développement de nouvelles technologies de communication, de plus en plus rapides et fiables, associé à l'avènement de la micro-informatique, ont bouleversé le monde de l'information.

Les entreprises conscientes de disposer d'un potentiel de communication énorme, n'en sont devenues que plus exigeantes, et ont progressivement revus leurs stratégies informatiques.

Leurs premières exigences étaient de permettre à leurs systèmes informatiques, utilisant du matériel et des logiciels d'exploitation de différents constructeurs, de s'échanger des informations. Il s'agissait donc de développer des **systèmes dits "ouverts"**, par lesquels des applications peuvent communiquer indépendamment du matériel utilisé.

Plusieurs organisations internationales se sont penchées sur ces problèmes et, en réponse, ont défini différentes conventions, appelées *protocoles*, imposant des règles standards de communication. Ainsi sont nés des modèles tels que OSI (Open Systems Interconnection), développé par l'ISO (International Standards Organization), qui définit 7 niveaux de protocoles standards.

Si, par ces modèles, des applications autonomes peuvent s'échanger des informations indépendamment du matériel utilisé, bien vite de nouveaux besoins de communication sont apparus. La volonté actuelle du monde industriel est de pouvoir développer de véritables systèmes d'information à l'échelle de leur entreprise, c'est-à-dire :

- au niveau des données: permettre aux utilisateurs d'accéder, manipuler et transmettre des informations à travers toute l'entreprise;

- au niveau des traitements: pouvoir répartir les traitements d'une même application sur les machines les plus appropriées pour les exécuter.

Ainsi, des besoins d'interconnexion de systèmes autonomes, les entreprises sont passées à la nécessité de développer des **applications dites "distribuées"**, où les composants d'une même application sont répartis sur différents environnements matériels.

Définition d'une application distribuée (A.D.)

Une application distribuée sera vue comme un **ensemble de programmes, localisés sur des plates-formes informatiques distinctes** - ensembles de machines, systèmes d'exploitation et logiciels de communication - **qui s'échangent des informations (communiquent) pendant leur exécution**.

Si, avec les systèmes distribués, nous restons dans le domaine des "open systems", le maître mot n'est plus ouverture mais ***interopérabilité***. Interopérer signifie que différentes plates-formes vont collaborer "comme si", (pour le programmeur et l'utilisateur), l'ensemble de l'application se trouvait sur une seule et même machine.

Cette interopérabilité va se traduire au niveau de l'infrastructure, c'est à dire l'inventaire des moyens matériels et logiciels utilisés pour communiquer, du système distribué par:
une couche logicielle, appelée ***middleware***, permettant aux différents programmes de l'application de communiquer:
- **indépendamment** du matériel utilisé et de leur localisation dans le système,
- et de manière **transparente** pour le programmeur et l'utilisateur final.

Précisons qu'il nous importe peu, lors de ce premier chapitre, de connaître les tâches implémentées par les différents programmes de l'application distribuée (A.D.). Par une étude de l'infrastructure des A.D., on entend seulement analyser les implications techniques de la distribution du code d'une même application sur différentes machines .

La figure 1.1 présente cette infrastructure, selon un schéma proche de celui proposé par [Gartn92], où on retrouve:
- les couches "hardware" et "communication" comme moyens matériels,
- les couches "middleware", "API" et "conversation facilities" comme éléments logiciels.

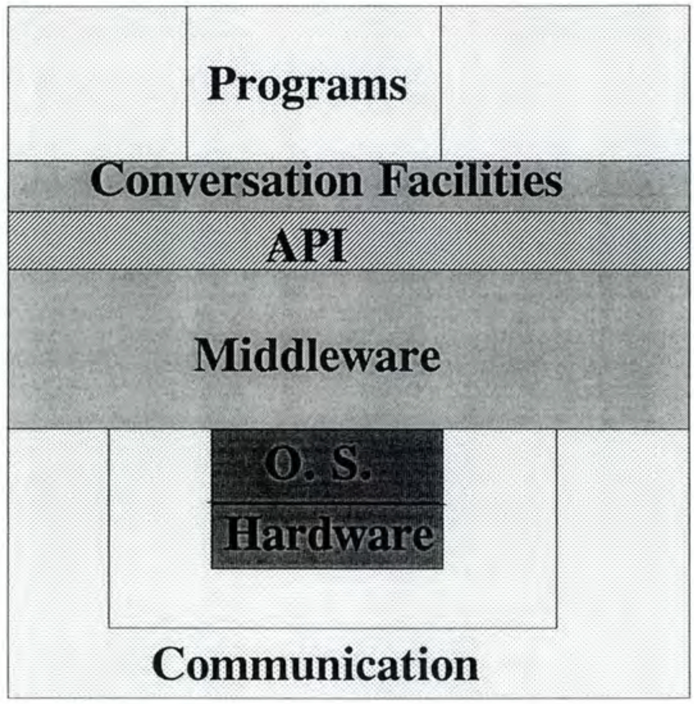


Figure 1.1: Infrastructure des A.D.

1.2 Composants matériels et logiciels

Le **matériel** utilisé par les programmes d'une A.D. comprenant les **outils classiques** de télécommunication, présentés notamment dans [Stall91] et [Tanen90], nous ne les détaillerons pas.

Nous considérons simplement que l'infrastructure matérielle d'une A.D. se compose de différents ordinateurs (P.C., stations de travail, mainframes, ...), reliés entre eux par des outils (répétiteurs, contrôleurs, ...) et réseaux (LAN, MAN ou WAN) de communication.

Précisons, tout de même que nous exclurons les infrastructures se limitant à une machine multi-processeurs. Bien qu'elles soient parfois reprises dans les typologies des systèmes distribués, notamment par [Coulo88], les applications dont les traitements s'exécuteraient sur différents processeurs au sein d'un même ordinateur sortent du cadre de notre étude (et de la définition donnée en 1.1).

Si, sur le plan du matériel une A.D. se ramène à un "simple" cas de communication entre ordinateurs, c'est au niveau du logiciel et des échanges de données entre programmes qu'apparaît un élément fondamental: la **transparence de communication**.

Les programmes d'une A.D. doivent s'échanger des informations de façon totalement transparente pour l'utilisateur et le programmeur, c'est-à-dire "comme si" l'ensemble des données et des traitements de l'application se trouvaient sur une seule et même machine.

Précision de la définition d'une A.D.

Une application distribuée peut être vue comme un ensemble de programmes qui s'exécutent sur des plates-formes distinctes, et communiquent de façon **transparente pour l'utilisateur et le programmeur**.

Pour le "end-user", cela implique qu'il **ne doit pas se soucier ni même peut-être avoir connaissance de la localisation des données et des traitements de son application sur différentes machines**.

Par exemple, on pourrait imaginer une A.D. où l'utilisateur accéderait et traiterait, à partir de son poste de travail, des données se trouvant sur n'importe quelle autre machine du système distribué. Il le ferait de la même manière qu'il accéderait à des données locales.

Pour le programmeur, même si bien évidemment il doit avoir conscience de la distribution des traitements, la transparence de communication se traduit **en un véritable environnement de programmation**.
Cet environnement est un ensemble de logiciels, souvent qualifiés de "**middleware**" ([CA91], [Gartn92]), permettant au développeur de répartir le code de son application sur différents ordinateurs, sans devoir gérer les communications entre ces machines.

Comme exemple, reprenons le cas d'un programme voulant accéder et traiter des données se trouvant sur un autre ordinateur.
Si on envisage le problème comme un échange d'information entre systèmes autonomes, la solution classique est de réaliser un transfert de fichiers ("file transfer") entre les deux systèmes. Cela implique que le programmeur doit entre autres:

- localiser la machine contenant les données demandées,

- connaître le format du fichier reprenant ces données,

- éventuellement le traduire en un format compatible pour les deux systèmes,

- etc ...

Une fois le fichier transféré, l'application peut alors seulement traiter les données; et ensuite, éventuellement, renvoyer ce fichier au système d'origine.

Dans le cadre d'une A.D., tous ces problèmes ne se posent pas au développeur. Il doit pouvoir dissocier le code de son application chargé des traitements de données, et l'implanter sur la machine qui stocke ces données. Son application peut être ainsi physiquement divisée en programmes répartis sur des machines distinctes. Tout au long de l'exécution de l'A.D., ces programmes vont s'échanger des informations via un environnement logiciel déchargeant le programmeur de toutes les tâches précitées liées au "file transfer" (telles que la localisation des machines, la conversion de format de présentation, ...).

1.3. La couche Middleware

L'organisation Open Software Foundation (OSF), regroupant quelques grands constructeurs informatiques, a développé un environnement de middleware, appelé Distributed Computing Environment (DCE), et le définit comme étant:

"an integrated software environment that makes a network of systems from a variety of vendors appear as a consistent, unified system. It masks the technical complexities of the network giving users transparent access to diverse network resources" [OSF91].

La couche middleware se compose donc d'un ensemble de logiciels, se situant entre les programmes d'application et le(s) système(s) d'exploitation, **garantissant la transparence de communication des programmes de l'A.D.**

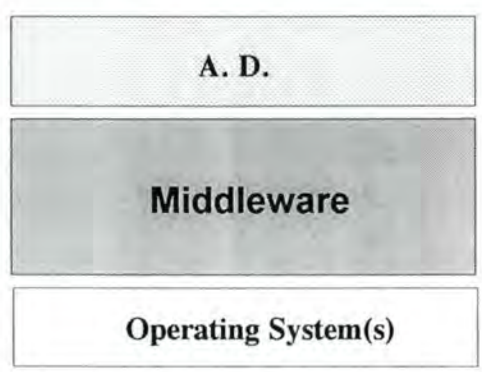


Figure 1.2: Eléments logiciels d'une A.D.

Bien que cette couche puisse être développée sur mesure par les programmeurs d'application, le plus souvent elle se compose de logiciels disponibles sur le marché informatique, tels que:

- SQLNetwork de Gupta Technologies,

- DCE de OSF,

- EDA/SQL de Information Builders,

- Access/kit de ARM-SIE,

- Sybase de Sybase Inc.

Si l'objectif est toujours le même pour ces logiciels de middleware, à savoir assurer la transparence de communication entre plates-formes distinctes, par contre les services qu'ils offrent et les techniques utilisées peuvent varier d'un produit à l'autre.

En fait, [Berso92] répertorie ces techniques de middleware en trois catégories:

- les ***Remote Procedure Calls***,
- les ***SQL Interactions***,
- et les ***Pipes***, ou moyens de communication inter-processus.

Nous détaillerons ces techniques et leur contexte d'utilisation (cfr. 1.3.2.) au travers de trois produits:
- *DCE* de OSF, pour les Remote Procedure Calls,
- *SQLNetwork* de Gupta Technologies, comme technique de SQL Interaction,
- *Access/kit* de ARM-SIE, comme moyen de communication inter-processus.

Avant de présenter ces logiciels, nous allons d'abord examiner les ***fonctions de communication,*** communes à tous les environnement de middleware. Ensuite nous expliquerons au travers de trois illustrations la manière dont les différentes techniques de middleware prennent en charge ces fonctions de communication.

1.3.1. Fonctions de communication

Avant de synthétiser les fonctions de tout environnement de middleware, nous allons définir un modèle très simple de communication entre programmes d'une A.D. Nous considérons que tout échange se réalise en deux étapes:

- tout d'abord, un programme *Appelant* envoit un message "***requête***" à un autre programme de l'A.D.,
  ce message contient une demande d'exécution d'un traitement, et les informations nécessaires à celui-ci,

- après un certain temps d'exécution, le programme *Appelé* envoit un message "***résultat***" au programme Appelant;
  ce message contient donc le (ou les) résultat(s) d'une requête.

Figure 1.3: Communication entre programmes

A priori, à tout moment, n'importe quel programme d'une A.D. peut être Appelant ou Appelé. Nous montrerons par la suite, lors de l'étude des architectures des systèmes distribués (cfr 2.3.2), qu'**a posteriori** cela ne se vérifie pas dans tous les cas de figure.

Pour permettre de rendre de tels échanges *requête-résultat* transparents au programmeur et à l'utilisateur final, le middleware doit exécuter différentes fonctions de communication, que nous regrouperons en trois catégories:

- les fonctions de l'Appelant,
- les fonctions de l'Interface,
- les fonctions de l'Appelé.

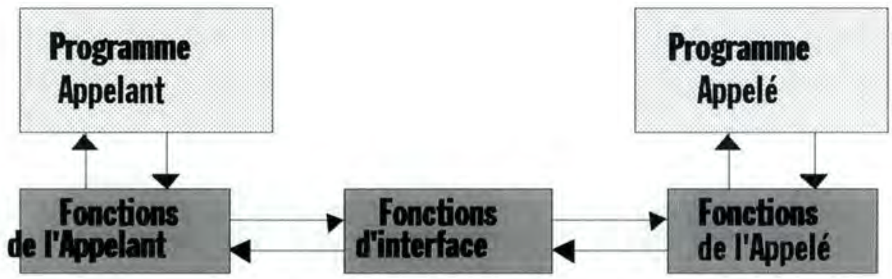


Figure 1.4: Fonctions de communication de la couche middleware

Insistons sur le fait que ces fonctions ne sont pas spécifiques à l'un ou l'autre produit ou technique de middleware. Elles se retrouvent dans chacun d'eux, et quel que soit l'environnement de programmation proposé.

Nous allons présenter brièvement ces fonctions, et les resituer dans l'architecture standard de communication de l'ISO (modèle OSI).
L'objectif n'est pas d'expliquer ces protocoles, mais de montrer que tout environnement de middleware prend en charge les services du modèle OSI, de manière transparente pour le programmeur.

Fonctions de l'Appelant

• Objectif

Diriger les requêtes du programme Appelant vers l'Appelé, et en réceptionner les résultats.

Les fonctions de communication sont celles classiquement attribuées aux couches **Présentation, Session et Transport** du modèle OSI, à savoir essentiellement:

- mettre les données de la requête, c'est-à-dire le traitement demandé et ses paramètres, sous un format standard de présentation,

- gérer (créer, maintenir et libérer) des connexions logiques entre programmes,

- garantir un transfert fiable de l'information,

- réceptionner les résultats (et gérer leur ordre de réception).

• **Acteur de middleware**

Dans tout environnement de middleware, ces fonctions de communication sont prises en charge par un logiciel localisé dans la même machine que le programme Appelant.

Fonctions d'Interface

• **Objectif**

Fournir le "chemin" et l'interface de communication entre les programmes voulant s'échanger des informations.

• **Fonctions**

Dans le modèle OSI, ce sont typiquement les tâches attribuées à ce que l'ISO appelle Internetworking Unit (ou parfois Gateway). [Stall91] définit son rôle de la façon suivante:

> *"Internetworking Units provide a communication path, and perform the necessary relaying and routing functions so that data can be exchanged between devices attached to different subnetworks".*

Ces fonctions sont donc les services typiquement reconnus au niveau **Réseau** (et ses couches inférieures) du modèle OSI, c'est-à-dire:

- la localisation des programmes,
- la conversion des protocoles réseaux,
- la gestion physique des connexions.

• **Acteur de middleware**

Il s'agit d'un logiciel localisé

- soit dans une machine du réseau spécialement dédiée à la communication entre les deux programmes (cette machine est parfois appelée "*platform to platform communication segment*"),

- soit dans chaque ordinateur exécutant les programmes de l'A.D.

Fonctions de l'Appelé

Corollaire des fonctions de l'Appelant, à savoir:

- la conversion de présentation de données,

- la gestion de sessions logiques de communication,

- le transfert fiable des informations.

A ces tâches, il faut encore rajouter une gestion de la concurrence des requêtes pouvant arriver simultanément, c'est-à-dire:

- gérer une file d'attente, où stocker les requêtes,

-gérer une politique d'exécution de ces requêtes suivant un ordre de priorités, soit déterminées par le concepteur de l'application, soit définies par le produit de middleware.

Toutes ces fonctions de communication sont implémentées par un logiciel se trouvant sur la même machine que le programme Appelé.

1.3.2 Techniques de middleware- Illustrations

Pour [Berso92], tout environnement de middleware se base sur une des trois techniques suivantes de connexion de programmes d'une A.D.:

- les **Remote Procedure Calls**, "*a mechanism by which one process can execute another process (subroutine) that resides on a different, usually remote, system*",

- les **SQL Interactions**, "*a mechanism for passing the Structured Query Language (SQL) requests and associated data from one process to another process*",

- les **Pipes**, "*a connection-oriented mechanism that passes data from one process to another*".

Nous détaillerons ces mécanismes au travers de trois produits:
- DCE de OSF,
- SQL Network de Gupta Technologies,
- Access/kit de ARM-SIE.

L'utilisation de l'une ou l'autre technique dépend du contexte de l'A.D.; plus précisément de la nature des traitements implémentés par les différents programmes distribués, et leurs échanges.
Nous analyserons ces éléments au deuxième chapitre (cfr 2.3.2), sous le terme d'architectures de traitements d'une A.D., où nous préciserons pour chaque architecture la (ou les) technique(s) de middleware qui semble(nt) la (les) plus appropriée(s).

1.3.2.1 DCE de OSF

Objectif

Par son produit DCE (Distributed Computing Environment), OSF (Open Software Foundation) est très ambitieuse puisqu'elle tente d'imposer un **standard de facto** en matière de logiciels de middleware. Etant donné l'importance croissante de cet environnement, nous nous attarderons quelque peu sur l'ensemble du produit, avant de détailler la technique du Remote Procedure Call en elle-même.

Présentation générale

OSF, organisation fondée en 1988 regroupant initialement huit grands constructeurs (HP, DEC, IBM, Bull, Philips, Siemens, Appollo, Nixdorf), visait tout d'abord le marché des systèmes d'exploitation, en mettant au point une version UNIX (appelée OSF/1) ouverte et commune à ses différents membres.

Progressivement, OSF a étendu ses activités aux environnements de middleware, via son produit DCE, en offrant six types de services au programmeur et à l'utilisateur de toute A.D.:

- les ***Remote Procedure Calls***,
- le ***Threads Sercice***,
- le ***Directory Service***,
- le ***Security Service***,
- le ***Time Service***,

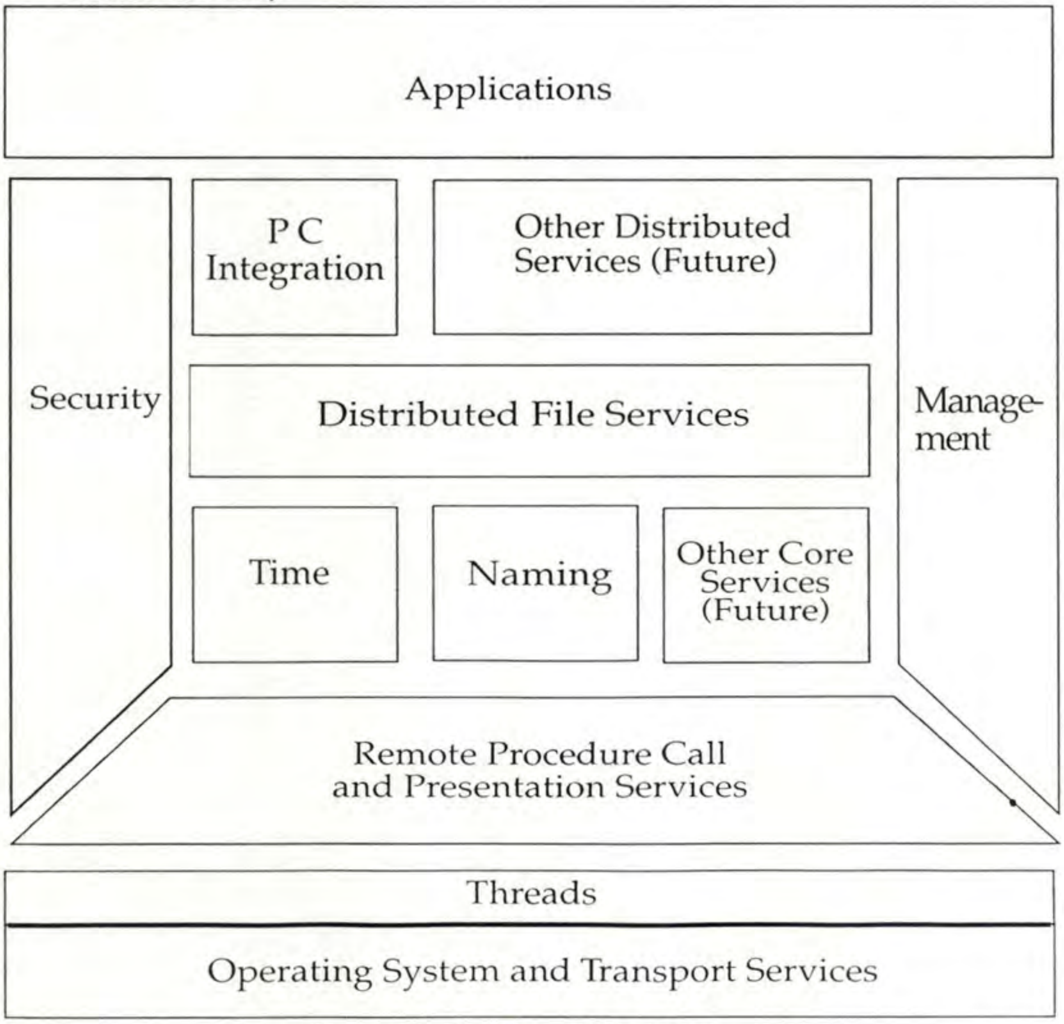


Figure 1.5: Services d'OSF/DCE

Ces services, fournis sous forme de code source C, ne sont pas des produits propriétaires d'OSF, mais le résultat d'appels d'offre au public.

Pour chaque nouveau composant de son environnement DCE, OSF émet ce qu'elle appelle un *Request For Technology* (RFT) au marché informatique. Ensuite, un comité de sélection choisit la technologie la plus appropriée (justifiée dans un rapport public), et l'intègre à son environnement.

Les **Remote Procedure Calls** (RPC) constituent l'**élément central de DCE** (qui a adopté, par son Request For Technology, le mécanisme de RPC de Appollo's NCS version 2..0), tous les autres services étant construits sur base de ce mécanisme.

• ***Threads Service***

Ce service permet de dissocier l'exécution d'un programme en sous-processus (appelés "*threads*", c'est-à-dire des flux séquentiels d'exécution d'instructions), qui pourront s'exécuter en parallèle tout en se partageant un seul et même espace d'adressage.

Ce mécanisme doit améliorer les performances et l'utilisation des ressources, et surtout permettre l'asynchronisme des RPC.

Ce service de DCE se fonde sur le produit Concert Multithread Architecture (CMA) de Digital.

• ***Directory Service***

Le Directory Service permet d'accéder à n'importe quel élément (appelé "objet") du système distribué simplement en le nommant, sans connaître son adresse ou sa localisation dans le système (un tel objet peut être un ordinateur, un fichier, une imprimante, ...).

Ce service est basé sur X/Open Directory Services.

• ***Security Service***

Le mécanisme d'authentification et d'autorisation d'accès aux ressources du système choisi par OSF, est le service de sécurité Kerberos du M.I.T. adopté par H.P.

• *Time Service*

L'objectif de ce service est de synchroniser l'ensemble des horloges du système distribué en fonction d'un seul temps de référence pour l'ensemble du système.

OSF a choisi le Time Synchronization Service de Digital.

• ***Distributed File System***

Il permet directement à l'utilisateur final d'accéder à des données de n'importe quel fichier de l'environnement distribué, à les manipuler et modifier sans aucune programmation préalable, .

Bien évidemment OSF a construit cette possibilité sur base des 5 services expliqués précédemment, et y a. intégré le produit NFS de Sun.

Signalons également que, parallèlement à cet environnement de programmation distribuée, OSF fournit également un produit appelé **DME** (Distributed Management Environment), d'assistance à la gestion (l'administration) de l'infrastructure distribuée (par exemple la surveillance des pannes de machine, le rétablissement de la configuration, le suivi des systèmes d'exploitation, ...).

Les Remote Procedure Calls (RPC)

Comme leur nom l'indique, les RPC permettent à un programme d'appeler et d'exécuter une procédure se trouvant sur une machine distincte du code appelant.

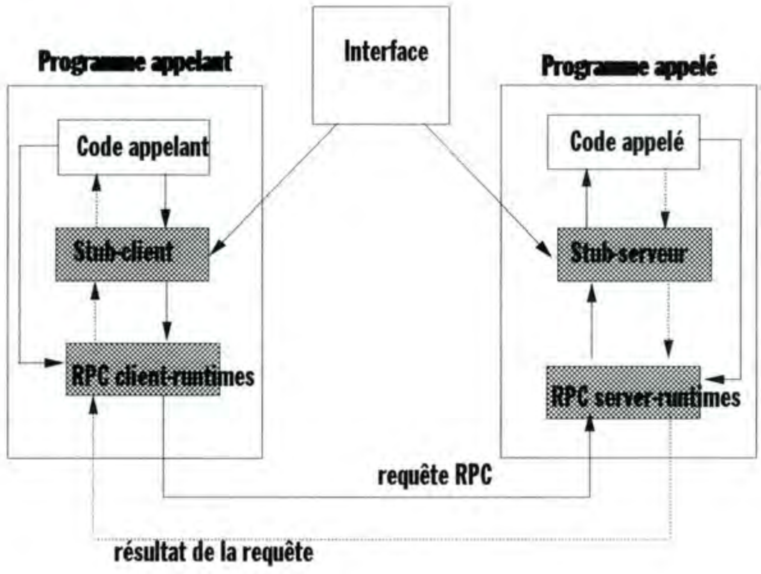


Figure 1.6: Technique de RPC

Nous allons décrire la procédure de construction d'un RPC car elle explique clairement cette technique. Cette procédure n'est pas propre à OSF, mais valable pour n'importe quelle technique de RPC.

Tout d'abord, le programmeur doit définir une **interface**, comprenant les noms des procédures "remote", les types de leurs paramètres, et générer un identifiant de cette interface.

Ensuite, il doit compiler l'interface, cela génère des procédures appelées "***stub-client***" et "***stub-server***". Ces procédures servent essentiellement à gérer une présentation standard des échanges entre programmes, et aussi d'appeler des ***RPC runtimes*** librairies (pour, par exemple, gérer la concurrence de requêtes).

Le concepteur peut alors développer le code de ses programmes Appelant et Appelé. La seule contrainte est de se référer, au début du code Appelant, aux librairies RPC runtimes propres au middleware choisi; pour interroger, sur base du nom du programme appelant, une b.d. (dite "***Name Service Database***") donnant l'adresse physique du code appelé.

Une fois compilés, les codes Appelant et Appelé sont "linkés" avec respectivement les stub-client et stub-server procedures, et les RPC runtime routines.

Fonctions de communication

Pour résumer cette technique de RPC, nous pouvons très facilement replacer les différentes fonctions de communication que nous avons définies (cfr 1.3.1) pour tout environnement de middleware.

Les fonctions de l'**Appelant** sont prises en charge:

- par le *stub-client,* pour la gestion des formats de présentation (dans la terminologie des RPC, cette fonction est appelée "*(de)marshalling*"),

- et par les *RPC client runtimes,* pour la gestion de sessions de communication (appelé "*binding*") et le transfert fiable de l'information.

Les fonctions d'**Interface** (pour rappel, la localisation des programmes, la conversion de protocoles réseaux et la gestion des connexions physiques) sont exécutées par les *RPC server et client runtimes*, et via également le *Name Service Database* (se trouvant sur n'importe quelle machine du système distribué), pour déterminer l'adresse physique du code appelé.

Les fonctions de l'**Appelé** sont prises en charge par:
- le *stub-server*, pour les formats de présentation,
- les *RPC sever runtimes*, pour la gestion de sessions de communication et le transport.

La gestion de requêtes concurrentes est:
- soit automatiquement prise en charge par les RPC sever runtimes,
- soit par le programmeur de l'application, qui se réfère explicitement dans son code appelé aux RPC server runtimes, pour gérer lui-même les priorités et politiques d'exécution des requêtes.

Conséquences

Pour conclure, nous pouvons dire que le mécanisme de RPC offre, outre la puissance du principe même de pouvoir déclencher un processus à distance, un **avantage de neutralité.**
La généralité de cette technique s'adapte très bien aux objectifs d'OSF/DCE, de devenir un standard de facto des environnements middleware.

Les techniques de RPC sont totalement indépendantes:
- des programmes développés; peu importe que l'Appelant ou l'Appelé soit chargé de gestion de données, de dialogue avec l'utilisateur, de calculs, ...
- du matériel et systèmes d'exploitation utilisés.

1.3.2.2 SQLNetwork de Gupta Technologies

Présentation générale

La technique de middleware offerte par le produit SQLNetwork se range dans la catégorie ***SQL Interactions*** de [Berso92]. Elle permet donc uniquement au programme Appelant de demander l'exécution de requêtes SQL sur des données. Et le programme Appelé, se trouvant sur une machine reprenant les données de l'application, exécute les requêtes via un SGBD relationnel.

L'environnement SQLNetwork, produit propriétaire de Gupta Technologies, se compose de trois logiciels:
- SQLRouter,
- SQLGateway,
- SQLHost.

Fonctions de communication

Notons que nous restituerons nos fonctions de communication (cfr 1.3.1) uniquement pour le produit SQLNetwork, ce qui ne peut pas être généralisé à tout environnement basé sur la technique de SQL Interactions.

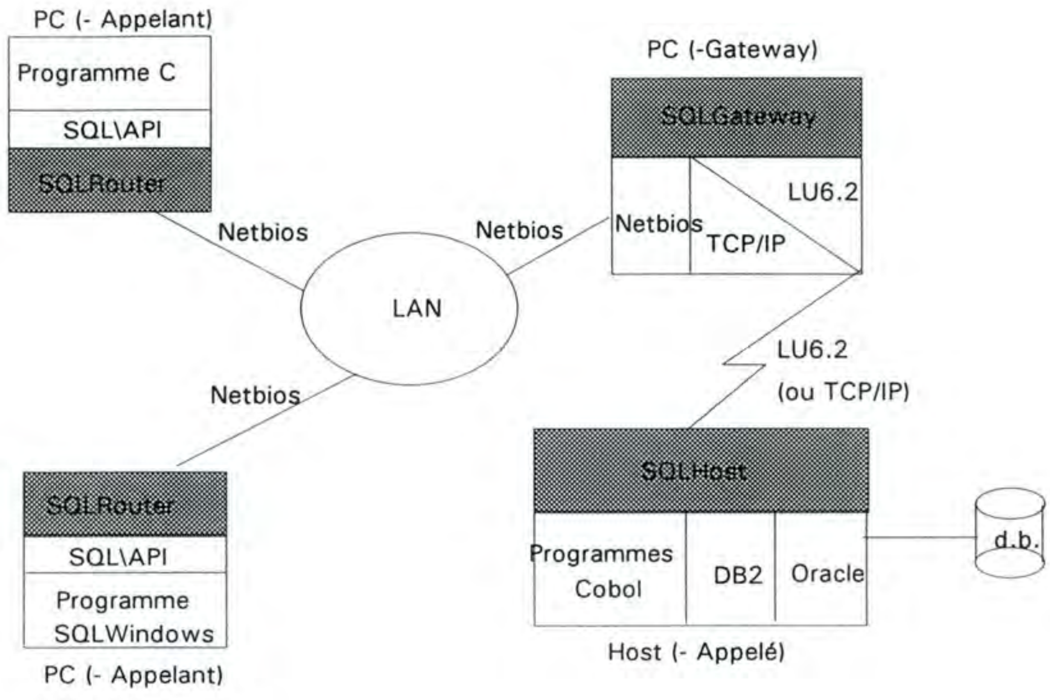


Figure 1.7: Technique de SQLNetwork

Les **fonctions de l'Appelant** sont prises en charge par le **logiciel SQLRouter** s'exécutant sur la même machine, qui doit nécessairement être un P.C., que le programme Appelant.
Ce logiciel s'occupe pour chaque requête SQL:
- d'identifier une connexion entre programmes,
- de constituer un message reprenant la requête, ses paramètres et le curseur (c'est-à-dire l'identification de la connexion),
- et de transmettre le message sur un réseau local.

Les **fonctions d'Interface** sont exécutées par le **logiciel SQLGateway**, se trouvant sur un P.C. du même réseau local que le code Appelant.
Ce logiciel se charge essentiellement:
- de réceptionner les requêtes de programme Appelant ou les résultats de l'Appelé,
- localiser le destinataire des informations,
- convertir les protocoles réseaux utilisés (nethbios, LU6.2 ou TCP/IP) et transmettre les informations.

Les **fonctions de l'Appelé** sont exécutées, sur la même machine que le programme Appelé, par le logiciel SQLHost. Cette machine est généralement un mainframe (par exemple, associé au SGBD DB2), ou un midrange (avec SGBD Oracle).

Ce logiciel se charge de:
- réceptionner les requêtes du GTWY
- traduire les différents formats SQL (convertir du SQL général en format propre au SGBD relationnel utilisé, tel que DB2 ou Oracle),
- gérer la concurrence des requêtes, en instanciant un processus de SQLHost par requête soumise,
- transmettre le résultat au SQLGTWY.

Conséquences

Par sa nature, la technique du SQL Interaction **limite** les programmes Appelés **à des traitements sur une base de données**. Nous préciserons ces traitements au deuxième chapitre (cfr 2.3.2.1.), mais signalons tout de même que cette gestion ne se borne pas nécessairement à l'exécution de requêtes SQL élémentaires de consultation, mise à jour ou suppression de tables.

Certains produits, basés sur les SQL Interactions, permettent d'enrichir les tâches du programme Appelé via des techniques telles que:
- les ***triggered processus***, de Sybase Inc.,
- les ***callback routines***, de Gupta Technologies.

Par les ***triggered processus*** de Sybase, le programmeur peut développer et stocker sur la machine appelée des procédures qui permettent de traduire une requête SQL de l'Appelant en une série de requêtes SQL sur les données distantes.

Par exemple, ces procédures permettent de localiser les traitements de vérification de C.I. sur la machine qui reprend les données de l'application. Une requête du programme Appelant, telle que la mise à jour d'une table, va générer une série de traitements du programme Appelé, comme la vérification d'identifiants étrangers, la supression d'éléments de différentes tables.

Le produit SQLNetwork, par son mécanisme de ***callback routine***, permet également de "complexifier" le programme Appelé. Par cette technique, l'Appelé ne se limite pas uniquement à des traitements sur une b.d. relationnelle. Néanmoins l'expression de la requête de l'Appelant doit toujours se faire par un langage SQL.

Le programmeur peut développer n'importe quel type de programme Appelé, qu'il porte directement sur les données ou non, que la b.d. soit relationnelle ou non, que ce soient des traitements de gestion des données ou autres (calcul arithmétiques, validation de modèles statistiques, ...). La seule contrainte se situe donc au niveau du dialogue qui doit toujours se faire sous format SQL, à charge du programmeur d'écrire une routine de conversion des requêtes SQL en instructions compréhensibles pour l'Appelé (par exemple, exécuter des traitements sur une b.d. non relationnelle).

Quoiqu'il en soit, les environnements de middleware basés sur les SQL Interactions, ne seront jamais totalement neutres:

- soit ils limitent les programmes Appelés à des traitements sur une b.d., et un SGBD relationnel,
- soit ils sont indépendants des programmes développés, mais n'offrent plus une transparence totale de communication au programmeur, qui doit écrire une routine de conversion des requêtes SQL.

1.3.2.3 Access/kit de ARM-SIE

Présentation générale

Le produit Access/kit de ARM-SIE se range dans les techniques de middleware de **communication inter-processus** (située sur des machines distinctes).

Cet outil Access/kit est un peu particulier dans la mesure où il se limite au programme Appelant, et détourne les moyens de communication locale, de MS-Windows 3.X, à savoir les **DDE** (Dynamic Data Exchange), à des fins d'échanges distribués.

Fonctions de communication

Access/kit prend uniquement en charge les fonctions (cfr 1.3.1) de l'**Appelant** et d'**Interface**, et présuppose que le programme Appelé possède toutes les fonctions nécessaires à la communication.

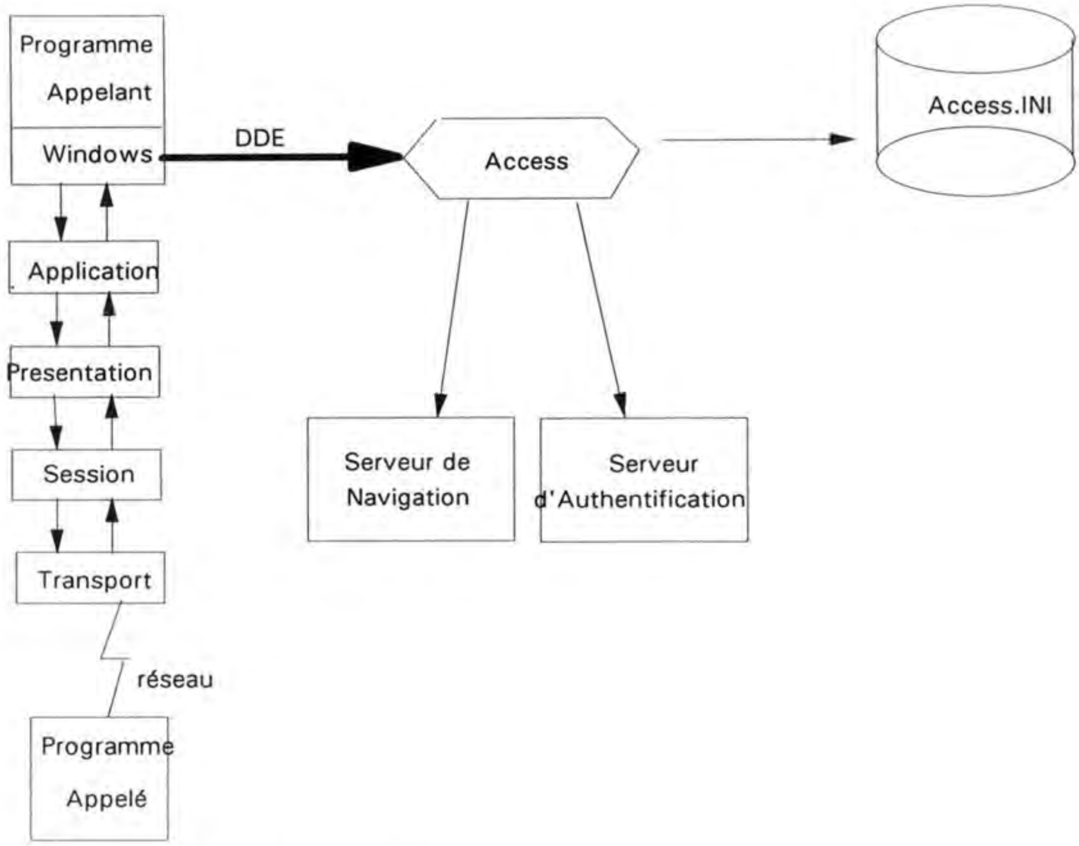


Figure 1.7: Technique d'Access/kit

Access/kit, logiciel se trouvant sur le même P.C. sous Windows que le programme Appelant, impose de générer une première requête DDE comprenant le prou de l'Appelé.
Après avoir intercepté la requête, Access/kit interroge des serveurs d'Authentification et de Navigation, se trouvant sur n'importe quelle machine du réseau distribué, pour localiser l'Appelé et vérifier les droits d'accès de l'Appelant.
En fonction des caractéristiques de l'Appelé, Access/kit consulte sa b.d. pour instancier, sur la machine Appelant, trois modules logiciels nommés Application Presentation, Session et Transport, qui correspondent parfaitement aux couches du modèle OSI.
Une fois le lien de communication établi, l'Appelant peut émettre n'importe quelle requête DDE ou n'importe quel programme Appelé.

Constatations

Si, normalement, les moyens de communication inter-processus doivent permettre de régler des communications entre programmes quelconques (tel que le mécanisme APPC de IBM), Access/kit est bien évidemment un cas particulier.

Access/kit ne vise pas une indépendance matérielle, mais veut ouvrir les applications P.C. sous Windows à l'environnement d'OSF/DCE.
Pour cela, Acces/kit travaille avec des serveurs d'Authentification et de Navigation conformes aux Security et Directory Services d'OSF/DCE.

1.3.3 Les API de middleware

A tout environnement middleware, quelle que soit la technique utilisée, est associé un ensemble d'API (Application Programming Interface), qui permet :

- aux programmes de l'A.D. d'utiliser les services des logiciels de middleware,

- et à la couche middleware de déterminer les fonctions de communication nécessaires.

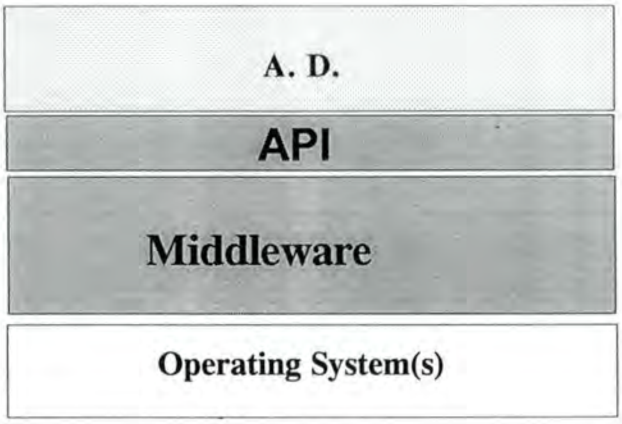


Figure 1.8: Eléments logiciels d'une A.D.

En fait, ces API sont des noms de routines et paramètres propres au produit de middleware utilisé, qui sont explicitement appelés par les différents programmes de l'A.D.
Ces appels précisent les étapes de communication entre programmes, permettant aux logiciels de middleware de déterminer les fonctions de communication (cfr 1.3.1.) à exécuter.

Par exemple, les API de SQLNetwork définissent essentiellement cinq étapes de communication, à savoir:

- une *étape de Connexion*, où le programme, qui veut émettre une requête, doit faire un appel explicite dans son code à *SQLConnect (nom_programme_appelé)*; le middleware peut en déduire:
    - qu'il doit localiser le programme dont le nom est passé en paramètre,
    - identifier la connexion,
    - etc ...

- une *étape de Compilation*, via l'API *SQLPrepare (nom_programme_appelé, instruction_SQL)*, par laquelle l'Appelant envoit sa requête SQL (*select, update, ...*) au programme passé en paramètre;

par API, le logiciel de middleware peut:
    - mettre la requête sous un format de présentation adéquat,
    - et la transmettre, sur base de l'identifiant défini, à l'étape suivante,
    - etc ...

- une *étape de Description*, via SQLDescribe (nom_programme_appelé), où l'Appelé peut transmettre les types des résultats de l'instruction demandée,

- une *étape d'Exécution*, par SQLExecute (nom_programme_appelé) l'Appelant demande l'exécution effective de la requête et lui revient le résultat,

- une *étape de Déconnexion (SQLDisconnect)*.

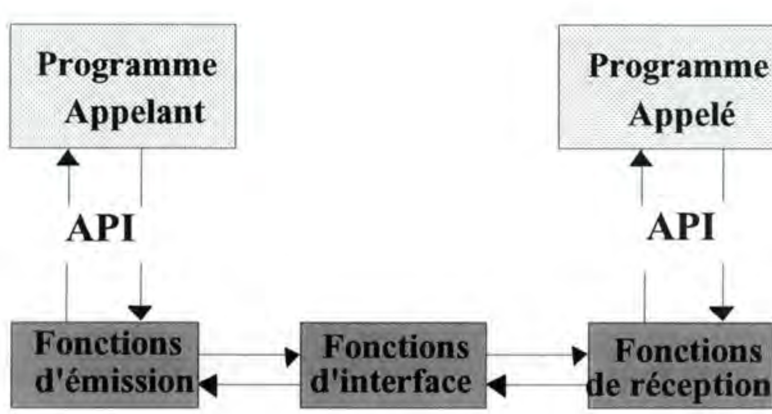


Figure 1.9: API de middleware

Remarque

Certains logiciels de middleware fournissent parfois ce que nous avons appelé des **Conversation Facilities**, dans la figure 1.1, qui sont des instructions permettant en un seul appel de générer l'exécution de plusieurs fonctions fournies par les API.

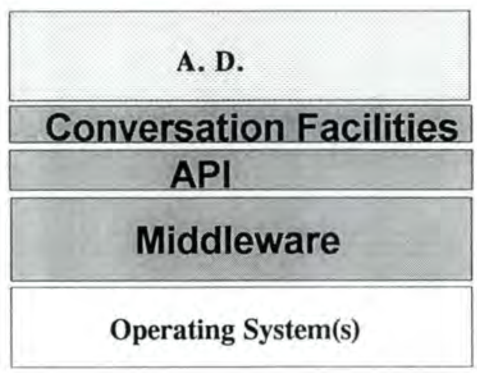


Figure 1.10 :Eléments logiciels d'une A.D.

Par exemple Gupta Technologies fournit, en plus de son produit SQLNetwork, un logiciel de développement d'applications sous Windows d'interrogation de b.d., appelé SQLWindows.

SQLWindows permet, par exemple, d'accéder à des bases de données distantes du reste de l'application via des instructions qui génèrent automatiquement, lors de leur appel, l'exécution de plusieurs API de middleware, tels que SQLCompile, SQLDescribe et SQLExecute en une seule instuction.

Chapitre 2

Chapitre 2: Architecture des applications distribuées

2.1 Introduction

Dans le premier chapitre, nous nous sommes uniquement intéressés à l'aspect technique des applications distribuées (A.D.), c'est-à-dire aux moyens matériels et logiciels de communication nécessaires à l'interaction des différents programmes d'une même application. Nous allons maintenant analyser les traitements implémentés par ces programmes et leurs relations.

Précision de la définition d'une A.D.

Une application distribuée sera vue comme un ensemble de **traitements** d'une même application **répartis sur différents** ordinateurs, appelés **agents**, qui peuvent s'échanger des informations pendant leur exécution.

Hypothèses

Nous restreindrons notre champ d'analyse aux **applications de gestion** c'est à dire, telles que définies par [Sacre91]:

> *"des systèmes [...] dans lesquels les fonctions de l'application manipulent le contenu d'une base de données"*.

Conformément à notre définition des A.D., nous étudierons uniquement la distribution des traitements sur différents agents:

- sans se préoccuper d'une répartition éventuelle des données, ce qu'on appelle b.d. distribuée,

- mais en envisageant la distribution de n'importe quelle catégorie de traitements d'une application de gestion, c'est-à-dire des opérations :
  - de dialogue avec l'utilisateur,
  - de gestion des données,
  - et de contrôle de l'application.

Objectif

Dans ce chapitre, nous chercherons à identifier différents **modèles de répartition des traitements**, que nous appellerons **architectures distribuées**.

Le choix de l'une ou l'autre de ces architectures devra permettre, au développeur, de répartir les opérations d'une même application sur les machines les plus appropriées pour les exécuter (nous détaillerons ces critères de distribution dans le troisième chapitre).

Démarche

Nous proposerons **d'abord une architecture générale des systèmes de gestion**, qui présentera et structurera les différentes catégories de traitements de toute application, qu'elle soit distribuée ou non.

Ensuite, de ce modèle général, nous déduirons **trois classes d'architectures distribuées**, correspondant à trois modes de répartition de ces catégories de traitements sur différents agents.

2.2 Proposition d'architecture des applications de gestion

En présentant une architecture générale, nous cherchons à définir un niveau logique permettant d'isoler les différentes catégories de traitements de toute application de gestion.

Cette architecture, proche de celles souvent proposées dans la littérature traitant du "distributed processing" ([White90], [Berso92], [Sybas93]), offre les avantages suivants:

- son **niveau d'abstraction**: cette architecture s'applique aussi bien à des applications centralisées que distribuées; nous dirons que la solution peut s'exécuter sur une machine "abstraite", c'est-à-dire "comme si" l'ensemble de l'application se trouvait sur un seul et même agent,

- ses principes de définition, à savoir la **modularité** et l'**indépendance** de ses composants, reconnus comme principes de base des méthodologies de développement de logiciels,

- et surtout sa facilité d'**adaptation à des systèmes composites (ou modèles multi-agents)**, qui permettra de définir différentes classes d'architectures distribuées.

2.2.1 Principes de définition

Notre architecture se base sur deux règles traditionnelles du génie logiciel: la modularité et, son corollaire, l'indépendance.

Principe de modularité

L'application est vue comme un assemblage de modules de traitements, proposant différents services. Notre découpe modulaire s'appuie sur deux critères [Parna72]:

- **critère de cohésion logique et fonctionnelle**: par module, on regroupe les traitements de l'application offrant un certain nombre de services de même type,

- **critère de couplage faible**: le nombre de relations entre modules sera minimal, permettant le développement le plus indépendant possible.

Principe d'indépendance

L'indépendance entre modules consiste à pouvoir développer ou modifier un module de manière isolée, sans en affecter le reste de l'application.

Notons que le terme "indépendance" est toujours utilisé abusivement, quelle que soit la découpe modulaire adoptée. Une véritable indépendance des différents modules de traitements d'une application est bien évidemment impossible; et le terme "autonomie" semble plus adapté.

Cette autonomie offre essentiellement comme avantages:

- **une spécialisation en unités de travail** [Duboi91]:
"les modules mis en évidence doivent être des unités de travail pour les analystes (et faciliter les tâches de validation, maintenance et documentation)",

- **une spécialisation en unités d'exécution:**
dans notre optique de développement d'A.D., une découpe en modules autonomes s'adapte facilement à des modèles multi-agents.

Contraintes

Comme l'explique [Sacre91], *"le prix à payer pour assurer l'indépendance"* entre modules est la nécessité d'un **dialogue interne** entre les composants de l'application.
Ce dialogue, transparent aux utilisateurs de l'application, est chargé des échanges d'information entre modules.

A ce dialogue interne, il faut encore rajouter le problème de la **localisation du contrôle**, c'est à dire décider de la partie de l'application (module ou ensemble de modules) qui sera responsable de l'enchaînement des traitements.

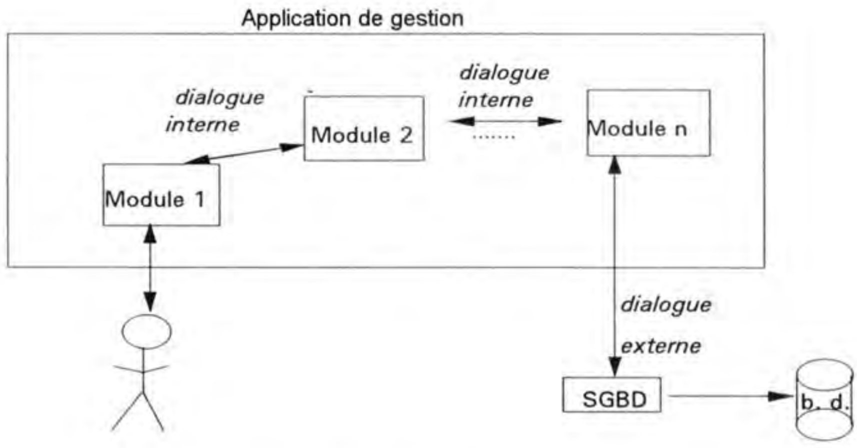


Figure 2.1: Dialogue interne

2.2.2 Description de l'architecture

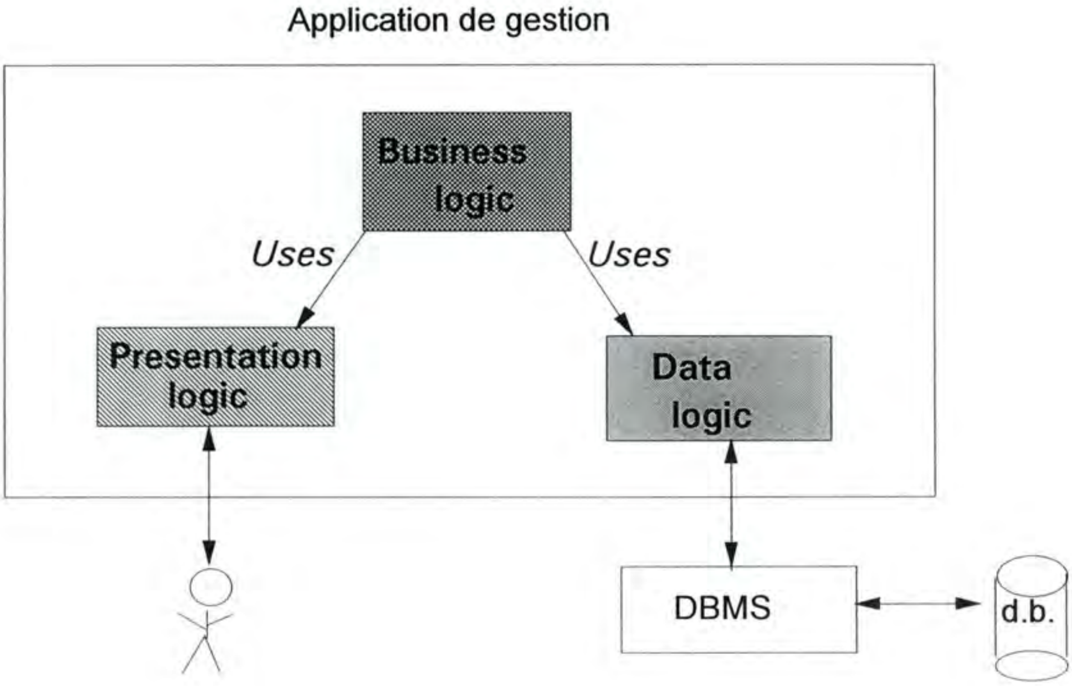


Figure 2.2: Proposition d'architecture

L'architecture proposée regroupe les traitements de toute application de gestion en trois catégories que nous appellerons, en référence à une terminologie souvent utilisée dans la littérature distribuée ([White90], [Berso92], [Sybas93]), modules de:

- Presentation Logic,

- Business Logic,

- Data Logic.

Nous caractériserons chacun de ces modules par les services qu'ils rendent, les traitements nécessaires à la réalisation de ces services, et leurs relations avec les autres modules de l'application.

2.2.2.1 Les modules

Presentation logic

• **Service**: gestion des entrées/sorties, du **dialogue avec l'utilisateur**.

Pour reprendre une distinction habituelle en matière d'interface homme-machine, [Hutch86], nous dissocions deux familles de styles de dialogue, à savoir les interfaces construites:

- sur la métaphore de la conversation,
- sur la métaphore du mini-monde.

Par la métaphore de la conversation on considère que l'utilisateur, via un langage, mène une conversation avec l'application au sujet d'un monde supposé mais non explicitement représenté dans l'interface.
Dans ce cas nous parlerons de **dialogue "conversationnel"** où l'utilisateur *définit,* par le clavier de son poste de travail, les actions que le système va exécuter à l'aide d'un langage de commandes et autres structures linguistiques (menus, formulaires, ...).

Par la métaphore du mini-monde, l'interface ne constitue plus un intermédiaire de conversation, mais un monde dans lequel l'utilisateur peut évoluer. L'utilisateur agit directement en manipulant, via la souris de son poste, des représentations visuelles des éléments de son application.
Ce dialogue, souvent qualifié de **manipulation directe**, peut se réaliser à l'aide de Graphic User Interfaces (GUI) standards tels que Microsoft's Windows, X Windows, OSF/Motif, etc ...

• **Traitements**:

- actions physiques nécessaires à la communication avec l'utilisateur; c'est à dire, comme le définit [Berso92], *"Presentation logic performs such tasks as screen formatting, reading, and writing of the screen information, windows management, keyboard, and mouse handling"*,

- la gestion des éléments de l'interface; c'est à dire la création, modification ou suppression d'objets de dialogue ou de structures linguistiques selon le style de dialogue,

- enregistrement d'événements extérieurs, plus précisément la réception de messages d'un GUI lors de manipulations directes, ou l'enregistrement d'instructions dans le cas d'un dialogue conversationnel,

- visualisation de l'état d'avancement de l'application, et de ses éventuels problèmes techniques,

- contrôle syntaxique (telle que la vérification du type et format) des données introduites par l'utilisateur.

- vérification de la cohérence entre l'affichage des données et leur enregistrement dans la b.d.

Par exemple pour une application de gestion des commandes des clients d'une entreprise, après que le concepteur ait choisi un style d'interface, les modules de Presentation Logic sont chargés:
- de la gestion des écrans de dialogue,
- de saisir et contrôler la syntaxe des commandes encodées,
- d'afficher la progression (ou le nombre) des lignes de commande enregistrées dans la b.d.,
- etc ...

Data logic

• **Service** : manipulation des données de l'application.

• **Traitements**:

- les **fonctions** de l'application c'est-à-dire, au sens de [Bodar89], les traitements élémentaires et non interactifs qui manipulent les informations mémorisées dans la b.d.,

- les **actions primitives** (de consultation, ajout, modification, et suppression) sur la b.d., utilisées par les fonctions de l'application.

Autrement dit, les fonctions sont le lieu de définition de l'appel aux actions primitives sur la b.d..

Pour reprendre l'exemple de gestion des commandes de clients, les modules de Data Logic de cette application devront entre autres se charger:
- d'une fonction *Enregistrement d'une commande* pour un client et des produits donnés; cette fonction pourra faire appel à une action primitive de *consultation* de l'identifiant d'un client, pour vérifier que ce dernier existe, et également à des actions primitives d'*ajout* d'éléments au fichier reprenant toutes les commandes de l'entreprise,

- de même pour une fonction *Modification d'une commande*,

- etc ...

Business logic

• **Service**: gérer le **contrôle de l'application**, c'est-à-dire décider de l'enchaînement des traitements de Presentation et Data Logic afin d'atteindre l'objectif assigné de l'application

• **Traitements**:

Indépendamment du type de contrôle à exercer, que nous aborderons par la suite, le comportement de la Business Logic (B.L.) peut être comparé à une table de décision.
Par cette table la B.L. peut, en fonction de l'état courant et d'événements provenant des autres composants de l'application, décider le déclenchement d'opérations de Presentation Logic (P.L.) ou de Data Logic (D.L.).

Pour reprendre le cas de gestion de commandes, la B.L. peut recevoir un événement de la P.L. signalant que l'utilisateur veut consulter l'ensemble des commandes d'un client donné. En fonction par exemple des droits d'accès de cet utilisateur, et de l'état courant de la b.d., la B.L. peut décider:

- de déclencher des traitements de P.L. pour signaler à l'utilisateur qu'il ne peut consulter ces informations,
- ou activer des fonctions de Data Logic de consultation de commandes et de ses produits associés,
- etc ...

	Etat de l'application	Etat *fichiers courants =x,y,z*	...	Etat	Etat
Evénement					
Evénement *=demande de consultation*		[redacted]			
....					
Evénement					
Evénement					

Figure 2.3: Table de décision de la B.L.

Par analogie avec les tables de décisions, les traitements de la Business Logic sont donc:

- l'enregistrement d'événements provenant des autres modules de l'application,

- la mémorisation de l'état de l'application, c'est-à-dire le résultat des traitements de P.L. et D.L. (par exemple les fichiers ouverts de l'application, les données introduites par l'utilisateur, la position des éléments de l'interface, ...),

- le déclenchement ("**triggering**") d'un traitement de P.L. ou D.L., cette opération générant un changement d'état dans la table de décision.

2.2.2.2 Le dialogue interne

La découpe modulaire proposée, pour tout système de gestion, répond à un double principe d'indépendance (ou plutôt d'"autonomie"), admis dans la plupart des méthodologies de développement, à savoir que les **fonctions de l'application** peuvent être développées et gérées **indépendamment:**

- du design et de la gestion **de la b.d.**,

- de la conception et de l'exécution **de l'interface homme-machine**.

Cela met en évidence la nécessité d'un dialogue interne précis entre modules de l'application.

A la figure 2.2 et 2.4, nous nous sommes contentés de représenter les deux relations de base entre modules de l'application, schématisant uniquement la hiérarchie de ces composants, à savoir que:

- les modules du niveau le plus élevé, c'est-à-dire de B.L., ne peuvent être utilisés par aucun autre module de niveau inférieur,
- les services des modules de niveau inférieur, c'est-à-dire P.L. et D.L., sont nécessairement utilisés par, au minimum, un module de niveau supérieur.

Nous avons représenté cette hiérarchie à l'aide de relations USES de [Parna72], qui suppose les échanges d'information entre modules comme étant instantanés.
Cette relation fait abstraction de toute notion technique d'exécution, elle ne considère pas que les modules de l'application vont être exécutés par des processeurs physiques (avec un certain temps d'exécution, une éventuelle concurrence d'utilisation simultanée des services d'un module, ...).

La relation USES de [Parna72] restreint donc l'exécution de l'application à une machine abstraite, dont les temps de traitement et de réponse à une requête sont considérés comme instantanés. Cela décharge (dans un premier temps) le concepteur de l'application de tout souci technique d'implémentation, et lui permet de se concentrer uniquement sur une solution réalisable par une machine abstraite. Nous dirons que le développeur a une vision statique de l'application.

Dans notre architecture cela signifie que la B.L., en tant que "contrôleur" de l'application, "utilise" les services des modules de P.L. et D.L.; mais cela ne détermine pas encore toutes les relations entre les composants.

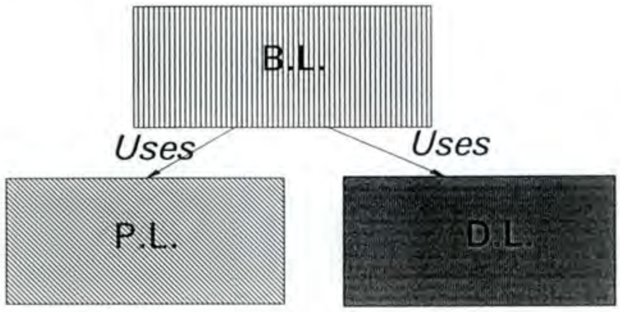


Figure 2.4: Relations de base

Les traitements de P.L. et D.L. étant au même niveau, les modules de P.L. peuvent utiliser des services de D.L., et inversement.
Par exemple, lors d'un contrôle syntaxique la P.L. pourrait directement s'adresser à la D.L. pour vérifier la cohérence entre une donnée enregistrée dans la b.d. et son affichage à l'écran.
De même, lors d'exécutions d'actions primitives de D.L. (telle que l'enregistrement des lignes de commandes de clients), un module de D.L. pourrait signaler, à un module d'affichage à l'utilisateur, la progression (le nombre) des enregistrements effectués.

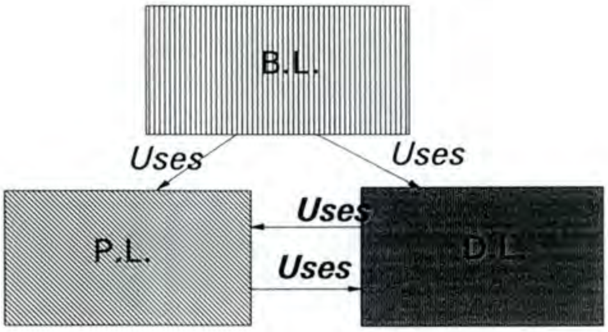


Figure 2.5: Relations entre P.L. et D.L.

Pour être tout à fait complet, précisons encore que chacun de nos composants (B.L., P.L. et D.L.) ne sont que des catégories de modules; ces catégories pouvant elles-mêmes être raffinées en hiérarchies.

Nous pouvons ainsi obtenir des architectures telles que représentées à la figure 2.6.

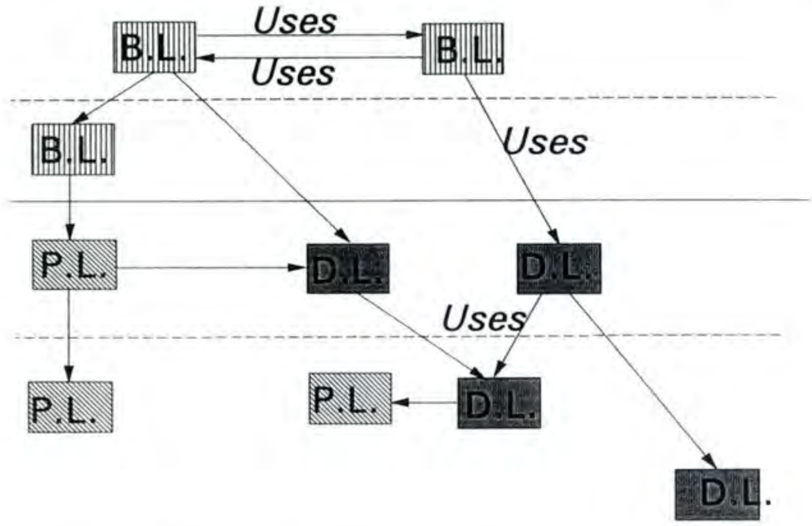


Figure 2.6: Exemple de hiérarchie complète

Les seules relations que nous excluons sont donc celles où un module de niveau inférieur utiliserait les services de modules supérieurs. Il est donc **impossible pour les traitements de P.L. et D.L. d'utiliser les services de B.L.**

Une synthèse complète des relations entre modules de notre architecture pourrait se représenter tel qu'à la figure 2.7.

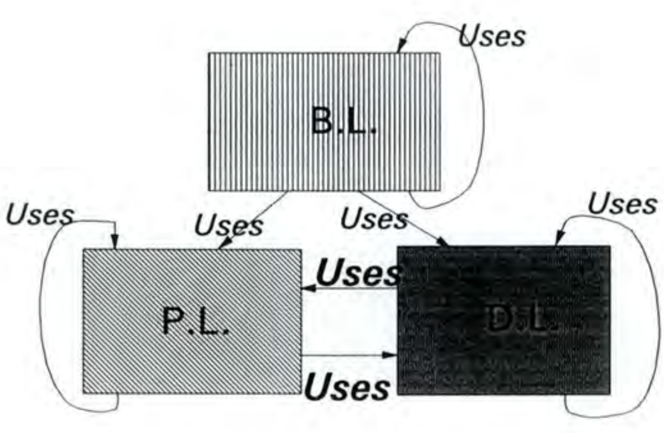


Figure 2.7: Représentation complète des relations

Figure 2.7: Représentation complète des relations

Pour des raisons de simplicité dans nos schémas, nous limiterons nos graphiques aux relations de base de la hiérarchie (cfr figure 2.4), c'est-à-dire entre les modules de B.L. et le reste de l'application.

Bien qu'elles soient toujours présentes, nous ne représenterons les autres relations uniquement lorsqu'elles devront faire l'objet d'une étude particulière par le concepteur de l'application (par exemple, quand ces relations seront soumises à une distribution de ses modules, cfr 2.3.2).

2.2.2.3. Contrôle de l'application

Le contrôle de toute application de gestion, pris en charge par la B.L., se caractérise par deux paramètres:

- son degré d'automatisation,
- le séquencement des opérations de l'application.

- **Degré d'automatisation**

Le contrôle d'un S.I. peut être entièrement automatisé par la B.L. si l'ordonnancement des traitements de P.L. et D.L. est parfaitement prévisible et formalisable.

Ce type de contrôle se retrouve essentiellement dans les applications de gestion opérationnelle, que nous détaillerons au troisième chapitre (cfr 3.2.1.), telles que des systèmes de calcul de salaires, de suivi de production,... .

Au contraire des S.I. stratégiques et basés sur des processus de décision semi-structurés (en référence à la classification des S.I. de Keen & Scott Morton, que nous préciserons au troisième chapitre), où le séquencement des opérations de P.L. et D.L. ne peut être totalement pré défini et pris en charge par la B.L.

- **Séquencement des opérations**

Par extension de principes définis en matière d'interface homme-macine, [Hart89], et indépendamment du degré d'automatisation du contrôle, nous parlerons de:

- **contrôle séquentiel**, ou synchrone, s'il existe *une seule séquence possible* des traitements (P.L. et D.L.) de l'application,

- **contrôle multi-fils** ("*multi-threads*"), ou asynchrone, si *plusieurs séquences* sont possibles, mais *une seule s'exécute à la fois* et peut être interrompue pour démarrer une nouvelle séquence, et y revenir par la suite.

  En matière d'interfaces hommes-machines, [Sacre91] explique que lors d'un dialogue multi-fils, "*l'utilisateur a plusieurs chemins disponibles vers des tâches différentes, c'est-à-dire qu'il a le choix entre plusieurs fils d'activités distincts mais une seule tâche est en cours d'exécution à un instant donné*".

- **contrôle concurrent**, il s'agit d'un contrôle multi-fils, où plusieurs schémas d'activités peuvent s'exécuter *simultanément*.

Précisons que ces deux caractéristiques, degré d'automatisation et séquencement, sont totalement indépendantes, et leur combinaison permet de définir six types différents de contrôle.

Ainsi, par exemple, on peut très bien imaginer une application, telle qu'un système expert, à plusieurs fils d'activités menés parallèlement et entièrement gérés par une B.L. suffisamment "intelligente".

2.2.3 Comparaison avec d'autres architectures

Architectures hiérarchiques

Comme leur nom l'indique, ces architectures (par exemple [Coutt86] et [Duboi91]) se caractérisent par une structure hiérarchique (cfr 2.2.2.2.) des modules de l'application.
Dans le cas de l'architecture de [Duboi91], les applications sont construites suivant une hiérarchie à cinq niveaux:

- niveau 5: Coordination: modules issus directement des spécifications fonctionnelles,
- niveau 4: Interface homme-machine, (ihm),
- niveau 3: modules de Data logic,
- niveau 2: SGBD, graphic user interface, ...
- niveau 1: système d'exploitation.

La petite différence avec notre architecture porte sur les positions hiérarchiques des modules de Presentation et Data Logic.

Pour [Duboi91], les traitements d'ihm sont d'un niveau supérieur aux composants de Data Logic. Ils peuvent donc appeler les "fonctions de l'application", mais celles-ci ne peuvent directement utiliser les services d'interface.
Tandis que dans notre architecture, les composants P.L. et D.L. sont considérés comme étant de même niveau (cfr 2.2.2.2.). Les échanges d'information directement entre eux, sans passer par la Business Logic, sont possibles dans les deux sens (de la P.L. vers la D.L., et inversement).

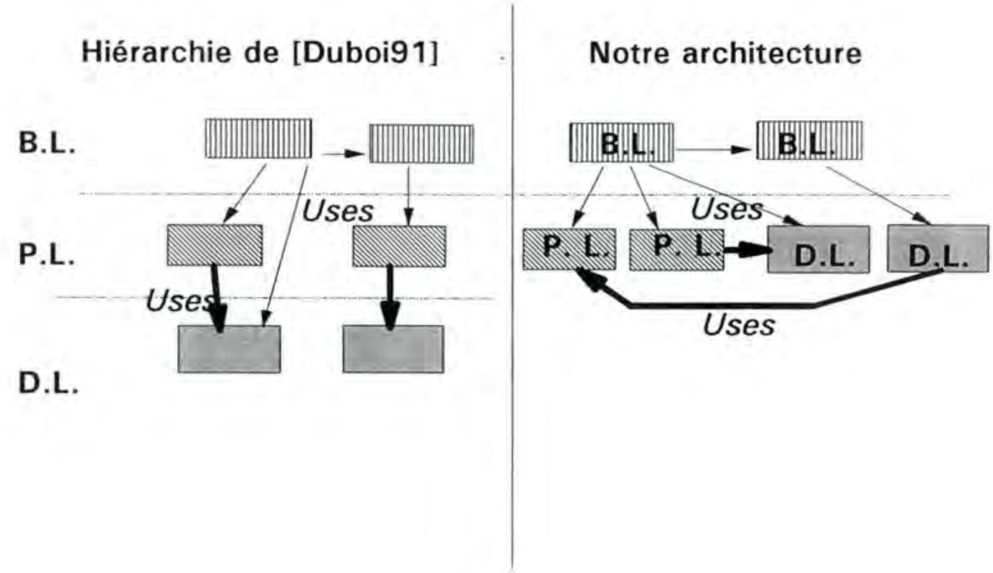


Figure 2.9: Comparaison hiérarcie [Dubois91]

Architecture du projet TRIDENT

Description

L'architecture, que nous allons brièvement présenter, a été définie dans le cadre du projet TRIDENT (Tools foR an Interactive Development EnvironmeNT, [Bodar93], [Sacre91]), et sert essentiellement à faciliter le développement d'interfaces homme-machine pour des applications de gestion hautement interactives.

Cette architecture détaille principalement les traitements de communication avec l'utilisateur, et subdivise toute application en trois composantes:

- l' "**application**" proprement dite, regroupant l'ensemble des traitements de consultation et de modification du contenu de la b.d.,

- la "**présentation**" , gérant les actions physiques nécessaires à la communication avec l'utilisateur, et la vérification des types et formats de données échangées,

- le "**dialogue**", responsable du séquencement des opérations d'entrées/sorties, du déclenchement des traitements du composant application, et de la gestion de la cohérence entre la b.d. et les données affichées à l'écran.

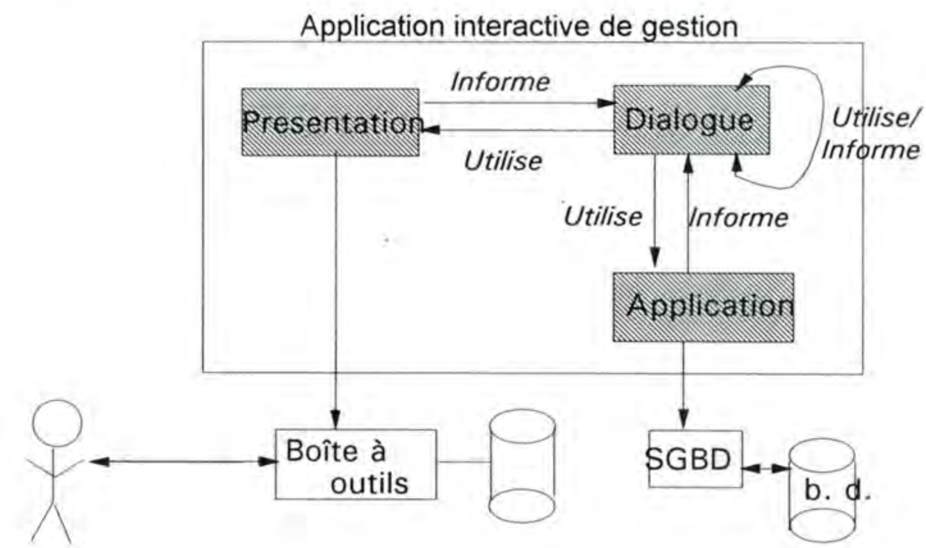


Figure 2.10: Architecture TRIDENT

Précisons encore que, pour faciliter la construction d'interfaces modernes et de qualité, l'architecture TRIDENT adopte une philosophie orientée objet et présuppose l'utilisation de boîtes à outils lors de la conception de l'ihm.

Les concepteurs de ce modèle se basent sur l'idée que ([Bodar93], [Sacre91]) "*l'utilisation des langages orientés objets (C++, Blaise, ...) implémentant sous forme de classes les fonctionnalités présentes dans les boîtes à outils, facilite la réalisation des interfaces [...]. L'utilisation de ces classes d'objets "prêtes à l'emploi" masque les détails d'implémentation des boîtes à outils et constitue un niveau d'abstraction plus proche de l'attente des utilisateurs.*".

En fait, à chaque composant de l'architecture TRIDENT est associé une classe d'objets:
- le composant Application est couplé à des objets indépendants proches des types d'entités du modèle conceptuel des données,
- au composant Presentation est associé des objets interactifs indépendants, tels qu'une fenêtre, un bouton, une icône, ...
- le composant Dialogue comprend une hiérarchie d'objets; chacun de ces objets de dialogue utilisant les services des autres objets afin d'atteindre un but précis, sous-objectif de l'ensemble de l'application.

Le principal avantage de cette architecture réside au niveau des objets de Dialogue, considérés comme abstraits (non visibles à l'utilisateur), qui non seulement gèrent et utilisent les objets de Presentation et d'Application, mais gèrent aussi récursivement d'autres objets de dialogue.
Et, comme conclut [Bodar93], ce niveau d'abstraction devrait permettre de développer une méthodologie de conception d'interfaces homme-machine assistée par ordinateur.

Comparaison

Le composant Presentation du modèle TRIDENT représente une partie du module de Presentation Logic de notre architecture, à savoir la partie visible de l'application et ses tâches techniques de communication.
Quant au module de dialogue de TRIDENT, il correspond à la fois aux autres traitements de notre composant de Presentation Logic (à savoir la gestion d'un style de dialogue, et de la cohérence avec la b.d.) et au module de Business Logic, chargé du contrôle du système de gestion.
La principale différence se situe donc au niveau de la localisation du contrôle de l'application, que le modèle TRIDENT confie à son composant Dialogue.

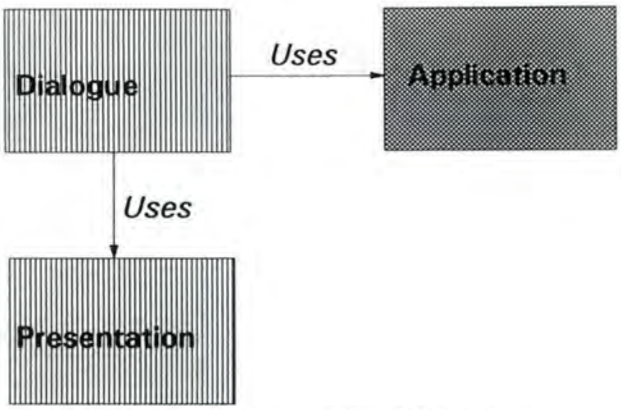


Figure 2.11: Contrôle du modèle TRIDENT

En fait, uniquement sur base de ce critère de localisation du contrôle, [Hart89], on pourrait recenser quatre types d'architecture de systèmes de gestion:

• Soit on intègre, comme dans le modèle TRIDENT, les traitements de Business et Presentation Logic; [Hart89] parle de ***contrôle externe*** de l'application.

Cela rend les traitements sur les données totalement dépendants du dialogue, mais peut présenter des avantages tels qu'un "*prototypage rapide de l'application*" [Sacre91], ...

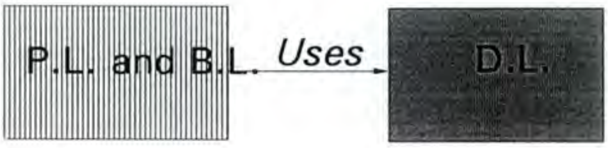


Figure 2.12: Contrôle externe

• Soit on associe le contrôle à la Data Logic (***contrôle interne*** selon [Hart89]); l'interface homme-machine est dès lors totalement subordonnée à l'organisation et la gestion des données.

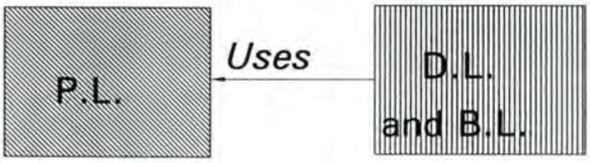


Figure 2.13: Contrôle interne

• Soit on répartit les tâches de pilote de l'application à la fois sur le dialogue et la Data Logic ([Hart89] parle de ***contrôle mixte***).

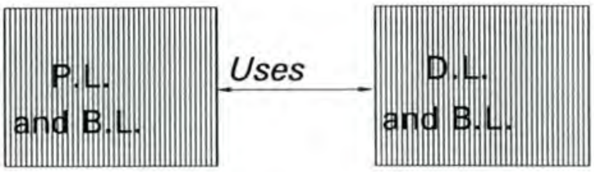


Figure 2.14: Contrôle mixte

• Soit, finalement, comme pour notre proposition d'architecture, on dissocie parfaitement les opérations de contrôle des autres traitements de l'application, ce qui permet la plus grande autonomie possible entre les modules de P.L. et D.L. (c'est à dire un **contrôle global** pour [Hart89])

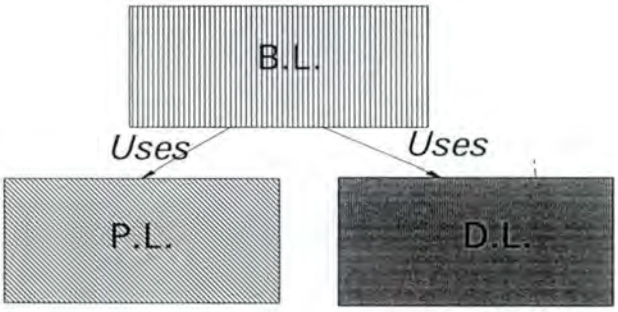


Figure2.15: Contrôle global

2.3. Classes d'architectures distribuées

2.3.1. Systèmes composites et architectures distribuées

Dans notre proposition d'architecture de toute application de gestion, nous n'avons tenu compte d'aucun concept de distribution. Nous allons maintenant descendre d'un niveau d'abstraction, en prenant en considération la répartition éventuelle des composants de l'architecture sur différents agents, et avoir une vision dynamique de leur interaction .

Système composite

Parmi le vocabulaire abondamment utilisé dans la littérature traitant des problèmes de distribution, nous adopterons la terminologie suivante:

- le terme "**système composite**", ou "modèle multi-agents", désignant selon [Duboi93]

  " ***a system made up of a combination of (often heterogeneous) components, called agents, acting in parallel and cooperating to achieve one or several goals assigned to the system considered as a whole.***
  The term multiple computing system is not used because we do not assume that all agents are computer systems.",

- et "**architecture distribuée**" désignant un **système composite dont tous les agents sont des ordinateurs**.

Le concept de modèle multi-agents est donc beaucoup plus général que notre notion d'architecture distribuée. Cela nous permettra d'essayer de transposer, aux applications distribuées de gestion, les recherches menées pour d'autres types de systèmes composites, tels que:

- les Computer Integrated Manufacturing (CIM) systèmes, où les agents sont des appareils de manutention [Duboi93],

- les systèmes parallèles (ou concurrentiels), où les agents sont des processeurs au sein d'un même ordinateur.

- les Systèmes d'Information d'Aide à la Décision (SIAD) distribués, où les agents sont des êtres humains et des automates.

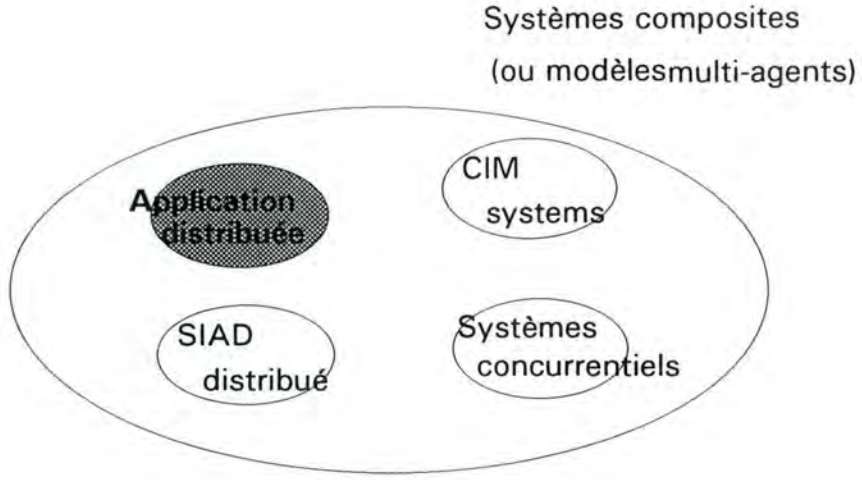


Figure 2.12: Typologie des systèmes composites

Architecture distribuée

Selon la définition d'un système composite, les agents d'une application distribuée de gestion prennent en charge les différents traitements de Presentation, Data, et Business Logic, et interagissent afin d'atteindre l'objectif assigné à l'ensemble du système.

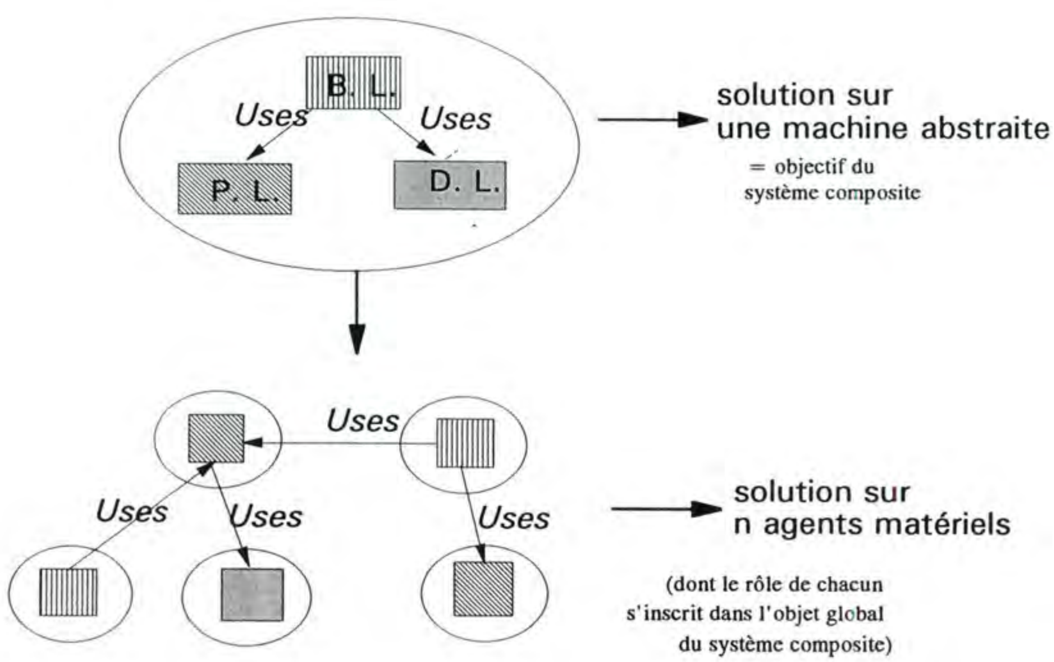


Figure2.13: Architecture distribuée

Avant même d'envisager une répartition par traitement de l'application de gestion, pouvant mener à des architectures distribuées complexes, nous raisonnerons d'abord **par distribution de catégories** de traitements, à savoir les trois composants (P.L., B.L., et D.L.) de notre modèle général des systèmes de gestion.

Nous définirons ainsi trois classes d'architectures distribuées, correspondant à la séparation d'un seul composant du reste de l'application; nous parlerons de :

- **client/server processing,**
- **cooperative processing,**
- **remote presentation processing.**

Conséquences de la distribution d'une application de gestion

Par la distribution de l'un ou l'autre composant de l'architecture, le concepteur doit prendre en considération deux nouvelles notions:

- une **vision dynamique du dialogue** interne à l'application,
- un nouveau et quatrième composant de l'architecture: **un module de communication**.

En voulant échanger des informations entre agents matériels distincts, le concepteur d'un système distribué ne peut plus considérer les flux d'échanges entre modules comme se réalisant instantanément. Il doit abandonner sa vision statique de l'application et la relation USES de [Parna72], pour tenir compte des processeurs physiques exécutant les différents modules. Le dialogue ne peut se faire instantanément, mais doit être envisagé en deux étapes:

- tout d'abord l'envoi ("***call***"), par un module, d'un événement déclenchant ("*trigger*") le traitement d'un autre module,
- ensuite, après un certain temps d'exécution, la réception du résultat ("***inform***").

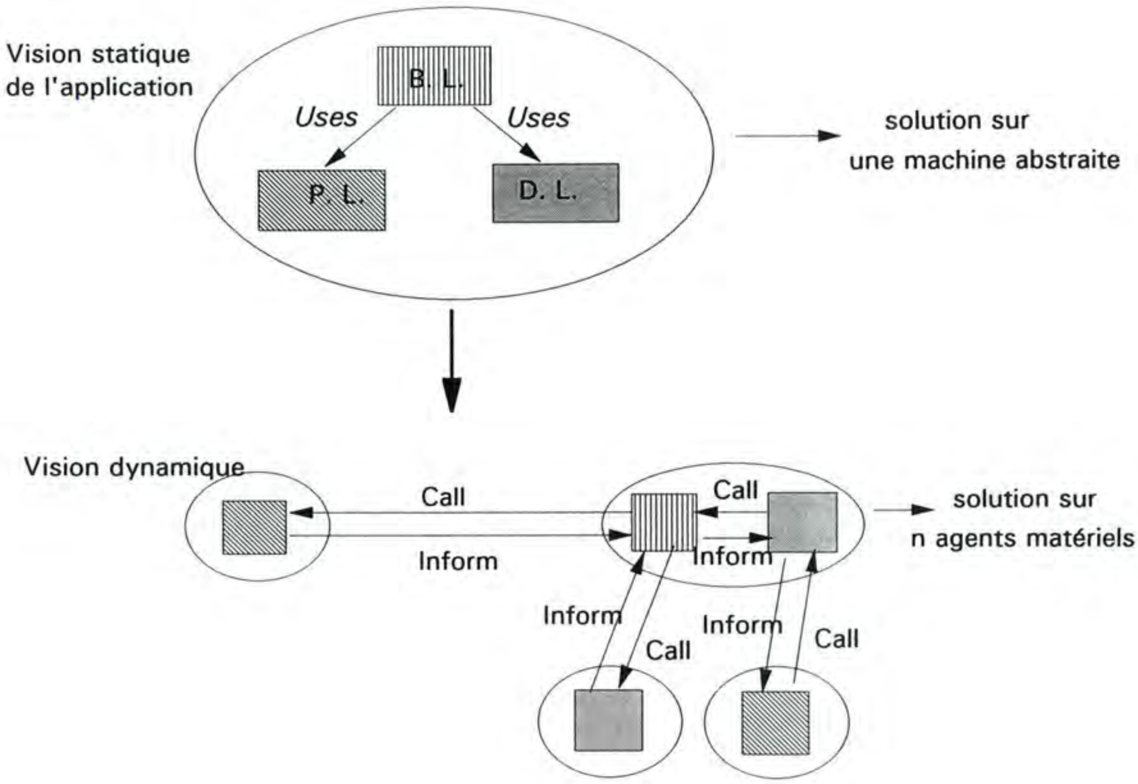


Figure 2.14: Vision dynamique du dialogue

Cette vision dynamique du dialogue interne implique également la définition d'une nouvelle catégorie de traitements, regroupés dans un module de communication.
Il s'agit de traitements nécessaires à toute communication entre agents matériels, et devant être supportés au niveau de l'infrastructure du système par des logiciels de middleware (cfr. 1.3.).

Ce composant de communication assure donc le lien entre l'architecture et l'infrastructure des applications distribuées, et est implémenté par une ou plusieurs fonctions de middleware (fonctions *de l'Appelé, d'Interface et de l'Appelant*) définies au premier chapitre (cfr. 1.3.1.).

Précisons que nous dissocions:

- les ***traitements de communication***, qui font partie des modules de communication de l'architecture, et sont explicitement utilisés par le concepteur de l'application;

- et les ***fonctions de middleware*** (à savoir fonctions de l'*Appelé*, d'*Interface* et de l'*Appelant*; cfr 1.3.1.) qui, pris en charge par tout environnement de middleware, permettent d'**implémenter**, de façon transparente, **les traitements de communication**.

Le concepteur de l'application distribuée gère tout échange d'information entre agents par des appels à ces traitements de communication, via les API (cfr 1.3.3.) de middleware.

Pour chaque appel à un traitement de communication, l'environnement de middleware exécute une ou plusieurs de ses fonctions, de manière totalement transparente pour le programmeur et l'utilisateur final.

Nous détaillerons ces modules de communication pour chaque architecture distribuée. Nous préciserons les fonctions de middleware qui les implémentent, et l'environnement de middleware qui semble le plus approprié, en fonction des catégories de traitements qui sont distribuées.

Exemple

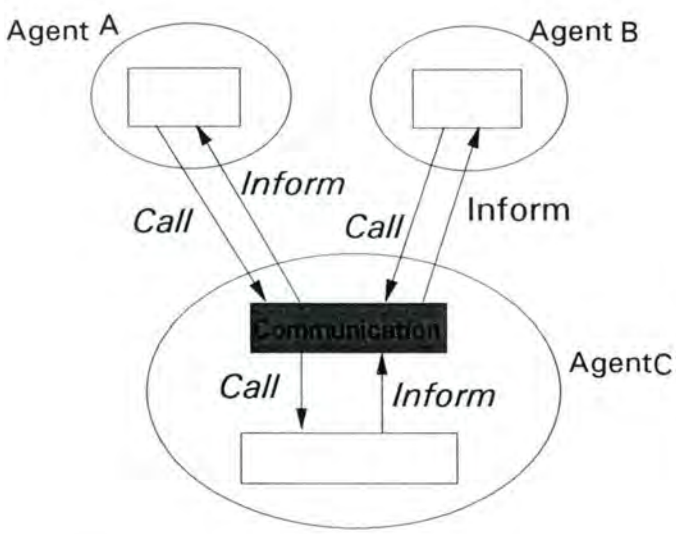


Figure 2.15: Module de communication

Remarque

Par souci de clarté dans nos schémas, nous continuerons à représenter le dialogue interne par **une simple flèche** (de l'appelant vers l'appelé) bien que, comme nous l'avons expliqué, à ce stade de la réflexion le dialogue est vu dynamiquement et se fait en deux étapes non instantanées.
Rappelons également que nous ne dessinerons que les relations **entre B.L. et le reste de l'application.** Les autres relations (cfr 2.2.2.2.) existent toujours, mais nous les représenterons uniquement lorsqu'elles relient deux modules distribués sur des agents distincts, et devront donc faire l'objet d'une analyse particulière du concepteur (appel aux API de middleware).

2.3.2. Trois classes d'architectures distribuées

Bien souvent dans la littérature, il est communément admis que toute application distribuée se base sur un modèle à deux agents appelés client et serveur tels que, [OSF91],

> " *The client side of the application is the part that [...] initiates the distributed request and receives the benefit of the service [...].*
> *The server side of the application is the part that [...] receives and executes the distributed request* " .

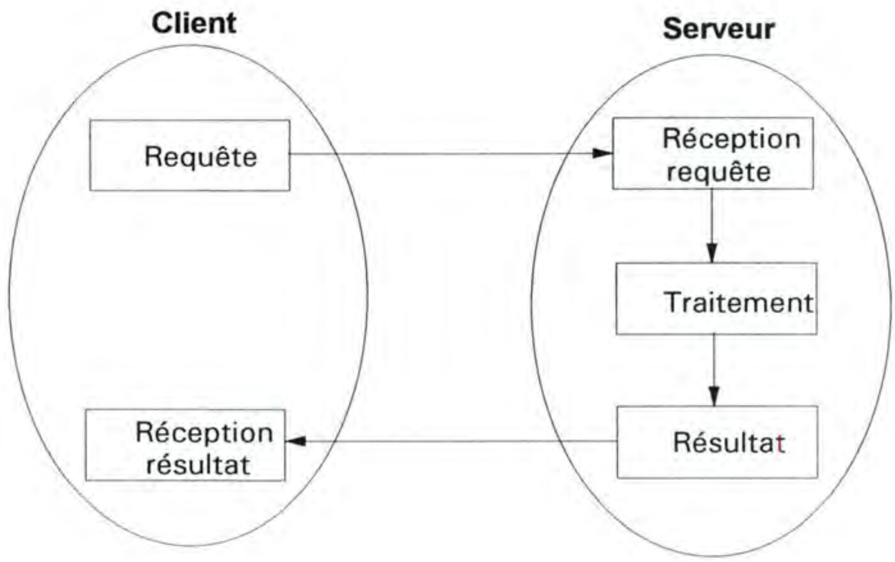


Figure 2.16: Paradigme c/s

En fait, ce modèle ne déterminant pas les types de traitements distribués, mais schématisant seulement les interactions entre les deux agents, nous l'assimilons simplement au protocole de dialogue interne à toute application distribuée, indépendamment de son architecture.

Ce flux "requête-traitement-résultat", souvent appelé **paradigme client/serveur (c/s)**, est donc pris en charge par l'infrastructure des applications distribuées (cfr 1.3.1.), et est tout à fait valable pour les trois classes d'architectures distribuées que nous définirons, à savoir:

- client/server processing,
- cooperative processing,
- remote presentation processing.

Précisons également que les rôles définis par ce paradigme C/S ne sont pas a priori définitivement liés à une machine ou un programme.
Tout agent, ou modèle de traitements, peut être client (ou Appelant, pour reprendre la terminologie du premier chapitre, cfr 1.3.1.) dans une relation, et serveur (ou Appelé) dans une autre.

2.3.2.1 Client/server processing

Dans le même ordre d'idée que [White90] et [Berso92], nous qualifions de *client/server processing* une architecture se caractérisant par la **séparation des** (ou une partie des) **traitements de Data Logic du reste de l'application**.

Répartition des traitements

Cette architecture comprend donc deux types d'agents:
- le **client**, qui exécute les traitements de Presentation, Business Logic et, éventuellement, d'une partie de la Data Logic,
- le **serveur,** qui implémente une partie des modules de Data Logic.

Cette répartition d'exécution des traitements de Data Logic peut prendre deux formes différentes, que nous appellerons architecture client/serveur (c/s):
- orientée donnée,
- orientée fonction.

Architecture c/s orientée donnée

Par cette solution, toutes les fonctions de D.L. (au sens de [Bodar89]) se trouvent sur l'agent client, et les actions primitives (de consultation, ajout, modification et suppression) sur la b.d. s'exécutent sur l'agent serveur.

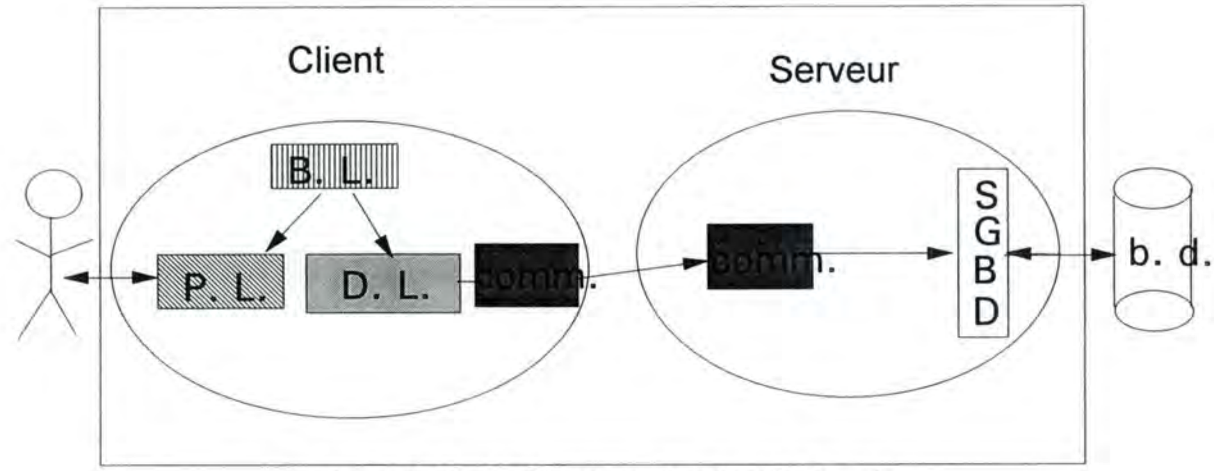


Figure 2.17: Architecture c/s orientée donnée

Le dialogue se fait donc via les fonctions de D.L. qui, en traduisant sur l'agent client les demandes d'information de l'application en actions primitives sur la b.d., soumettent des requêtes à l'agent serveur.

Précisons que la relation "distribuée", c'est-à-dire pour laquelle le concepteur devra faire appel au module de communication, se situe au niveau de la D.L. Ce sont les fonctions de D.L. qui doivent utiliser les API de middleware, pour demander l'exécution d'actions élémentaires sur la b.d. d'un autre agent. **Tout** le dialogue distribué se fait via les fonctions de l'application.

Pour reprendre l'exemple de l'application de gestion de commandes de clients, une architecture c/s pourrait signifier que les fonctions du système telles que l'*Enregistrement* ou *Modification d'une commande* se trouvent sur le poste de l'utilisateur. Et les opérations élémentaires de consultation des fichiers Client, Commande et Produit sont exécutées sur l'agent serveur à la demande des fonctions.

Signalons également qu'une architecture c/s orientée "donnée" pourrait facilement se généraliser à des modèles de *b.d. distribuée*. Dans ce cas un des "modules" du SGBD, tout en restant *serveur* pour les fonctions de D.L., deviendrait *client* des autres modules du SGBD, pour accéder aux autres composants de la b.d.

Architecture c/s orientée fonction

Par ce deuxième modèle, le serveur ne se contente pas d'exécuter des actions primitives sur la b.d., mais peut prendre en charge des fonctions entières de l'application.

Le serveur peut dès lors traduire lui-même les demandes d'information de l'application en actions élémentaires sur la b.d., et permet la REUTILISATION de ces fonctions par différents clients.

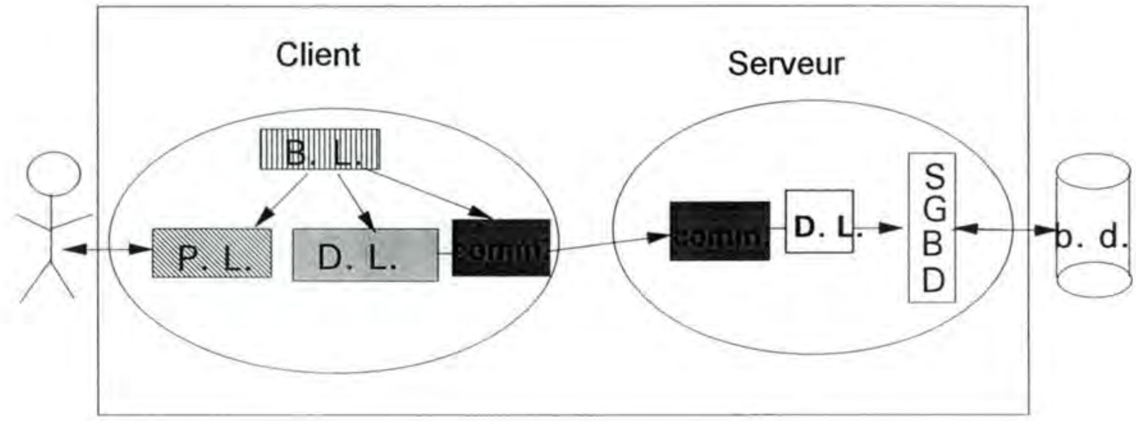


Figure 2.18: Architecture c/s orientée fonction

Dans notre exemple de gestion de commandes, cela signifierait que des fonctions entières de l'application, telles que *Enregistrement d'une commande* ou *Comptabilisation de l'ensemble des commandes d'un client*, se trouveraient sur l'agent serveur. Quant aux postes utilisateurs, ils prendraient en charge la gestion du dialogue et du contrôle de l'application.

Les communications "distribuées" se font donc:

- soit entre modules de D.L., plus précisément une fonction du client qui utilise les services d'une autre fonction, se trouvant sur le serveur;
- soit entre modules de B.L. et D.L., où le contrôle de l'application déclenche directement une fonction du serveur.

Au niveau du client, l'appel au module de communication (et aux API de middleware) ne peut donc se faire que par des modules de B.L. ou D.L.
Nuançons quelque peu ces propos en précisant, comme nous l'avons expliqué précédemment (cfr 2.2.2.2.), que les modules de P.L. et D.L. sont de même niveau hiérarchique.
Ils peuvent donc aussi s'appeler mutuellement en utilisant directement les modules de communication mais, s'il ne faut pas les perdre de vue, ces situations sont exceptionnelles (et, par souci de clarté, nous ne les avons pas représentées dans notre figure 2.18).

Constatations

Ce principe de réutilisation de fonctions, stockées sur un serveur, faisant appel à de nouvelles méthodes de programmation proches des conceptions O.O., semble ne pas encore avoir véritablement percé dans le domaine industriel. A ce jour, [Berso92], il semble que ce soient les architectures c/s orientées données les plus utilisées.

En général le serveur se contente donc d'exécuter sur la b.d., via un SGBD relationnel, des requêtes d'un (ou plusieurs) programme(s) client(s) telles que la sélection, la mise à jour ou la suppression d'enregistrements.
Les quelques fonctions éventuellement dévolues au serveur sont chargées de la vérification de contraintes d'intégrité (c'est-à-dire, [Bodar89], de "*propriétés non représentées par les concepts de base du modèle que doivent satisfaire les données appartenant à la mémoire du S.I.*"). Cette solution, [Sybas93] parle de "*data driven integrity*", offre l'avantage d'éviter toute redondance, le respect des règles d'intégrité se faisant une fois pour toutes au niveau du serveur, et pour différents clients.

Mais cette gestion des contraintes d'intégrité (C.I.) au niveau du serveur n'est nullement imposée au concepteur de l'application. Il peut également envisager une solution où les C.I. seraient vérifiées par chaque client, [Sybas93] parle d' "*application driven integrity*".
Si cette dernière solution semble moins performante, elle peut très bien se justifier par exemple pour des raisons de portabilité des applications clients sur de nouvelles b.d. (et SGBD).

Relation entre les agents

La **relation entre les agents** d'une architecture c/s est de type **maître/esclave**. Seul le client émet des requêtes et le serveur se contente d'y répondre

Rappelons qu'a priori ces rôles "client" et "serveur" ne sont **pas définitivement** liés à des machines bien précises. Un agent peut très bien jouer le rôle de client d'un agent, et serveur d'un autre.

Modules de communication

Pour rappel, ces modules doivent prendre en charge tous les traitements nécessaires à la communication entre deux agents. Les traitements qu'ils offrent sont supportés par des outils d'infrastructure, plus précisément des logiciels de middleware (cfr. 1.3.), qui assurent une transparence complète de communication entre les agents.

Au premier chapitre (cfr. 1.3), nous avons montré qu'indépendamment de la technique utilisée tout environnement de middleware doit exécuter les fonctions que nous avons qualifiées de:

- fonctions de l'Appelant
- fonctions d'interface,
- fonctions de l'Appelé.

Pour une architecture c/s, les modules de communication des agents **clients** doivent être pris en charge par ces fonctions d'***Appelant*** et d'***Interface*,** pour permettre au programme client, via des API de middleware:
- de localiser le serveur,
- d'identifier, ouvrir et fermer une communication avec le serveur,
- de transmettre une requête et demander son exécution,
- de réceptionner le(s) résultat(s) de la requête,

Quant au module de communication de l'agent **serveur,** il doit être implémenté par les fonctions d'***Interface*** et d'***Appelé*,** pour:

- réceptionner les requêtes,
- gérer la concurrence des requêtes, c'est à dire gérer des files d'attente, des ordres de priorité et des politiques d'exécution,
- identifier une communication,
- renvoyer les résultats aux clients.

Les fonctions de communication étant localisées, il reste au concepteur de l'application à se choisir un **environnement de middleware.** Si le choix final revient bien entendu au programmeur, et aux autres personnes impliquées par le S.I., nous allons tout de même conseillé l'un ou l'autre type d'environnement.

A cette fin nous adoptons uniquement un critère technique, même si bien d'autres considérations devraient être envisagées (cfr. 3.4.).

Au premier chapitre, nous avons répertorié les environnements de middleware selon la technique de communication utilisée (cfr 1.3.2.), nous avons ainsi distingué:

- les *Remote Procedure Calls* (RPC),
- les *SQL Interactions*, qui peuvent être éventuellement enrichis de mécanismes tels que:
  - les *triggered procedures*,
  - les *callback routines*,
- les *Pipes*.

Pour autant que l'infrastructure dispose d'un SGBD relationnel, la technique la plus appropriée pour exécuter des traitements distribués de Data Logic se fait bien entendu via un environnement de ***SQL Interactions.***

Si des fonctions entières de Data Logic sont séparées du reste de l'application (c/s orientée "fonction"), cet environnement doit permettre des solutions de *triggered procedures* ou *callback routines* (cfr. 1.3.2.2.).

Les deux autres techniques, RPC et Pipes, sont envisageables mais, n'étant pas orientées "traitements de Data Logic", ces solutions sont moins efficaces (pour le programmeur).

2.3.2.2 Cooperative processing

Par le cooperative processing, comme le définit [Gupta91]

> "*une application est fonctionnellement divisée en deux programmes [...] qui s'exécutent sur des ordinateurs différents*".

Répartition des traitements

Conformément aux principes de [CA91], [Berso92], et suivant la terminologie de notre modèle général des systèmes de gestion, une architecture de **cooperative processing se caractérise par la distribution des modules de Business Logic, c'est-à-dire du contrôle de l'application, sur différents agents**.

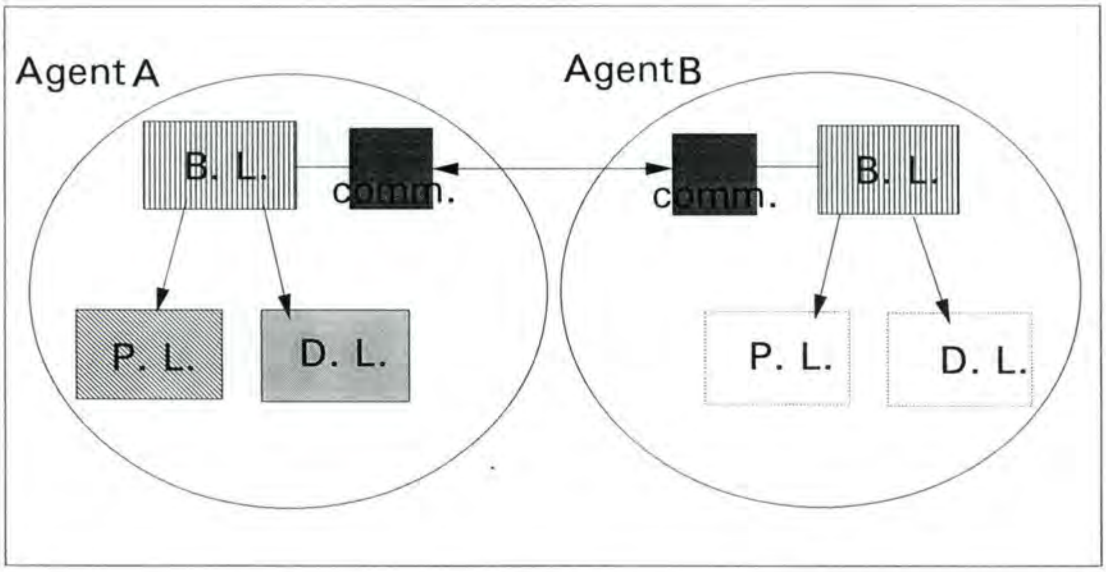


Figure 2.19: Architecture de cooperative processing

Pour reprendre l'analogie avec une table de décision, cela signifie qu'elle est répartie sur plusieurs agents. Dès lors, la table de décision d'un agent peut générer une action qui fera passer l'application à un état exporté, c'est-à-dire se trouvant dans la table d'un autre agent.

Relation entre les agents

Comme l'explique [CA91], les agents coopèrent en tant que véritables partenaires dans la réalisation d'un objectif. Nous pouvons qualifier leur **relation de type "*égal-à-égal*"**, c'est-à-dire que:

- a priori, n'importe quel agent peut émettre une requête à son vis-à-vis,
- chaque agent est contrôlé par des traitements de B.L..

Comme nous l'avons déjà précisé (cfr 2.2.2.2.), les modules de B.L. se trouvent au niveau supérieur de notre hiérarchie, et ne peuvent être utilisés par aucun module de niveau inférieur, c'est-à-dire de P.L. ou D.L.

Dès lors, toutes les communications distribuées entre agents de cooperative processing se font entre modules de B.L. Ces agents peuvent ainsi prendre en charge:

- soit des **sous-ensembles distincts** de l'application, par exemple des "phases" de [Bodar89], possédant chacun leur B.L.,
- soit des **applications différentes,** qui coopèrent dans la réalisation d'un objectif de plus haut niveau (cfr 3.3.).

Dans l'exemple de gestion de commandes, un cooperative processing pourrait se traduire par la répartition des activités d' *Enregistrement* de différentes commandes, de *Préparation* des bons de stocks, et de *Facturation* des commandes sur différents agents matériels.

Modules de communication

Cette relation d'égalité entre les agents implique une communication "full-duplex". Chaque partenaire pouvant émettre une requête ou fournir un service, tout module de communication doit garantir les traitements suivants:

- localiser son partenaire,
- identifier, ouvrir, et fermer une communication,
- transmettre des messages (requêtes ou résultats),
- réceptionner des messages (requêtes ou résultats),
- gérer la concurrence des requêtes (files d'attente, ordres de priorité et politiques d'exécution),

Ces opérations sont donc prises en charge, au niveau de l'infrastructure de **chaque agent**, par les fonctions d'et d'***Appelant*, d'*Interface et d'Appelé*** de tout environnement de middleware.

Quant au choix de cet environnement, la distribution ne portant plus sur les traitements de Data Logic, cela écarte quelque peu les techniques de *SQL Interactions*.

Les logiciels de middleware basés sur les *Remote Procedure Calls* (RPC) et les *Pipes* semblent, par leur neutralité, plus appropriés; mais reprécisons ces deux solutions:

- les RPC permettent à un processus de déclencher et exécuter un autre processus, qui réside sur un agent distant,
- les Pipes gèrent des communications entre processus.

Dès lors, les ***RPC*** semblent plus appropriés aux situations de cooperative processing où un agent veut, via sa B.L., déclencher un fil d'activités (par exemple, une "phase" au sens de Bodar89]) se trouvant sur un autre agent, possédant aussi une B.L. .

Les ***Pipes***, étant des moyens de communication inter-processus, ils conviennent d'autant mieux au cooperative processing où des activités distinctes sur deux agents s'exécutent simultanément (par exemple deux applications), et veulent s'échanger des informations via leur B.L. .

2.3.2.3 Remote Presentation processing

Répartition des traitements

Cette dernière classe d'architecture distribuée consiste simplement à **séparer les modules de Presentation Logic du reste de l'application**.

Le dialogue entre les agents se fait essentiellement entre B.L. et P.L.
Même si des modules de P.L. pourraient directement appeler des traitements de D.L., ou inversement, ces relations étant assez rares (cfr 2.2.2.2.), nous considérerons que l'agent chargé de la P.L. est entièrement sous le contrôle de la B.L. de l'autre agent.
Cette architecture peut sembler peu performante, notamment parce qu'une machine responsable uniquement de la gestion du dialogue ne sera certainement pas utilisée à sa pleine capacité. Cependant cette solution peut très bien convenir:

- à des "*nonconversational applications*", [Berso92], où les interactions avec l'utilisateur sont peu nombreuses et prédéterminées,

- à des rénovations de S.I. où, par exemple, le concepteur se contenterait d'améliorer l'interface homme-machine d'une application déjà développée sur un mainframe, en adoptant un dialogue de manipulation directe et le transfert de cette interface sur un P.C. .

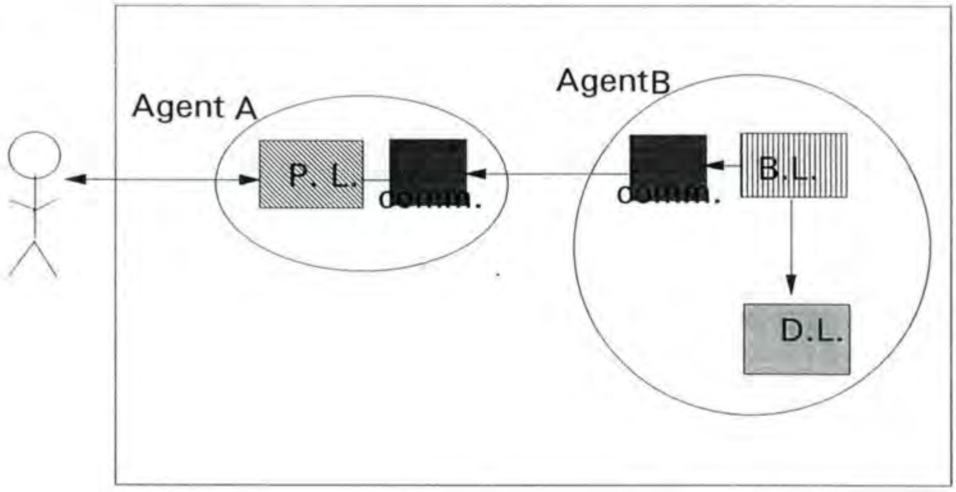


Figure 2.20: remote Presentation Processing

Module de communication

L'agent responsable des traitements de B.L. doit disposer des traitements de communication suivants (qui devront être implémentés par les fonctions d'Appelant et d'Interface de tout environnement de middleware):

- localiser les autres agents,
- identifier, ouvrir et fermer une communication,
- transmettre des requêtes de traitements de P.L.,
- réceptionner des messages-résultats de la P.L. .

Au niveau de l'agent chargé du dialogue, il faut associer traitements (fonctions d'*Interface* et de l'*Appelé*):

- la réception de requêtes, et une éventuelle gestion de la concurrence,
- identifier une connexion,
- transmettre les messages résultats.

Une technique de middleware adéquate peut être de type RPC ou alors, solution idéale pour une telle distribution, la technique ***X Window System***. Cette technique définit un dialogue , *X protocol*, uniquement pensé et orienté "traitements de P.L. distribués"; cependant nous ne la détaillerons pas..

Chapitre 3

Chapitre 3: Critères de distribution

3.1. Introduction

Dans ce chapitre, nous présenterons les facteurs qu'il nous semble important de prendre en considération lors de l'élaboration d'une application distribuée.
Il s'agit, en fait, d'exposer quelques réflexions permettant de déterminer la pertinence d'une solution distribuée pour une application de gestion donnée, et d'orienter son concepteur vers l'architecture la plus adéquate.

Précision de la définition d'une A.D.

Une application distribuée (A.D.) sera vue comme un ensemble de traitements d'une même application **répartis sur les agents les plus appropriés** pour les exécuter.

Par "approprié", on entend montrer que le choix d'une architecture distribuée relève d'un ensemble de facteurs (techniques, logiques, organisationnels, ...) dont le poids dans le processus de décision peut varier d'une application à l'autre.

Démarche

Dès lors qu'il nous semble inutile de vouloir dresser les avantages et inconvénients de chacune des classes d'A.D (définies au deuxième chapitre, cfr 2.3), nous préférons préciser divers besoins de distribution que peuvent rencontrer des applications de gestion, et pour chacun de ces besoins, que nous appellerons **"critères de distribution"**, déterminer l'architecture distribuée la plus satisfaisante.

Précisons que la liste des critères que nous retiendrons est loin d'être exhaustive. Nous présenterons seulement des principes que nous avons déduits de notre étude de différentes applications distribuées, pour la plupart développées par le service informatique d'une société de l'industrie chimique.

Tous ces éléments sont donc uniquement tirés d'analyses empiriques, que nous présenterons tout au long de ce chapitre. Ils ont simplement pour objectif de faciliter et clarifier la réflexion de l'informaticien lors de la conception d'une architecture, mais certainement pas de lui imposer une architecture distribuée.

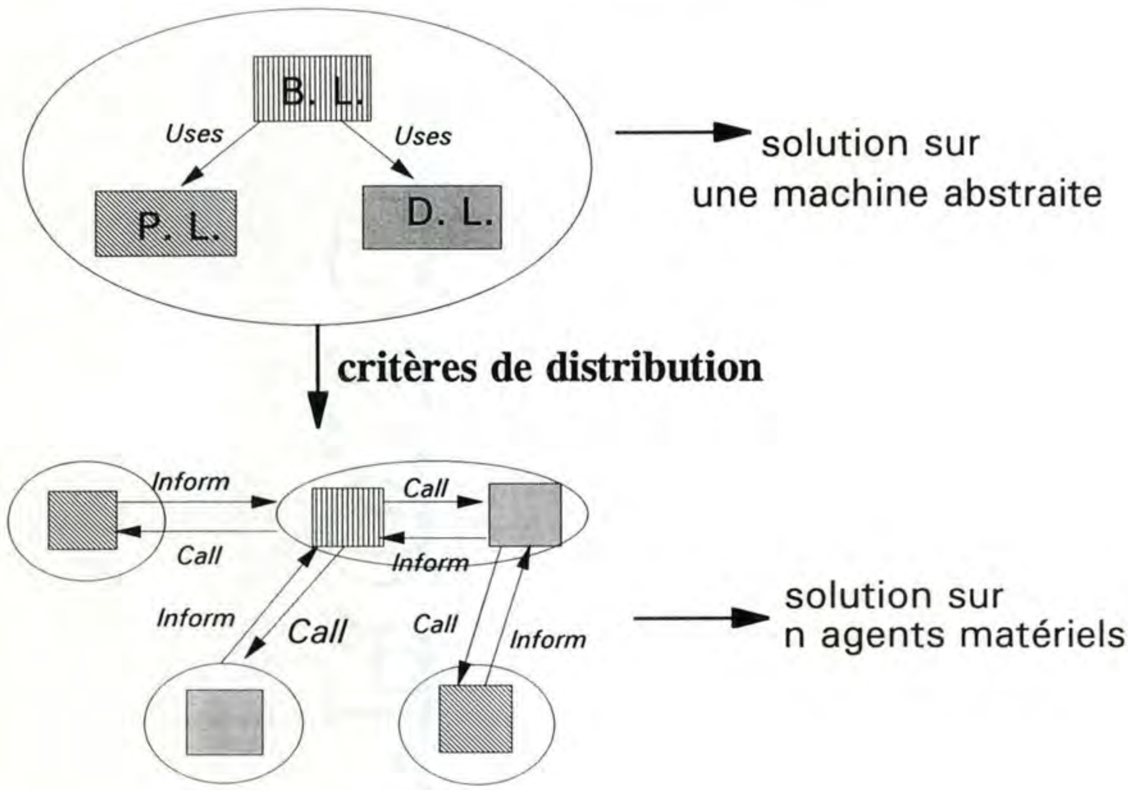


Figure 3.1: Critères de distribution

La démarche de réflexion que nous proposons au concepteur de l'application se déroule en deux temps:

- tout d'abord nous conseillons d'analyser le **type de l'application** et son **intégration** dans différents S.I. déjà existants, ces deux critères permettant de se donner une première idée d'une distribution de traitements, indépendamment du matériel utilisé;

- ensuite, par des critères appelés de **rightsizing**, il s'agit d'affiner la répartition des traitements sur différentes plates-formes (et outils de middleware) disponibles et nécessaires selon des considérations de coût/efficacité des équipements, de politique et d'organisation de l'entreprise.

Précisons qu'avant de procéder au rightsizing d'une application, un autre critère devrait certainement être évalué: la REUTILISATION. Cependant par manque d'expérience, et des notions de réutilisation de traitements qui commencent seulement à se forger,et sont encore mal maîtrisées, nous n'aborderons pas ce critère.

Ainsi, nous examinerons les critères de distribution suivants:

- **le type d'application,**
- **l'intégration,**
- **le rightsizing.**

Dans le quatrième chapitre, nous illustrerons et combinerons l'ensemble des concepts au travers d'une étude de cas.

3.2. Le type d'application

Par ce premier critère, nous allons classer simplement les S.I. pour en conseiller, de manière très intuitive, le choix de l'un ou l'autre modèle de distribution (client/serveur, cooperative ou remote presentation processing) pour l'ensemble de l'application.

A cette fin, nous nous baserons sur la classification proposée par Keen & Scott Morton où, comme l'explique [Bodar89], les applications informatiques sont rangées selon deux axes:

*"- d'une part, du **niveau de fonctionnement*** [du S.I.] *au sein de l'organisation, du plus opérationnel au plus stratégique,*

*- et d'autre part, du **degré de structuration des décisions** qui s'y prennent, des plus structurées aux plus informelles."*

Classification par degré de structuration	Classification par niveau Opérationnel →			Stratégique
Structurées	- Tenue des stocks - Tenue de la comptabilité - Calcul des salaires - Exécution de processus de fabrication	- Gestion des stocks - Contrôle de qualité - Calcul de prix de revient - Calcul de productivité - Contrôle de production		
		- Gestion de la trésorerie - Ordonnancement de la production - Gestion des équipes de production - Suivi des budgets - Suivi de la production, des ventes, etc.	- Elaboration des prévisions de ventes - Elaboration des budgets - Définition d'un tarif	- Analyse d'un plan d'investissement - Analyse d'un plan de financement - Localisation d'une usine
↓ Informelles			- Recrutement des cadres - Négociation de conventions paritaires - Négociation de contrats d'approvisionnement	- Choix des projets R & D - Choix d'une stratégie de marchés - Lancement de nouveaux produits - Fusion avec une autre organisation

Nous allons utiliser cette classification en la redéfinissant dans les termes de notre architecture générale des applications de gestion (proposée au deuxième chapitre, cfr 2.2.1).
Cette redéfinition, qui sera volontairement caricaturale et correspondra réellement à un nombre très restreint de S.I., permettra de se faire une idée générale des possibilités de distribution de l'application.

Ainsi nous retiendrons deux types opposés de S.I., que nous appellerons:

- les systèmes "**transactionnels**", pour des applications se situant à un niveau de gestion purement opérationnel et basées sur des décisions parfaitement structurées, (c'est-à-dire permettant essentiellement d'exécuter des *transactions* sur la b.d.);

- les systèmes "**analytiques**", pour des applications de gestion stratégique reposant sur des décisions semi structurées voire informelles, (c'est-à-dire servant d'outils d'*analyse* au gestionnaire).

3.2.1. Applications transactionnelles

• **Traitements de:**

- **Business Logic** (B.L.)

Reposant sur des processus de décision structurés, la principale caractéristique des applications transactionnelles est d'automatiser entièrement le séquencement des opérations du système.
Comme l'explique [Bodar92], ce sont des *"S.I. destinés à l'automatisation de processus physiques (process control) et de procédures administratives [...ayant] pour objet les flux de base d'une organisation"*.
Le **contrôle** étant **entièrement automatisé**, cela signifie que l'ordre d'exécution des traitements de P.L. et D.L. est entièrement pris en charge par la B.L.

- **Presentation Logic** (P.L.)

Actuellement la majorité des applications transactionnelles se basent sur un dialogue de **style conversationnel**.

Cet état de fait est souvent justifié par le nombre important d'utilisateurs qui, en général, possèdent une bonne connaissance du monde supposé par l'interface, et ne cherchent qu'à exécuter rapidement de nombreux traitements sur la b.d.

- **Data Logic** (D.L.)

  - Les fonctions de l'application travaillent sur de petites quantités de données,

  - elles sont fréquemment exécutées, et leur résultat dépend souvent des fonctions exécutées précédemment,

  - les actions primitives sur la b.d. appelées par les fonctions, sont essentiellement des opérations de **mise à jour**.

Il est à noter que, souvent, certaines tâches moins critiques pour l'application, et ne nécessitant pas d'intervention des utilisateurs, sont déléguées à des programmes "*batch*", s'exécutant à des périodes de moins grande activité du système. Par exemple, la consolidation des différents comptes des comptes clients d'une banque s'exécutent en général la nuit.

• **Données d'une application transactionnelle**:

- informations "up to date" (voire même "up to minute"): à tout moment les données reflètent l'existant et les mises à jour sont immédiatement enregistrées,

- pas (ou peu) d'information à caractère historique,

- les donnés enregistrées constituent des informations détaillées, et représentent souvent des entités du monde réel (données primaires).

Exemple

Prenons le cas d'un système de réservation de billets d'avion, il s'agit bien d'une application transactionnelle, étant donné que:

• au niveau des traitements:

- de B.L.: la réservation de billets se fait selon des procédures bien établies et prévisibles,

- de P.L.: une telle application possède, en général, beaucoup d'utilisateurs, à savoir des agences de voyage; cela peut donc générer de nombreux traitements sur la b.d., commune à l'ensemble du système;

- de D.L.: ce sont essentiellement des traitements de mise à jour et consultation de données telles que les destinations des avions, leurs heures de vol, les places disponibles, etc ...; et le résultat d'un traitement dépend des opérations exécutées précédemment.

Par exemple, et trivialement, si l'utilisateur veut consulter les places encore disponibles pour un vol donné, il faut qu'auparavant toutes les opérations de réservation aient été enregistrées.

- au niveau des données:

Comme expliqué précédemment, il est primordial pour un tel type d'application que les données reflètent parfaitement l'existant. Des actions, telles que la réservation de places doivent être immédiatement mémorisées.
Ces données contiennent peu d'information à caractère historique. Il est, par exemple, peu intéressant de connaître les dix derniers vols d'un avion donné (du moins pour l'application "réservation de billets").

Choix d'une architecture

Pour conseiller **a priori**, c'est-à-dire indépendamment des critères que nous présenterons par la suite, le choix de l'une ou l'autre architecture, il faut reprendre l'examen des composants d'un système transactionnel et tenter d'en déduire quelques principes.

- Les traitements de B.L. automatisant entièrement le traitement de l'application, il faut encore préciser le séquencement des opérations.

  - Dans le cas d'un *contrôle séquentiel* où, pour rappel, il existe une seule séquence possible des opérations de l'application, mieux vaut **centraliser** la B.L. sur un seul agent.

  Même si ce n'est pas irréalisable, nous préférons exclure toute architecture de cooperative processing. Le cheminement d'une exécution séquentielle sur différents agents ne semble pas idéale, essentiellement pour des raisons d'efficacité et de gestion de la sécurité (où l'interruption des activités peut s'avérer catastrophique.).

  - Par contre, pour des contrôles de type *multi-fils ou concurrent*, le **cooperative processing** semblera d'autant plus intéressant que les différents flux d'activités de l'application pourront être pris en charge par des machines spécialisées, voire même s'exécuter simultanément.

  Ces considérations de performance et de fiabilité devront être affinées par le concepteur, lors du "rightsizing" de son application (cfr 3.4).

- Le dialogue conversationnel ne se caractérisant pas par une haute complexité des traitements d'interface, mais une grande fréquence d'utilisation de ces traitements de P.L., cela ne plaide **pas en faveur de la séparation de la P.L.** du reste de l'application transactionnelle.
  Le principal avantage qui pourrait tout de même être retiré d'une telle distribution, qui correspond donc à du remote presentation processing (voire même du client/serveur), serait la prise en charge du contrôle syntaxique des données saisies, directement par l'agent utilisateur.

- Connaissant la nature des données transactionnelles et les caractéristiques des traitements de D.L., mieux vaut laisser les fonctions de l'application le plus proche possible des données et du reste du système.

  A nouveau cela se justifie par des raisons d'efficacité et de sécurité où, par exemple, les nombreux accès à des données distantes du reste de l'application doivent se faire de manière totalement fiable (surtout pour ces données concernant les *flux de base de l'organisation*) et s'exécuter rapidement.
  Ces motivations devront également être affinées par le concepteur lors de son évaluation des critères de "rightsizing" (cfr 3.4).

Exemple

Reprenons le cas d'un système de réservation de billets d'avion et considérons, de manière extrêmement simplifiée et abusive, le contrôle de l'application comme étant séquentiel. Dans ce cas, une bonne solution semble être la centralisation de l'ensemble des traitements et données sur une seule machine, et chaque utilisateur (agence de voyage) étant relié à cet machine via des terminaux.

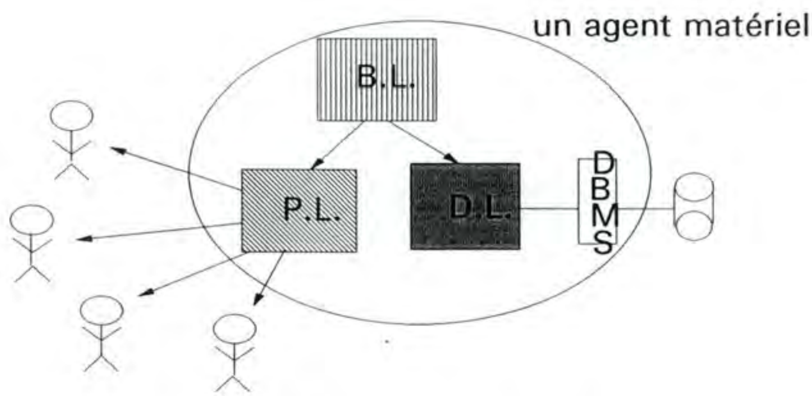


Figure 3.2: Application centralisée

Insistons sur le fait que même pour un tel système, il n'est pas impossible d'adopter une architecture distribuée, mais la motivation de cette solution devrait certainement reposer sur d'**autres critères que le seul type d'application à développer.**

Ainsi, on pourrait par exemple envisager une solution de "local data repository" ou serveur intermédiaire, où chaque agence de voyage (ou groupe d'agences) posséderait, sur un poste de son réseau local, des données qui lui seraient exclusives et régulièrement mises à jour sur le système central.
Cette deuxième solution peut dès lors être vue comme l'échange d'informations entre deux applications (cfr 3.3), où l'application "*Centralisation des Réservations*" délègue des données à des serveurs intermédiaires permettant aux applications clients "*Réservation parAgence*" de mener leurs activités simultanément.

Cette solution, permettant une certaine désynchronisation des activités et des opérations concurrentes, sera détaillée par la suite (cfr. 3.3.1).

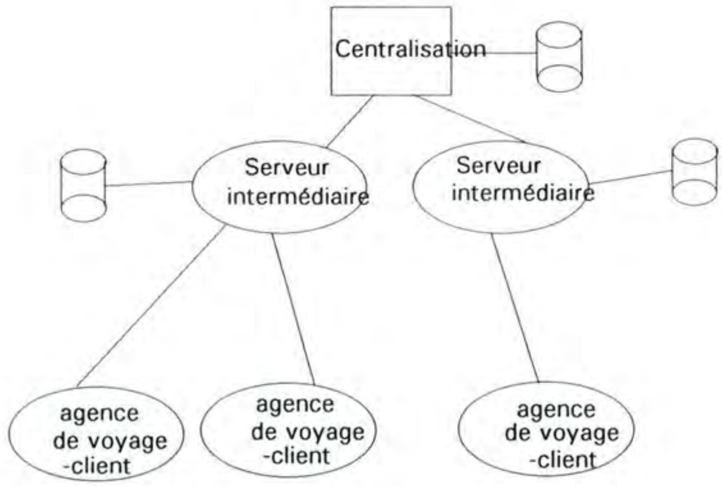


Figure 3.3: Solution de client/serveur

3.2.2. Applications analytiques

• **Traitements de:**

- **Business Logic** (B.L.)

Etant donné que, pour un système analytique, les processus de décision ne sont pas parfaitement structurés (mais semi-structurés ou informels), il est bien évidemment impossible pour la B.L. d'entièrement automatiser le séquencement des opérations.
Pour un tel système, [Bodar92], *"il n'est pas aisé d'isoler de façon précise les différents types d'activité et de mettre en évidence des séquences d'exécution [...]. Les gestionnaires procèdent selon des schémas d'activités multiples avec interférences fréquentes entre celles-ci"*. L'activité de l'utilisateur n'étant pas totalement formalisable, l'application sert seulement d'aide à la décision, d'outil analytique et revient souvent sous le terme S.I.A.D. (Système d'Information d'Aide à la Décision) ou encore S.I. stratégique.

- **Presentation Logic** (P.L.)

La plupart de ces systèmes gère un dialogue de **manipulation directe.**
Ce dialogue, se faisant donc en manipulant des représentations visuelles des éléments de l'application, est d'autant plus avantageux qu'il facilite l'apprentissage d'un système qui est utilisé ponctuellement et par un petit nombre d'utilisateurs.

- **Data Logic** (D.L.)
  - les fonctions de l'application travaillent sur de grandes quantités de données,

  - elles sont exécutées ponctuellement, et plus ou moins indépendamment les unes des autres,

  - les actions primitives sur la b.d., appelées par les fonctions, sont essentiellement des opérations de **lecture**.

• **Données des applications analytiques**:

- ne constituent pas nécessairement des informations toutes récentes (pas "up to minute"),

- les données utilisées sont souvent à caractère historique,

- et généralement sont l'agrégation de données transactionnelles (elles ne représentent donc pas nécessairement des entités du monde réel).

Exemple

Examinons le cas d'un chef de département devant rendre, deux fois par an à son comité de direction, ses plans et budgets de production pour les six mois suivants.
Dans une telle situation l'application utilisée ne peut qu'être de type "analytique", et servir d'outil au gestionnaire.

- Au niveau des traitements de l'application:

  - de B.L.: il est difficile, voire impossible, de prévoir la manière dont le décideur va procéder, et de cerner l'ensemble des paramètres à prendre en considération (tendance du marché, position de la concurrence, politique de publicité, ...),

  - de P.L.: l'application ne va servir qu'à un seul, (ou quelques), responsable(s) de département; par une utilisation bisannuelle, ils n'ont certainement pas le temps de s'y familiariser, et peuvent avoir besoin d'une aide par rapport aux concepts et modèles utilisés,

  - de D.L.: ce sont essentiellement des actions de consultation des données, de calculs arithmétiques, ou d'utilisation de modèles statistiques, de R.O.

- Au niveau des données:

  Les données utilisées pour élaborer les budgets proviendront de b.d. opérationnelles. Ces informations sont agrégées (par unité de production, unité de temps,. ..) puis stockées dans des b.d. "analytiques".

Choix d'une architecture

En procédant selon la même démarche que pour les applications transactionnelles, nous pouvons facilement établir quelques constatations.

- Outre qu'elles n'automatisent pas entièrement le contrôle des activités, les applications analytiques se distinguent des transactionnelles par le niveau d'efficacité (temps de réponse) exigé, qui sera moins élevé. En effet, les applications anlytiques ne travaillant pas directement sur les flux de base, une efficacité moins grande sera tolérée.
  Pour ces raisons, même dans le cas d'un contrôle séquentiel, une architecture de cooperative processing peut être envisagée.

Dissocions tout de même les DSS (Decison Support System) des GDSS (Group Decison Support System).
Dans le cas d'un DSS, une architecture de cooperaive processing cherchera essentiellement la distribution d'outils coopérants,

par exemple, la coopération d'outils d'aide à la gestion tel qu'un tableur, un programme linéaire et un SGBD répartis sur différentes mmachines de l'environnement distribué.

Pour des situations de GDSS, où la prise de décisions repose sur un groupe de personnes, on cherchera plutôt la coopération des outils utilisés par ces différentes personnes.

- Pour les traitements de P.L., gérant le plus souvent un dialogue de manipulation directe, mieux vaut les déléguer aux différents postes de travail des utilisateurs (c'est-à-dire adopter des architectures de remote presentation ou client/serveur) pour autant qu'ils permettent de développer des interfaces graphiques, etc...

- Quant aux traitements de D.L., pas nécessairement exécutés fréquemment, mais travaillant sur des grandes quantités de données, leur prise en charge par des **serveurs spécialisés** semble très intéressante.
  Cette solution client/serveur est d'autant plus appropriéequ'elle permet à chaque gestionnaire de mener ses traitements sur les données comme il l'entend, via ses modules de P.L. et B.L. de son poste client. Cela signifie que sans faire appel à un informaticien, l'utilisateur pourra exécuter ses actions primitives sur la b.d. comme il l'entend (pour autant qu'il en ait l'autorisation).

Exemple

Pour reprendre le cas de l'élaboration de budgets, on peut très bien envisager une solution où toutes les données concernant l'entreprise sont centralisées sur un seul serveur. Chaque responsable peut accéder, via son poste client, aux données de son département et à certaines informations générales à l'entreprise. Ils peuvent exécuter toutes les requêtes de consultation qu'ils désirent sur ces données.

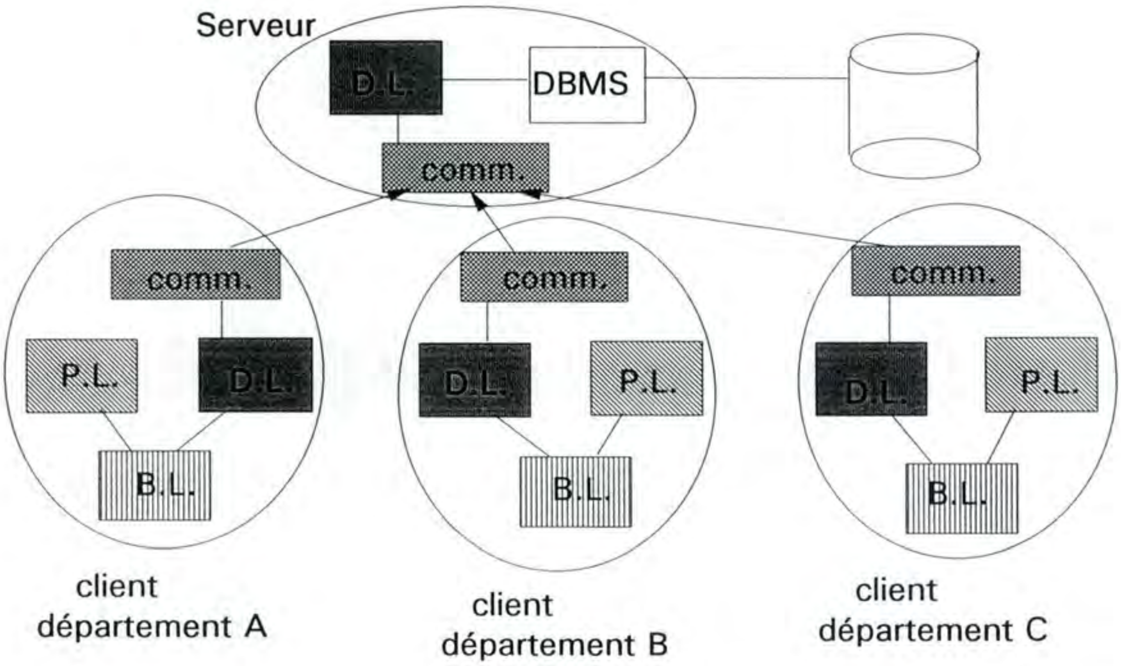


Figure 3.4: Solution client/serveur

Supposons qu'en même temps qu'il collecte différentes informations sur le serveur, le responsable C veut générer des opérations d'analyse d'investissements sur base de nombreux calculs statistiques.
Une bonne solution serait de confier ce nouveau "fil d'activités" à un autre agent selon un lien de cooperative processing.

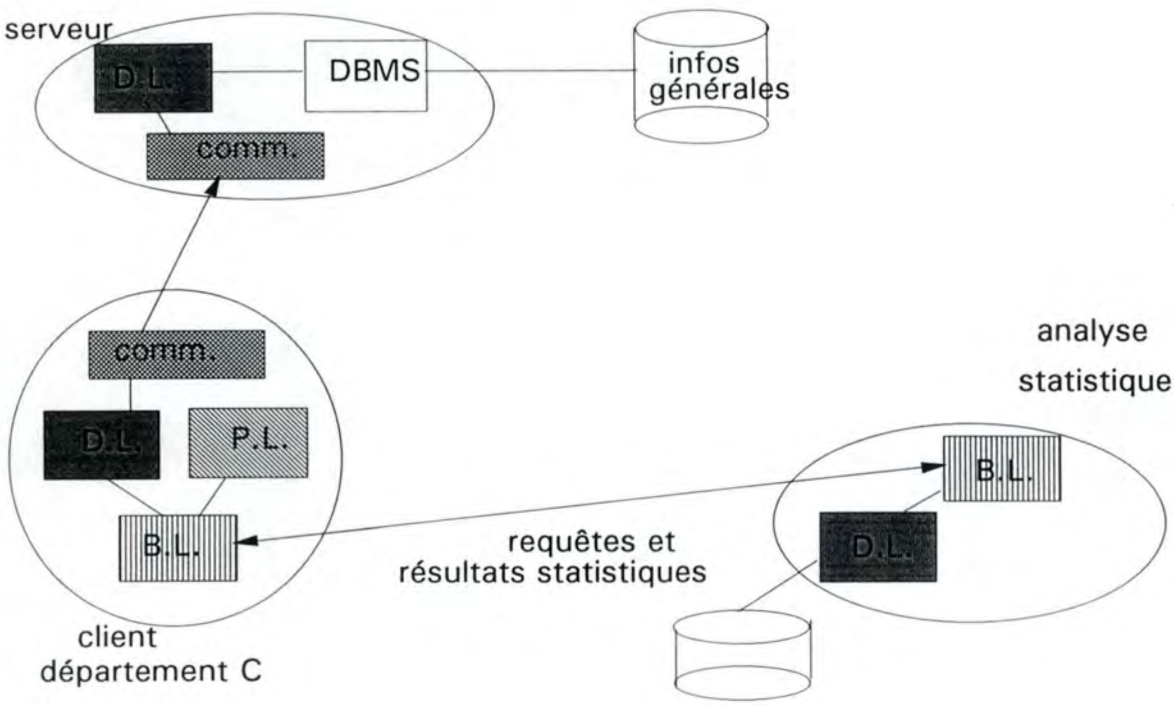


Figure 3.5:: Solution c/s et cooperative processing

3.3. Critère d'intégration

Par ce deuxième critère, il s'agit de définir la meilleure solution permettant d'intégrer deux applications, c'est-à-dire de supporter les échanges d'informations pour:

- éviter un maximum de redondances des données (par exemple, dans les b.d. de chaque S.I.), et des activités (par exemple, le double encodage manuel des données),

- minimiser les pertes d'information au sein de l'entreprise.

Dans le cas d'une nouvelle application, cela devrait imposer au concepteur, avant de la développer, d'eefectuerà une véritable analyse de l'existant des différents S.I. pour y relever les informations pertinentes.

Nous classons ces informations en deux catégories, très influentes dans le choix d'une architecture destinée à supporter l'intégration, à savoir des informations de type:

- "**donnée**",

- "**événementiel**".

3.3.1. Information de type "donnée"

Nous dirons qu'une information présente dans un S.I. est de type "donnée" pour une autre application, si elle remplit les conditions suivantes:

- l'information doit être **récupérable**, c'est-à-dire stockée dans la mémoire d'un S.I. dont la *structuration des informations est connue* (ou compréhensible via des techniques de décodage) par l'application requérante;

- l'enregistrement de l'information dans la b.d. du S.I. doit se faire sans perte sémantique par l'application requérante, (par exemple l'enregistrement d'une information ne peut se faire avec perte de relations à des objets, faits ou événements présents dans l'application requérante mais pas dans les S.I. de son environnement);

- l'information n'est pas *critique* pour l'application requérante, c'est-à-dire que toute mise à jour de cette donnée ne doit pas lui être immédiatement signalée.

Choix d'une architecture

A priori, une réutilisation idéale de ces informations de type "donnée" se ferait via **une architecture client/serveur**, où:

- aux b.d. des S.I. environnants, on adjoindrait des serveurs de traitement de D.L. (serveurs orientés "donnée" ou "fonction",

- ces serveurs pouvant satisfaire les requêtes d'agents clients chargés des traitements de P.L. et B.L. (et d'une partie de la D.L.) des applications requérantes.

Cette solution est d'autant plus intéressante qu'elle permet à chaque application requérante d'individualiser sa récupération des données et peut facilement intégrer toute autre application de gestion.

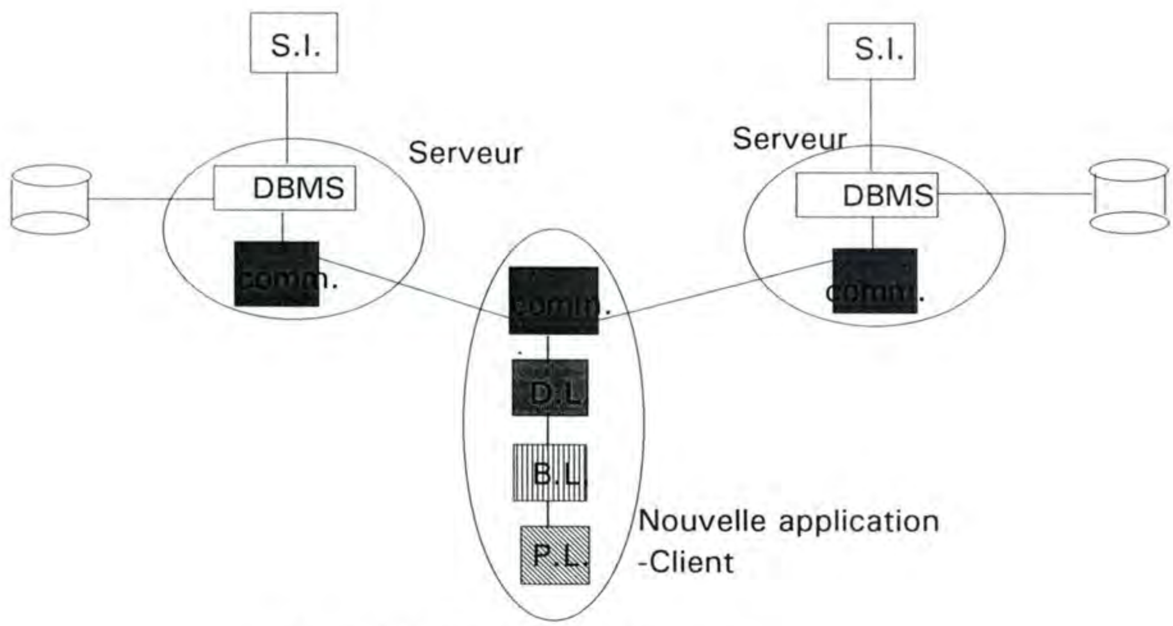


Figure 3.6: Solution client/serveur.

Conséquence

Toute solution de type c/s entraîne une gestion de la **concurrence d'accès aux données**.
Les informations échangées n'étant pas critiques pour l'application requérante, le concepteur peut facilement déléguer cette gestion de la concurrence:

- soit aux modules de communication des serveurs associés aux S.I. environnants, en affectant une priorité très faible aux nouvelles applications clients, laissant ainsi la priorité d'accès au système qui détenait le premier l'information;

- soit en adoptant une solution de "data repository" (ou même un véritable serveur "intermédiaire") où les données sont ponctuellement, par exemple une fois par jour, transférées des S.I. environnants et centralisées vers le repository (ou la b.d. du serveur intermédiaire)..

Exemple

Une application analytique alimentant sa b.d. avec des informations en provenance de b.d. liées à des applications transactionnelles, illustre parfaitement ce principe de récupération de données.

Supposons que pour établir leurs prévisions budgétaires, les responsables de départements aient besoin d'informations transactionnelles stockées dans le système GPAO (Gestion de Production Assistée par Ordinateur) utilisé par l'ensemble de l'entreprise. Ce GPAO et sa b.d., véritables moteurs de la production de l'entreprise, ne pouvant être monopolisés par une simple manipulation analytique, on pourrait adopter une solution telle que:

- les responsables de département réalisent leurs analyses sur leurs propres machines (P.C., stations de travail, ...), permettant d'individualiser leurs activités, via lesquelles ils accéderaient à des serveurs de données analytiques;

- ces serveurs s'alimentent périodiquement, par exemple la nuit, par des transferts et agrégations d'informations transactionnelles.
Il est à noter que ces transferts périodiques entre b.d. transactionnelles et analytiques peuvent tout aussi bien se réaliser par des techniques de "file transfer" que client/serveur (le serveur analytique devenant client d'un serveur transactionnel).

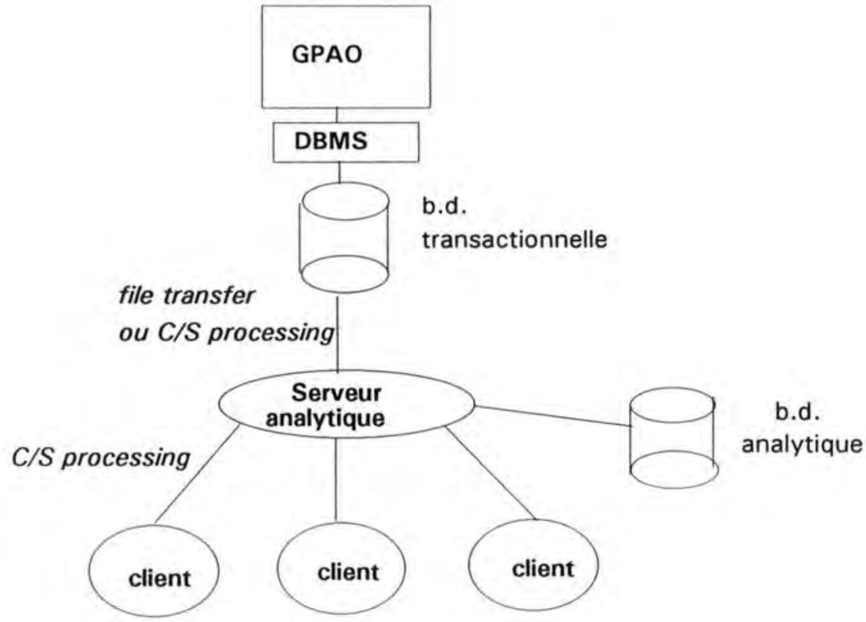


Figure 3.7: Solution client/serveur.

Cette solution de serveur intermédiaire est d'autant plus intéressante qu'elle permet de repenser l'organisation des données, dont le design de départ n'avait peut-être pas été développé dans un souci d'intégration d'applications.

3.3.2. Information de type "événementiel"

Nous dirons qu'une information présente dans un S.I. est de type "événementiel" pour une autre application :

- si sa survenance (ou sa mise à jour) doit être *immédiatement signalée* à l'application requérante, sinon l'information serait perdue ou irrécupérable,

- ou si l'information transférée est immédiatement "*traitée*" par l'application concernée, c'est-à-dire nécessite des *opérations de niveau B.L.*.avant d'être enregistrée dans la b.d. (et ne peut être stockée telle quelle, brute).

Choix d'une architecture

Pour reprendre l'analogie avec les tables de décision, exposée au deuxième chapitre, lorsque la B.L. d'un S.I. environnant se rend compte qu'elle dispose d'une telle information, elle doit générer un événement dans la table de décision de l'application requérante.

Sur base de cet événement et de l'état courant de l'application, la B.L. doit déclencher un ou plusieurs traitements de P.L. ou D.L., tels que l'envoi de messages à l'utilisateur, la modification d'éléments de sa b.d., ...

Etant donné que, pour la B.L. du système requérant, l'événement ne provient pas des autres composants de son application, mais des modules de B.L. d'un autre S.I., il semble que l'intégration devrait se faire via une architecture de **cooperative processing**.

Exemple

Reprenons le cas d'un système de GPAO, chargé de la planification et du suivi du processus de production de l'ensemble d'une entreprise. Examinons cette fois-ci son interaction avec une application, appelée OLSE (Ordonnancement Lancement Séquencement et Enchaînement), devant déterminer l'enchaînement des procédures de fabrication pour un produit et des ressources disponibles données.

Supposons que cette application OLSE déclenche un fil d'activités pour chaque ordre de production lui provenant du GPAO, comprenant le type de produit à fabriquer et les quantités de matières premières nécessaires.
Les activités de OLSE démarrant immédiatement lorsque ces événements "ordres de production" lui parviennent, nous pouvons considérer les informations à échanger entre les deux systèmes comme des événements.

Après avoir exécuté ces ordres de production, OLSE doit retourner au GPAO les quantités de matières effectivement utilisées. Ces éléments doivent également être immédiatement pris en charge par le système de GPAO afin d'adapter ses planifications.

Un lien de cooperative processing semble donc bien convenir entre ces deux applications.

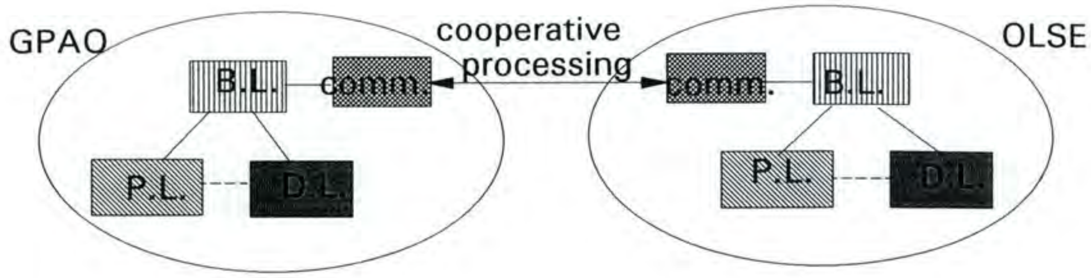


Figure 3.8: Solution de cooperative processing.

Conséquence

Un échange d'information entre deux applications par une architecture de cooperative processing nécessite une gestion de la **synchronisation de leurs activités**.

Si le système devant réceptionner un événement s'exécute sous un contrôle multi-fils ou concurrent, les échanges ne posent aucun problème étant donné que la gestion de ses activités est asynchrone, et celles-ci peuvent être interrompues à tout moment.

Par contre, si le système requérant est sous contrôle séquentiel:

- soit l'application peut prévoir ou attendre les moments où parviendront des événements externes; dès lors cela ne pose pas de problème,

- soit ces informations peuvent survenir à n'importe quel moment et l'application ne peut suspendre ses activités; alors le concepteur doit adopter des solutions intermédiaires permettant tout de même d'échanger indirectement des événements de B.L. à B.L. .

Un exemple d'une telle solution serait la mémorisation de l'événement dans un "data repository" commun aux deux applications, et que le système sous contrôle séquentiel consulterait à intervalles de temps réguliers pour y relever les événements le concernant.

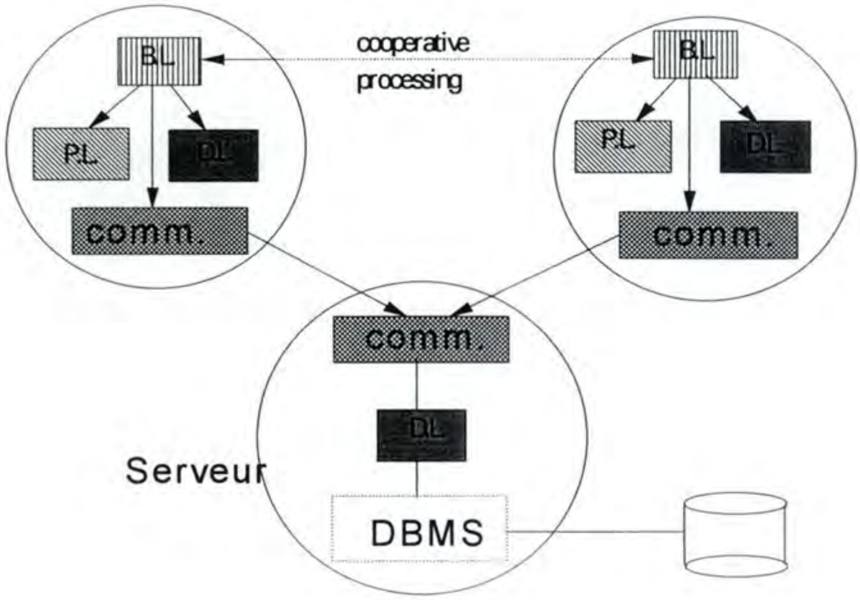


Figure 3.9: Solution intermédiaire de client/serveur.

Il faut préciser que nous ne pouvons apporter actuellement de solution universelle à un tel type de problèmes. Si l'échange d'événements ne se fait pas directement, il faut tenter de perdre le moins d'information possible, et la solution varie d'une application à l'autre.
Cela dépend notamment des quantités d'information, de l' "importance" de l'événement à échanger, etc ...

Nous illustrerons un cas bien précis d'une telle solution intermédiaire au quatrième chapitre.

3.4. Critères de rightsizing

Nous définissons le "rightsizing" d'une application comme:

> la répartition des traitements d'une application sur les **plates-formes informatiques** les plus appropriées pour les exécuter, selon des considérations de **coût/efficacité** de ces environnements, de **politique** et d'**organisation** de l'entreprise.

Il s'agit donc d'affiner les solutions dégagées par les deux critères précédents (le type d'application et son intégration), et choisir les plates-formes informatiques les plus adéquates, c'est-à-dire les machines, logiciels de base (O.S., GUI., ...) et outils de middleware répondants à différents critères.

Par exemple, il faut tirer profit des capacités de traitement à faible coût des P.C., améliorer les représentations graphiques des interfaces utilisateurs en les déléguant à des stations de travail, prendre en charge la gestion documentaire par des mini-ordinateurs, etc ...

Précisons que notre définition de "rightsizing" est assez générale que pour s'adapter à la fois:
- au développement de nouveaux S.I.,
- et à la migration d'applications centralisées sur des plates-formes multi-machines.

Ainsi les principes que nous présenterons sous le terme "rightsizing" s'inscrivent également dans des démarches:

- de "***downsizing***" où, [CA91], "*Downsizing, the process of moving work traditionnally performed on mainframe to smaller systems, leads to reduced costs*";
([Gartn92] préfère parler de "downcosting");

- ou de "***upsizing***", impliquant la migration de certaines activités d'une application centralisée (par exemple sur P.C.) sur une machine de gamme supérieure (par exemple un midrange).

Pour résumer, [CA91], " *"Rightsizing" allows the I.S. user to choose from a wide range of systems to the processing tasks requireds, [...and] implies having the flexibility to make the best financial and operational decision regarding the processing system*". L'application ainsi développée se basera très certainement sur un ou plusieurs modèles de distribution (cooperative, client/serveur ou remote presentation procesing).

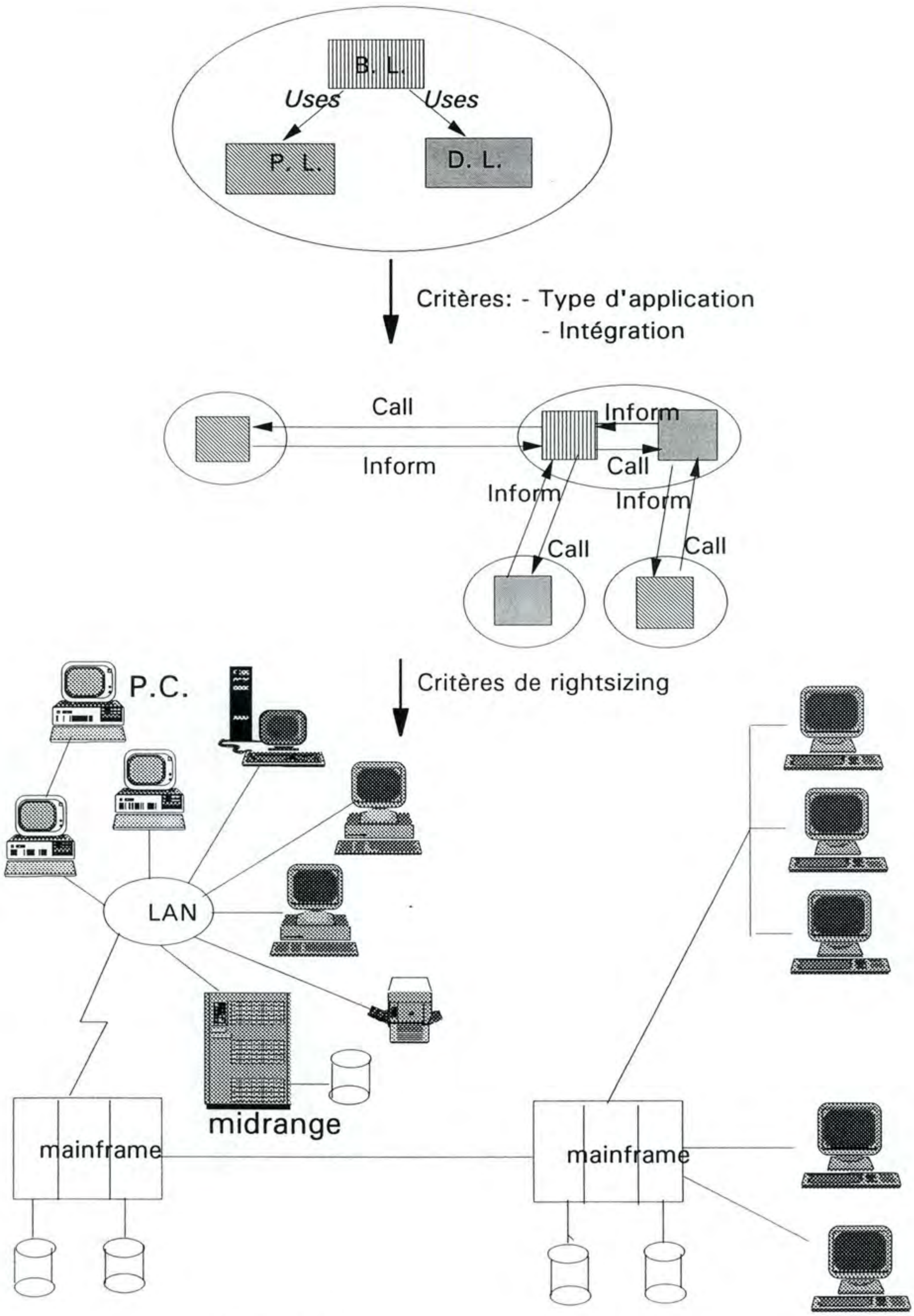

B. L.
Uses
Uses
P. L.
D. L.
Critères: - Type d'application
- Intégration
Call
Inform
Inform
Call
Inform
Inform
Call
Call
Critères de rightsizing
P.C.
LAN
midrange
mainframe
mainframe

Figure 3.10

Le rightsizing laisse donc au concepteur le choix des plate-formes informatiques les plus appropriées, au sein de toute une gamme d'environnements, pour exécuter les différentes tâches de son application. Ce choix se fait par évaluation d'un rapport coût/performance, de considérations politiques et organisationnelles.

Le poids de ces éléments, que nous appellerons "**critères de rightsizing**", dans le processus de sélection d'un environnement informatique doit être déterminé par les personnes impliquées dans le S.I., qui précisent elles-mêmes:
- un bon rapport *coût/efficacité* pour leur application,
- les implications pour son S.I. de la *politique* de l'entreprise et de son mode d'*organisation*.

Le rightsizing pouvant ainsi varier d'une entreprise à l'autre, voire même pour chaque application, nous nous contenterons d'énoncer quelques notions d'évaluation de ses critères.
D'ailleurs, la principale difficulté ne se situe pas nécessairement au niveau du choix d'un environnement. Elle se trouve plutôt au niveau de la découpe de l'application en sous-ensembles d'activités (ou "tâches") qui, ensuite, seront prises en charge par différentes plates-formes informatiques.

Par manque d'expérience en la matière , nous ne pouvons tirer de conclusions, et nous nous limiterons à une présentation succincte de quelques principes, très intuitifs et illustrés au quatrième chapitre.

3.4.1 Rapport coût/efficacité

Ce premier critère de rightzing, où le terme "rightcosting" serait d'ailleurs plus approprié, consiste à sélectionner les environnements matériels qui offrent le meilleur rapport coût/efficacité et qui sont à la "taille" ("sized") des tâches devant être exécutées.

A cette fin, le concepteur pourrait se baser sur un inventaire des différents environnements disponibles (ou qu'il compte acquérir), tel que par exemple:

- Pour les machines:

  - les **terminaux:** éléments purement passifs (écran et clavier); ils ne permettent pas d'envisager la moindre distribution,

  - les **X-terminaux,** éléments actifs prenant uniquement en charge des traitements de P.L., permettant des architectures de *remote presentation processing*. Les fonctionnalités limitées de ces machines réduisent significativement leur prix par rapport aux P.C. ou stations de travail.

- les **ordinateurs personnels**, mono ou multi-tâches (workstations); beaucoup de motivations de rightsizing lors du développement d'applications distribuées viennent de ces ordinateurs personnels où, [CA91], il s'agit de

  " *to take maximum advantage of the often underutilized power available in the growing number of workstations and desktop systems*".

- les **machines départementales** (midrange systems), ce sont des processeurs *spécialisés* à l'une ou l'autre activité éventuelle d'une entreprise telle que:
  - le calcul (batch ou interactif),
  - la gestion de fichiers,
  - la gestion documentaire,
  - le serveur d'impression,
  - le multimedia,
  - la représentation 3D,
  - le temps réel pour le contrôle de processus,
  - etc ...

- les **machines inter-départementales** (mainframe systems); pouvant être utilisés par l'ensemble de l'entreprise.

• Pour les **logiciels de base**:

- O.S., mono- ou multi-tâches
- GUI;
- couches logicielles de communication.

• Pour les **outils de middleware** (cfr 1.3.), nous avons conseillé au deuxième chapitre:

- pour les techniques de **Remote Procedure Call** :

  - des architectures de *coooperative processing*, étant donné qu'elles permettent le déclenchement d'activités d'agents distants; le seul inconvénient étant qu'elles ne gèrent pas de connexion réellement full-duplex,
  - et éventuellement des modèles *c/s orientés fonctions* et *remote presentation processing*,
  - et dans une moindre mesure des architectures *c/s orientées données*.

- Les mécanismes de **SQL Interactions** répondent essentiellement à des architectures *c/s orientées "données"* (et "*fonctions*", s'ils possèdent des mécanismes de *triggered procedures* ou de *callback routines*).

- Les techniques de **communication inter-processus** permettent du *cooperative processing* réellement full-duplex.

Sur base de cette classification, le concepteur peut décider des plates-formes les plus adéquates pour une ou plusieurs tâches de son application en fonction d'un rapport coût/efficacité, qui pourrait notamment comprendre:

- au niveau des **coûts**:
  - le prix et de l'amortissement des équipements,
  - les coûts de communication,
  - les frais de maintenance et d'exploitation,
  - le coût de l'environnement physique (locaux, installation,...),

- au niveau de l'efficacité des environnements:
  - la vitesse et le temps de transmission,
  - la fiabilité,
  - le contrôle d'erreur,
  - la tolérance de panne.

3.4.2 Considérations politiques et organisationnelles

Sans détailler ce sujet, signalons que le choix d'une plate-forme informatique se base également sur des considérations politiques et organisationnelles, propres à chaque entreprise ou secteur d'activités.

Ainsi au niveau **politique**, il faudrait investiguer:

- la *stratégie* (offenvive ou défensive) et la tactique de l'entreprise dans son secteur d'activités

  par exemple, si l'entreprise recherche essentiellement une rentabilité à court terme nécessaire à sa survie, elle ne va sans doute pas investir dans de nouvelles technologies mal maîtrisées par son personnel (informatique ou utilisarteur), rentable seulement à long terme, etc ...,

- la place et l'*impact de l'informatique* dans l'entreprise (ou son secteur d'activités),

- la recherche d'*homogénéité ou non* des équipements (multi-vendeurs, multi-O.S., multi-middleware, ...),

- la politique de *sécurité* (confidentialité), qui peut varier d'un secteur à l'autre,

  par exemple, les contraintes de sécurité et de fiabilité sont bien plus importantes dans le milieu bancaire, que le secteur de l'alimentation.

Au niveau **organisationnel**, bien évidemment en étroite relation avec les considérations politiques, il faudrait tenir compte:

- de l'*utilité de l'information* au sein de l'organisation, c'est à dire le niveau de gestion où elle intervient (déjà détaillé en 3.2) et le nombre d'utilisateurs;
  par exemple si, indépendamment des autres critères, une information ne sert qu'à un petit nombre d'utilisateurs on pourrait envisager son enregistrement sur un ordinateur personnel; sinon un midrange, voire un manframe si elle est utilisée par les différents composants de l'organisation.

- la *structure de l'organisation*: verticale ou horizontale, par produit ou flux de fabrication,...

- le *style de "management"* (centralisé, départementalisé, ...),

- la *transparence* d'organisation.

3.4.3 Conseils méthodologiques

Comme nous l'avons signalé, la principale difficulté du rightsizing est certainement de découper l'application en sous-ensembles d'activité pour, par la suite, pouvoir en évaluer l'adéquation à l'une ou l'autre plate-forme.

En cette matière, bien des méthodes pourraient être envisagées. Ne pouvant ni les juger, ni même les valider, nous nous contenterons d'énoncer les principes d'une démarche qui, intuitivement, nous semble bien adaptée.

Tout d'abord, il s'agirait de raisonnner sur l'ensemble des activités du S.I., où globalement on tenterait de déterminer les plates-formes informatiques les plus adéquates aux exigences de coût/efficacité, de la politique et de l'oganisation de l'entreprise.

Ensuite, le concepteur évaluerait ses critères de rightsizing réellement pour chaque tâche de son application, en prenant comme critère d'identification d'une "tâche" la notion de "phase" de [Bodar89], c'est à dire:

> " *une phase est un traitement possédant une unité spatio-temporelle d'exécution. [Cela ...] implique que la phase soit entièrement exécutée dans une cellule d'activités, c'est à dire un centre d'activité homogène dans le temps et l'espace, doté de ressources humaines et/ou matérielles et pourvu de règles de comportement nécessaires à son fonctionnement.* ".

Cela permettrait à l'informaticien de pouvoir envisager l'exécution de chaque phase de son application par des agents matériels distincts, sur base de critères de rightsizing. pparemment, toutefois sans pouvoir le démontrer, cette localisation spatio-temporelle des activités serait une bonne base à l'examen de différents environnements informatiques.

Pour chaque phase, on pourrait détailler ses catégories de traitements (P.L, B.L. et D.L.), ses échanges (volume et fréquence) d'information ("données" ou "événements") avec le reste de l'application, et en fonction des critères de rightsizing en déduire la plate-forme matérielle la plus appropriée.

Une telle analyse pourrait éventuellement mener à regrouper différentes phases sur un même équipement, ou répartir les traitements d'une même phase sur différentes machines.

Chapitre 4

Chapitre 4: Etude de cas

4.1. Introduction

Si pour mener, au travers des trois premiers chapitres, une analyse descriptive des applications distribuées nous sommes partis des aspects techniques pour remonter à la conception d'architectures de traitements; nous conseillons la **démarche** inverse, ***top-down***, pour effectivement développer des applications distribuées, à savoir:

- tout d'abord, identifier clairement l'application à développer, et la structurer en fonction de notre **architecture générale** des systèmes de gestion,
(où, pour rappel, on raisonne sur une machine abstraite, indépendamment de tout facteur de performance technique, cfr 2.2.2.),

- ensuite analyser les différents **critères de distribution** que nous avons relevés (cfr troisième chapitre), les préciser et les compléter en fonction de la situation donnée (particulièrement les critères de rightsizing, cfr 3.4.),

- ainsi, concevoir une **architecture** (éventuellement) **distribuée**, et choisir les **plates-formes informatiques**, qui semblent les plus appropriées.

Insistons sur le fait que, par nos critères de rightsizing, les conceptions de l'architecture distribuée et de l'infrastructure pour une application donnée, ne se font ni indépendamment, ni successivement, mais **"parallèlement"** avec des interférences fréquentes (***feedback***) entre les deux développements. Ainsi le choix d'une infrastructure peut affiner ou modifier l'architecture, et inversement.

Nous ne développerons pas davantage ces aspects méthodologiques, mais nous les illustrerons au travers d'une étude d'application distribuée, effectivement développée par le service informatique d'une industrie chimique.

4.2 Présentation du cas

Nous allons examiner une application de **G**estion du **M**atériel **F**erroviaire (**GMF**), pour une usine de fabrication de produits chimiques.

Ce système est destiné à gérer l'ensemble du matériel ferroviaire (wagons, trains de marchandises, containers, ...) permettant d'acheminer aux différents magasins de l'usine, les matières premières à stocker.

Les fonctionnalités de cette application sont essentiellement:

- la *gestion des annonces* de wagons,
  qui enregistre les télex, provenant d'une gare proche de l'usine, qui annoncent l'arrivée d'un wagon,

- la *gestion des entrées* de wagons dans l'usine,
  qui se charge de l'identification du wagon, de la vérification de son contenu, et l'enregistrement de sa date d'entrée,

- le *pesage*,
  qui doit, à l'entrée, donner le poids brut du wagon; et à la sortie de l'usine, donner la tare du wagon, et en déduire la quantité de matières livrées,

- le *dispatching*,
  qui doit déterminer la destination du wagon dans l'usine,

- la *gestion des sorties*,
  qui enregistre la date de sortie.

Dans notre **architecture générale** des systèmes de gestion (cfr 2.2.2.), cela se traduit:

- au niveau de la **Business Logic** (B.L.):

  - par plusieurs fils d'activités, correspondant à une ou plusieurs des fonctionnalités décrites précédemment, qui devraient pouvoir s'exécuter en parallèle (contrôle **concurrent**),

    par exemple, l'activité *pesage* en même temps que la *gestion des entrées/sorties* de wagons,

  - une **automatisation complète** du contrôle de l'application, étant donné que toutes les activités s'exécutent selon des procédures bien établies et parfaitement structurées,

    par exemple, un wagon ne sera soumis au *dispatching* qu'à la seule condition qu'il ait été identifié, lors de la *gestion des entrées*, et qu'il contienne le produit demandé,

- au niveau des traitements de **Presentation Logic** (P.L.),

  les concepteurs de GMF ont adopté un **dialogue conversationnel**, où les utilisateurs expriment les opérations qu'ils veulent exécuter via un langage de commandes et de sélection de menus,

- au niveau de la **Data Logic** (D.L.),

  il s'agit de fonctions classiques de **gestion opérationnelle**, se chargeant uniquement d'enchaîner différentes actions primitives (de mise à jour et consultation) sur la b.d.; telles que, par exemple, les fonctions *Enregistrement du poids net*, ou *Consulter la destination d'un wagon*.

4.3. Critères de distribution

4.3.1. Le type d'application

Tous les traitements que nous avons brièvement abordés caractérisent bien une application ***transactionnelle*** (cfr 3.2.1.).

Les activités des utilisateurs sont parfaitement formalisables, et entièrement sous le contrôle de la B.L. . Les fonctions de l'application se situent à un niveau de gestion opérationnelle, travaillant régulièrement sur de petites quantités de données, qui représentent des entités du monde réel telles qu'un wagon, un produit, une commande, ...

Le contrôle de l'application étant concurrent, une architecture de *cooperative processing*, où les activités (pesage, entrée/sortie, etc..) seraient réparties sur différentes machines, avec échanges d'information entre processus, était envisageable mais tel n'a pas été le cas.

Les développeurs du système ne disposant pas au moment de sa conception de l'infrastructure nécessaire, c'est-à-dire des outils de RPC ou de communication inter-processus pour des traitements s'exécutant sur un réseau de P.C., ils n'ont pas examiné cette solution.
L'application a été **centralisée** sur un "serveur" classique de fichiers (bien évidemment, ce terme n'est pas utilisé ici au sens de la relation client/serveur de traitements distribués) connecté à un réseau de P.C.

4.3.2. Critère d'intégration

Pour prendre en charge toutes les fonctionnalités attendues du système (cfr 4.1.), GMF doit disposer d'informations présentes dans un autre S.I. de l'entreprise, le système de GPAO (Gestion de Production Assistée par Ordinateur).

Ce GPAO, chargé du suivi et de la planification du processus de production, détient les informations suivantes nécessaires à GMF:

- les identifiants des wagons devant arriver à l'usine,
- les produits commandés,
- leur quantité,
- leur lieu de destination dans l'usine.

De même, GMF va détenir des informations intéressant le GPAO, à savoir le poids effectif des marchandises livrées.

Pour éviter la lourdeur des traitements manuels, et la redondance des activités, l'entreprise a décidé d'automatiser l' interface entre ces systèmes.

Solution idéale

Les échanges entre les deux applications sont, selon nos définitions (cfr 3.3), **événementiels** étant donné que :

- si les informations détenues par le GPAO ne sont pas directement transmises à GMF, elles lui sont irrécupérables,

  car la structuration des données du GPAO est très complexe, il s'agit d'une énorme b.d. hiérarchique (SGBD TOTAL de Cincom), où les informations nécessaires à GMF sont éclatées dans divers fichiers,

- les informations de GMF intéressant le GPAO, c'est-à-dire le poids des marchandises, doivent être transmises rapidement au GPAO, afin d'assurer un suivi correct du processus de production.

La solution idéale pour un tel type d'échange est donc un lien de **cooperative processing** entre les deux systèmes.

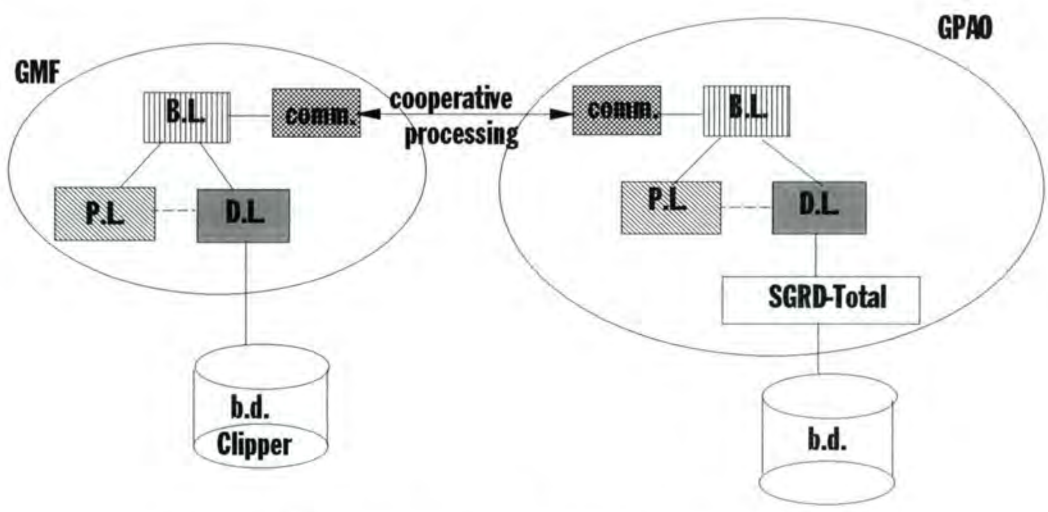


Figure 4.1: Solution idéale de cooperative processing

Chaque application peut ainsi transférer les événements au système requérant, et y déclencher le contrôle de nouvelles activités par la B.L., telles que:
- au niveau du GPAO, par exemple, une gestion des fournisseurs n'ayant pas respecté les quantités commandées,

- au niveau de GMF, l'enregistrement d'une série de nouvelles commandes décrétées par le GPAO, et de leur destination dans l'usine.

Solution implémentée

Comme nous l'avons déjà expliqué (cfr 3.2.2.), l'élément crucial d'une intégration par cooperative processing est la **synchronisation des activités** des applications en présence.

Etant donné que GMF et le GPAO sont des applications transactionnelles fréquemment utilisées (par l'ensemble de l'entreprise dans le cas du GPAO), il ne serait pas très efficace d'interrompre leurs activités à tout moment, pour échanger des informations.

Dès lors, la solution effectivement implémentée est un peu plus complexe. Nous allons montrer qu'elle passe par les couches Data Logic de chacun des systèmes, pour leur permettre des communications indirectes de B.L. à B.L., qui traitent les échanges d'événements à des instants bien déterminés.

• Du GPAO vers GMF

Dès que le GPAO détient une information intéressante pour GMF, il la stocke via des fonctions de D.L., dans une table relationnelle, servant de *mailbox*.

Tous les quarts d'heure GMF consulte la *mailbox* **via ses traitements de D.L. et une relation client/serveur** (orientée "donnée", cfr. 2.3.2.1.), pour y relever ses données, et les transférer dans sa propre b.d.

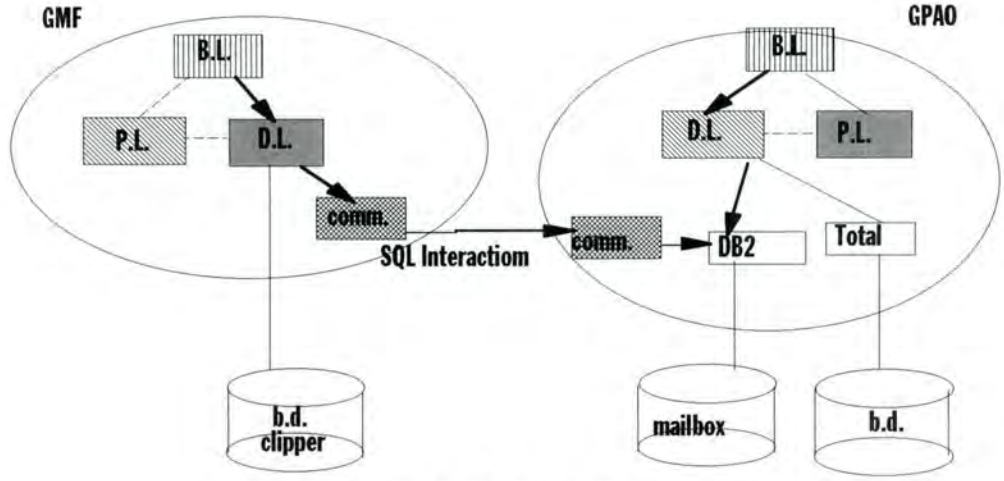


Figure 4.2: Echanges du GPAO vers GMF

Principe de la *mailbox*

L'utilité d'une *mailbox* (ou "*boîte aux lettres*") est de permettre l'échange de données entre deux applications indépendamment de leur environnement. La mailbox est une structure de données (par exemple, une table relationnelle) à laquelle, pour y accéder, est adjoint des API.

Principes d'utilisation:

- l'application qui détient l'information à échanger, la dépose dans la boîte (introduction d'une nouvelle donnée),
- l'application qui veut une information consulte la mailbox, transfère les données qui l'intéresse dans sa base de données, et les enlève de la boîte (suppression d'une ou plusieurs données).

Implémentation technique

La mailbox utilisée est une table DB2, et les applications y accèdent via des SQL API.

Dès que le GPAO possède une information intéressante pour GMF, il active une routine Cobol qui, par des SQL API, introduit un nouveau record dans la table DB2.
GMF consulte la mailbox via une relation client/serveur orientée "données", et une technique de **SQL Interactions** (cfr 1.3.2.2.), c'est-à-dire:
- parcourt tous les enregistrements de la table DB2,
- transfère dans sa base de données les records qui l'intéressent et suppriment ces enregistrements de la table.

• De GMF vers le GPAO

Tous les quarts d'heure également, la B.L. de GMF déclenche une fonction de mise à jour des poids des wagons dans la b.d du GPAO.
A cette fin, la fonction de D.L. de GMF émet une requête, **via une relation client/serveur**, de mise à jour des poids dans la b.d distante. Une fois cette action exécutée, le GPAO peut connaître les poids effectifs quand il le désire, simplement en consultant sa b.d. .

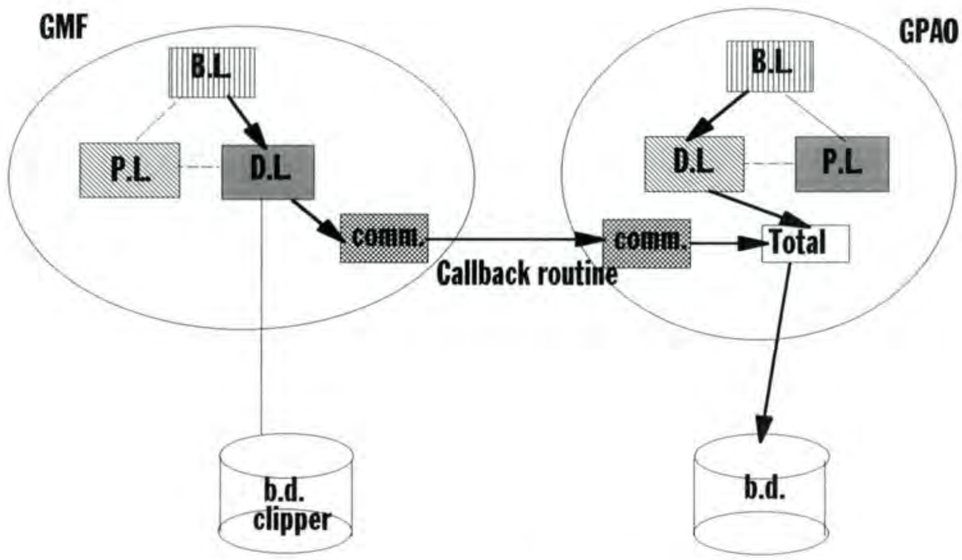


Figure 4.3:: Echanges de GMF vers GPAO

Implémentation technique

L'action de mise à jour est transmise au GPAO sous forme de requête SQL (*update*), via un mécanisme de ***callback routine*** (cfr 1.3.2.2.). Cette routine, stockée sur la même machine que le GPAO, se charge de traduire la requête SQL en format TOTAL, et effectue ainsi les mises à jour de la b.d. du GPAO.

Ainsi par exemple, GMF peut mettre à jour les données du GPAO via la requête:

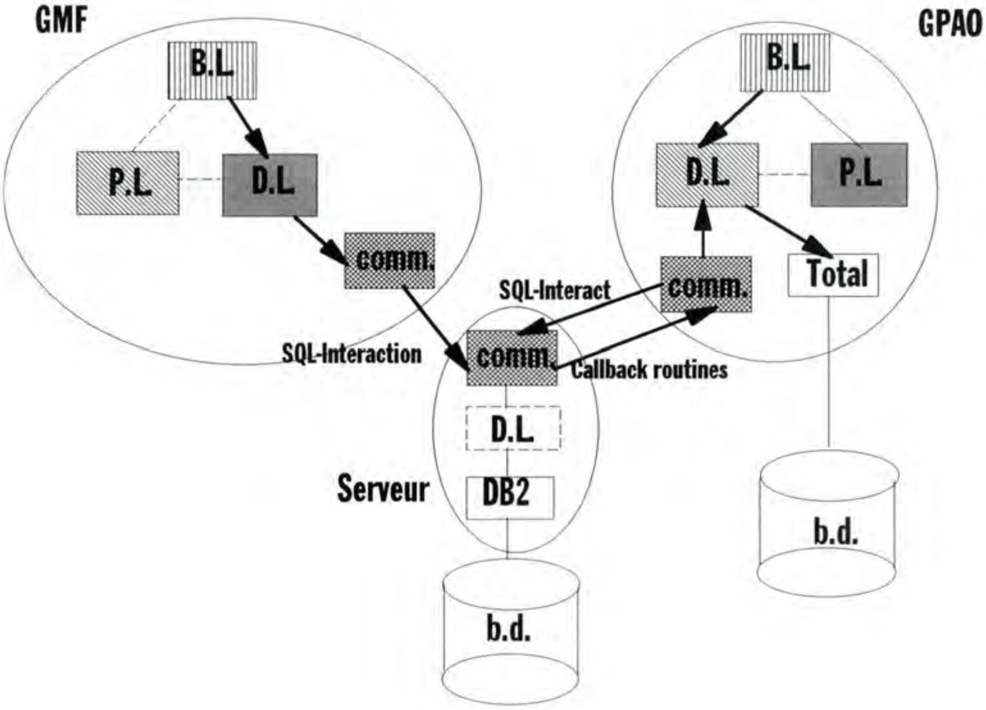
update WAGON set POIDS = YY where WAGON-ID = XX;

La callback routine, se trouvant sur le même agent que le GPAO, se charge de traduire cette requête SQL en exécution d'une ou plusieurs procédure(s) de mise à jour des fichiers non-relationnels du GPAO.

Solution plus efficace

Une solution plus efficace serait de remplacer le prinicipe de mailbox par une véritable b.d. relationnelle. Cette solution permettrait aux deux applications d'échanger leurs "événements" via une véritable b.d. commune, sur un agent serveur.

Cette b.d. serait prise en charge par un agent serveur à la fois des clients GMF et GPAO.
Ainsi GMF pourrait même éventuellement supprimer sa propre b.d. Clipper et ne travailler qu'avec le serveur. Quant au GPAO, il ne travaillerait plus avec une boîte au lettre, mais une mise à jour directe des données de GMF



4.3.3. Critères de rightsizing

Ces critères n'ont pas été pris en considération par les développeurs de l'application, du moins pas dans l'idée de distribuer les traitements de GMF.

Cependant, on pourrait très bien envisager une découpe en phases de GMF, on identifierait ainsi les centres d'activités spatio-temporels suivants:

- le poste d'entrée des wagons dans l'usine:
- le poste de pesage et dispatching,
- le poste de sortie.

Et pour chacune de ces cellules d'activités, on pourrait appliquer différents critères de rightsizing, que nous laisserons au libre choix du concepteur. Rappelons toutefois que cette approche d'identification par phase n'est qu'une solution parmi d'autres

4.4. Infrastructure

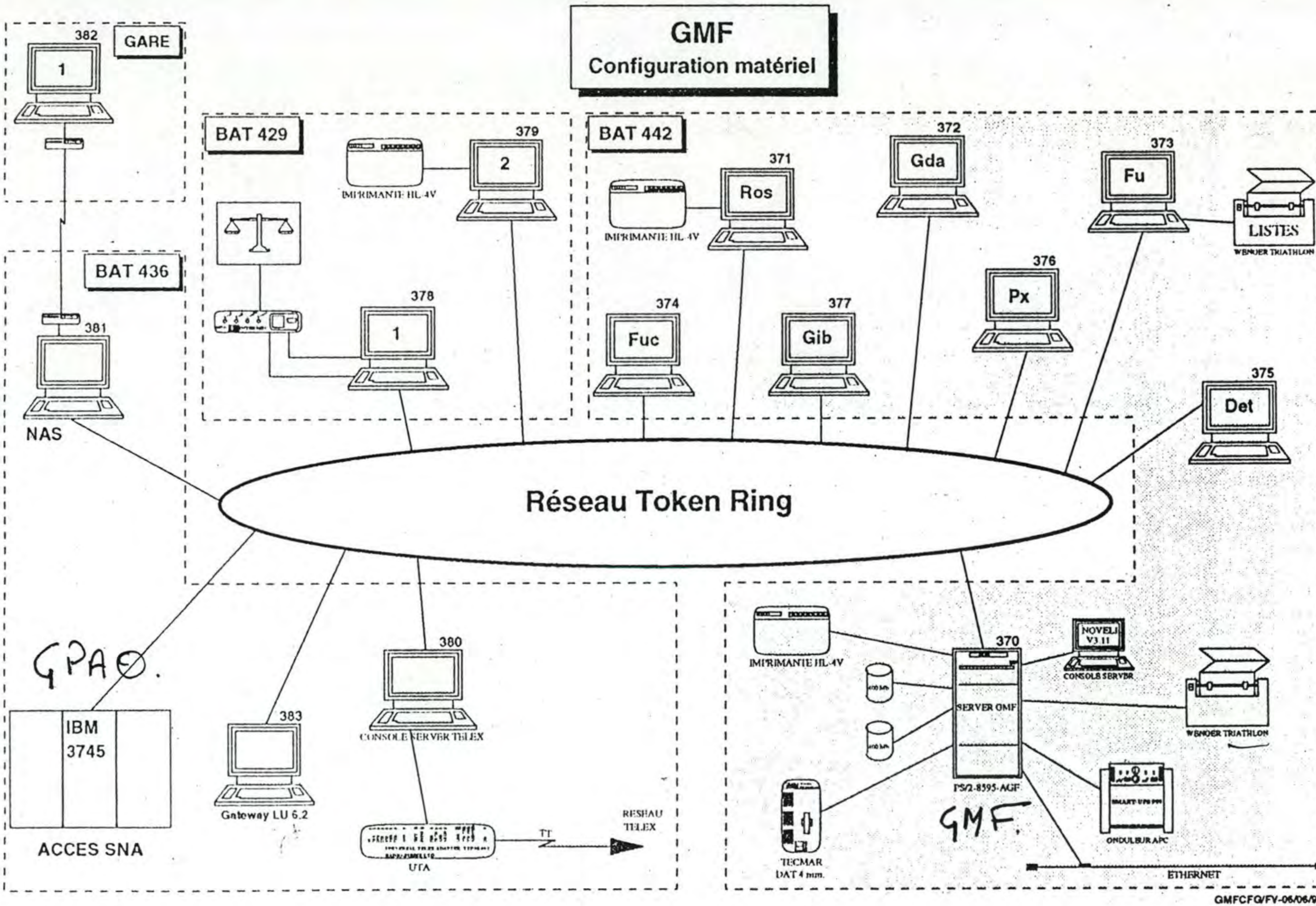
L'application GMF est entièrement centralisée sur un "serveur" de fichiers Clipper, connecté à un réseau de P.C.

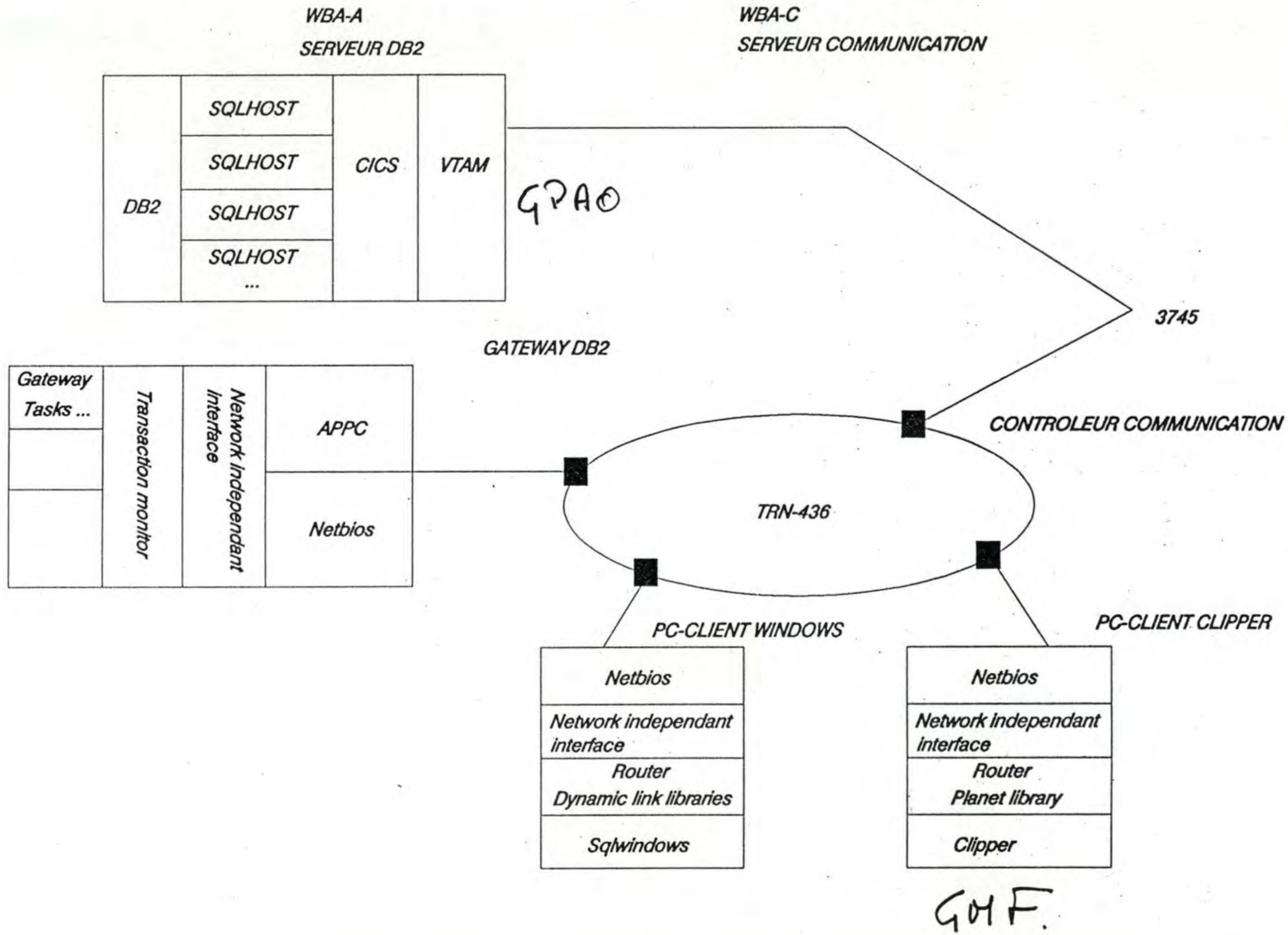
L'environnement de middleware choisi, utilisé donc uniquement pour l'échange d'informations avec le GPAO, est celui de Gupta Technologies, à savoir SQLNetwork et son prinicipe de callback routines (décrits au premier chapitre, cfr 1.3.2.2.).

Le GPAO, s'exécutant sur mainframe IBM, utilise également ces logiciels de SQLNetwork pour communiquer avec GMF.

Chapitre 4: Etude de cas

4.1. Introduction ... 90
4.2 Présentation du cas ... 91
4.3. Critères de distribution ... 93
4.3.1. Le type d'application ... 93
4.3.2. Critère d'intégration ... 93
Solution idéale ... 94
Solution implémentée ... 95
Solution plus efficace ... 98
4.3.3. Critères de rightsizing ... 99
4.4. Infrastructure ... 99


GMF
Configuration matériel
GARE
382
1
BAT 429
379
2
IMPRIMANTE HL-4V
378
1
BAT 436
381
NAS
BAT 442
371
Ros
IMPRIMANTE HL-4V
372
Gda
373
Fu
LISTES
WENGER TRIATHLON
374
Fuc
377
Gib
376
Px
375
Det
Réseau Token Ring
GPAO.
IBM
3745
ACCES SNA
383
Gateway LU 6.2
380
CONSOLE SERVER TELEX
UTA
TT
RESEAU
TELEX
IMPRIMANTE HL-4V
370
NOVELL
V3.11
CONSOLE SERVER
SERVER GMF
WENGER TRIATHLON
PS/2-8595-AGF
GMF.
ONDULEUR APC
TECMAR
DAT 4 mm.
ETHERNET
GMFCFG/FV-06/06/92



WBA-A
SERVEUR DB2
WBA-C
SERVEUR COMMUNICATION
DB2
SQLHOST
SQLHOST
SQLHOST
SQLHOST
...
CICS
VTAM
GPAO
3745
GATEWAY DB2
Gateway
Tasks ...
Transaction monitor
Network independant interface
APPC
Netbios
CONTROLEUR COMMUNICATION
TRN-436
PC-CLIENT WINDOWS
PC-CLIENT CLIPPER
Netbios
Network independant interface
Router
Dynamic link libraries
Sqlwindows
Netbios
Network independant interface
Router
Planet library
Clipper
GMF.

Conclusion

Conclusion

En réponse aux objectifs fixés au début de cette étude, nous avons montré que les applications distribuées ne se limitent pas à des questions technologiques ou de communication entre programmes.

Elles relèvent de différents aspects que nous avons répertoriés:

- au niveau de l'**infrastructure:** où toute application distribuée se caractérise par une couche logicielle de middleware, assurant une transparence de communication et de programmation;

- au niveau de l'**architecture**: où, sur base d'une proposition générale d'architecture des systèmes de gestion, le concepteur peut envisager différentes répartitions des traitements de son application. Nous avons ainsi définis trois modèles de distribution:

  - le client/serveur processing,
  - le cooperative processing,
  - le remote presentation processing.

- et nous avons relevé quelques **critères de distribution**, permettant d'assister le concepteur lors de l'élaboration de son architecture distribuée.

Même si la présentation de ces différents éléments peut encore faire l'objet d'études plus approfondies, la manière dont nous avons abordé le domaine des applications distribuées permet d'éliminer bon nombre d'ambiguïtés et confusions fréquemment relevées dans la littérature.

En l'absence actuelle d'une démarche systématique de développement de systèmes distribués, divers éléments de ce travail peuvent servir à l'élaboration d'une méthodologie de développement d'applications distribuées.

Pour réaliser notre étude, nous avons d'abord examiné les aspects techniques (infrastructure) avant l'étude des différentes architectures; nous ne pouvons conseiller une telle démarche au concepteur d'une application distribuée.

Nous avons laissé entrevoir, notamment par les critères de rightsizing, l'inefficacité d'une démarche inverse, à savoir l'étude de l'architecture indépendamment de toute considération technique.

Par conséquent, le concepteur d'une architecture distribuée doit prendre en compte les plates-formes informatiques, les environnements middleware disponibles et des critères de coût/efficacité.

Tous ces éléments doivent donc être développés "simultanément", affinés progressivement et s'élargir à des considérations politiques, économiques et organisationnelles. Il varie selon les applications et les secteurs d'activités.

Un des prolongements intéressants d notre travail, serait l'étude de l'intégration des différents aspects de distribution, dans une démarche classique de développement de S.I., à savoir:

- étude d'opportunité,
- analyse conceptuelle,
- méthodologie de développement de logiciels:
    - spécifications fonctionnelles,
    - conception globale et détaillée,
- implémentation.

Malgré les progrès importants réalisés ces dernières années, notamment dans la portabilité des environnements middleware, la conception d'une architecture d'application distribuée et le choix de son infrastructure reste un exercice difficile mais intéressant.

Bibliographie

Bibliographie

[Berso92] Berson A.,
Client/server architecture,
Mc Grawhill, 1992.

[Bloom92] Bloomer J.,
Power programming with RPC,
O'Reilly & Associates,1992.

[Bodar89] Bodart F., Pigneur Y.,
Conception assistée des systèmes d'information - Méthodes, Modèles, Outils,
Masson, 2ème ed., 1989.

[Bodar92] Bodart F.,
Systèmes d'Information d'Aide à la Décision,
FUNDP, notes de cours, 1992.

[Bodar93] Bodart F., Hennebert A.-M., Leheureux J.-M., Sacré B., Provot I., Vanderdonckt J.
Architecture Elements for Highly-Interactive Business-Oriented Applications,
FUNDP, TRIDENT Project, 1993.

[CA91] Computer Associates
Computing Architecture for the 90's,
Computer Association Inc., 2nd ed.,1990.

[Coulo88] Coulouris G.F., Dollimore J.,
Distributed systems, Concepts and Design,
Addison-Wesley, 1988.

[Coutt86] Couttaz J.,
Abstractions pour Interfaces Interactives,
T.S.I. vol. 5, n_4, 1986.

[Davies83] Davies D.W.,
Distibuted Systems-Architecture and Implementation,
Springer-Verlag, 1983.

[Duboi91] Dubois E.,
Méthodologie de développement de logiciels,
FUNDP, Notes de cours, 1991.

[Duboi93] Dubois E., Du Bois P., Petit M.
Elicitating and Formalising Requirements for C.I.M. Information Systems,
Proc. Fifth Conference on Advanced Information Systems Engineering,
Paris, June 1993.

[Gartn92] Gartner Group,
Planning Client/Server Architecture: The Next Two Years,
Research Note, February 1992.

[Gupta91] PC-to-DB2 Connectivity Software version 3.0,
Insatllation and Operation Manual,
Gupta Technologies, 1991.

[Hart89] Hartson H., Hix D.,
Human -Computer Interface Development: Concepts and Systems for its Management,
ACM Computing Surveys, vol. 21, March 1989.

[Hutch86] Hutchins E. I., Hollan J. D., Norman D.A.,
Direct-Manipulation Interfaces, in User-Centered System Design,
Norman D.A. and Draper S.W. eds,
Lawrence Erlbaum Assoc., Hillsdale, pp 87-124, 1986.

[Jacks83] Jackson M.,
System Development,
Prentice Hall, 1983.

[Osf91] - Introduction to OSF DCE,version 1.0
- Application development guide, OSF DCE version 1.0
- Administration development guide, OSF DCE version 1.0

[Parna72] Parnas,
On the Criteria to be used in Decomposing into Modules
Communication of the ACM, vol. 15, 1972.

[Sacre91] Sacre B., Provot I.
Proposition d'un langage de spécification de l'interface homme-machine d'une application de gestion hautement interactive,
FUNDP, rapport interne, Décembre 1991.

[Stall91] Stallings W.,
Data and Computer Communications,
Maxwell Macmillan International, 1991.

[Sybas93] Sybase,
Client/Server Architecture and Data Management,
Sybase Software, 1993.

[Tanen90] Tanenbaum A.,
Réseaux, architectures, protocoles, applications,
InterEditions,Paris,1990.

[White90] White C.J.,
Client/server Computing with DB2,
Database Associates,1990