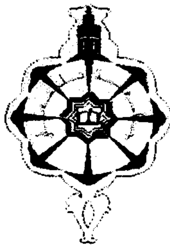
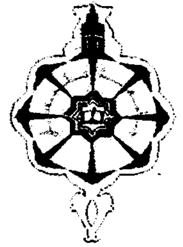
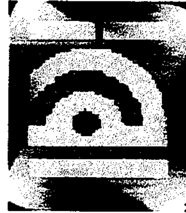
IN/003-33/01

République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique





Université Aboubakr Belkaid
Faculté des Sciences de l'Ingénieur
Département d'Informatique

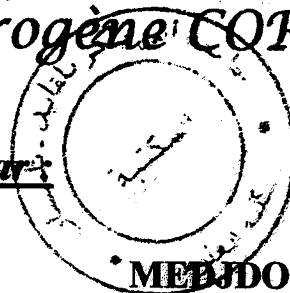


67 1998

Mémoire pour l'Obtention du Diplôme
D'Ingénieur d'État en Informatique

Thème :

Développement d'une application répartie par objet hétérogène CORBA



Présenté par :

TABTI Redouane **MEDJDOUB Abdelwaheb**

Soutenu devant le jury :

Mr	**ABDERAHIME Mohamed Amine**	**Président**
Mme	**BENALLEL Mounira**	**Examinatrice**
Mr	**BENAMAR Abdelkarim**	**Encadreur**

Promotion 2006/2007

RESUME

Les systèmes distribués occupent une place de plus en plus importante dans l'informatique actuelle. La distribution rend les systèmes plus complexes à concevoir et à gérer car ces derniers doivent tenir compte de nouveaux types de problèmes, tels que les pannes partielles ou les défaillances du réseau. Afin de répondre à la demande grandissante des technologies distribuées, plusieurs environnements de type "middleware" sont apparus au cours des dernières années.

L'object Management Croup propose une solution globale, à l'aide de technologies orientées objet, pour la construction d'applications distribuées résolvant les problèmes de communication, d'hétérogénéité, d'intégration et d'interopérabilité.

Au cœur de cette proposition on trouve le bus à objets répartis CORBA *(Common Object Request Broker Architecture).* Ce bus fournit les mécanismes de base pour la communication entre les objets distribués et hétérogènes.

Cet ouvrage présente une vision globale des travaux de l'OMG : le modèle orienté objet, l'architecture de gestion des objets OMA, le bus CORBA, le langage de définition des interfaces OMG IDL, les services objet communs, les utilitaires communs et les interfaces de domaine.

SUMMARY

Distributed computing is one of the major trends in the computer industry. As systems become more distributed, they also become more complex and have to deal with new kinds of problems, such as partial crashes and link failures. To answer the growing demand in distributed technologies, several middleware environments have emerged during the last few years.

Object Management Croup proposes a total solution, using object-oriented technologies, for the construction of distributed applications solving the problems of communication, heterogeneity, integration and interoperability.

Inside of this proposition the bus of object distributed CORBA (Common Object Request Broker Architecture) is found. This bus provides the basic mechanisms for the communication between the distributed and heterogeneous objects.

This work presents a global vision of OMG work: the object-oriented model, the Object Management Architecture OMA, bus CORBA, the Interfaces Definition Language OMG IDL, the common object services, the common facilities and the domain interfaces.

SOMMAIRE

INTRODUCTION GENERALE 1
PROBLEMATIQUE 3
1. ORIGINES ET HISTORIQUE 3
1.1 Avant 1980 3
1.2 Les années 80 3
2. CARACTERISTIQUE DES ARCHITECTURES CENTRALISÉES 4
3. AVANTAGES ET INCONVÉNIENTS 4
SOLUTION 4

CHAPITRE I L'ARCHITECTURE CLIENT-SERVEUR

INTRODUCTION 5
1. CARACTÉRISTIQUES DE L'APPROCHE *C/S* 5
1.1 Une répartition hiérarchique des fonctions 5
1.2 Définition 5
1.3 Une grande diversité d'outils 5
2. LES AVANTAGES DE *C/S* 6
2.1 Les contraintes de l'entreprise 6
2.2 Mieux maîtriser le système d'information 6
2.3 Prendre en compte les évolutions des technologies 6
2.4 Réduire les coûts 7
3. LES TROIS NIVEAUX D'ABSTRACTION D'UNE APPLICATION 7
4. L'ARCHITECTURE UN TIERS 8
4.1. Présentation 8
4.2. Les solutions sur site central (mainframe) 8
4.3. Les applications un tiers déployées 9
4.4. Limites du model client-serveur un tiers 9
5. LE SCHEMA DU GERTNER GROUP 10
5.1. Présentation distribuée 10
5.2. Présentation distante 10
5.3. Gestion distante des données 11
5.4. Traitement distribué 11

5.5. Données et traitements distribués 11
6. L'ARCHITECTURE DEUX TIERS 11
6.1. Présentation 11
6.2. Le dialogue client-serveur 12
6.3. Le middleware 13
6.4. Les services rendus 13
6.5. Limites du model client-serveur deux tiers : Le client lourd 14
LES ARCHITECTURES DISTRIBUÉES 15
7. L'ARCHITECTURE TROIS TIERS 15
7.1. Présentation 15
7.1. Les premières tentatives 15
7.2. La révolution Internet 16
7.2.1. Introduction 16
7.2.2. Les standards d'Internet 16
7.2.3. Adaptation à l'entreprise (Intranet) 16
7.3. Le client léger 17
7.4. Répartition des traitements 17
7.5. Exemples de clients légers 17
7.6. Limites du model client-serveur trois tiers 17
8. LES ARCHITECTURES N-TIERS 18
8.1. Présentation 18
8.2. L'approche objet 19
8.3. Les objets métier 19
8.3.1. Les Java Beans 19
8.2.2. Les EJB (Entreprise Java Beans) 20
8.3. La communication entre objets 20
8.3.1 Principe 20
8.3.2 Le modèle CORBA 20
8.3.3 L'appel de procédure distante (RMI) 21
CONCLUSION 21

CHAPITRE II LE MODEL CORBA

LE MODÈLE CORBA 22
INTRODUCTION GENÉRALE 22

1. INTRODUCTION 22
2. PRESENTATION DE CORBA 22
2.1. L'architecture CORBA et les objets 23
2.2. Notion d'objet dans CORBA 23
3. L'ARCHITECTURE CORBA 24
3.1. L'OMG 24
3.2. L'OMA 24
3.3. Les composants de l'architecture CORBA 25
3.3.1. L'ORB 25
3.3.2. Objet applicatif 25
3.3.3. Utilitaires communs 26
3.3.4. Interface de domaine 26
3.3.5. Service objet 26
4. LES COMPOSANTES DE L'ORB 27
4.1. Langage OMG/IDL 28
4.2. Le langage de définition d'interface 28
4.2.1. Modules stub et skeleton 29
4.2.2. Adaptateur d'objets 29
4.2.3. Identification des objets (IOR) 31
4.2.4. Protocole IIOP 32
4.2.5. Les référentiels 32
5. EXEMPLE : Salut en Java 33
CONCLUSION 34

CHAPITRE III LE LANGAGE IDL

INTRODUCTION 35
1. UN LANGAGE DE DESCRIPTION 35
1.1 Le contrat IDL 35
1.2 La grammaire du langage IDL 36
1.3 Le précompilateur IDL 36
1.4 Les commentaires 37
1.5 Les espaces de définition 37
1.6 Les pragmas 37
2. UN LANGAGE FORTEMENT TYPÉ 38

2.1. Les types simples ... 39
2.2 Les constantes ... 39
2.3 Les définitions de type ... 40
2.4 Les types complexes ... 40
2.4.1 Les énumérations ... 40
2.4.2 Les structures ... 41
2.4.3 Les unions ... 41
2.4.4 Les tableaux ... 42
2.4.5 Les séquences ... 42
2.5 Les métatypes de données ... 43
3. LES INTERFACES ... 43
4. LES OPERATIONS ... 44
4.1 Les exceptions ... 45
5. LES ATTRIBUETS ... 45
6. L'HÉRITAGE ... 46
7. LA PROJECTION DU LANGAGE IDL ... 46
7.1 Le mécanisme de projection ... 46
7.2 Les règles des projections ... 47
CONCLUSION ... 48

CHAPITRE IV LES SERVICES CORBA

LES SERVICES CORBA ... 49
INTRODUCTION ... 49
1. SERVICE DE NOMS ... 49
1.1 Présentation ... 50
2. LE SERVICE VENDEUR ... 51
2.1 Présentation ... 51
3. LE SERVICE D'EVENEMENTS ... 52
3.1. Présentation ... 52
4. SERVICE DE SÉCURITÉ ... 54
4.1. Présentation ... 54
5. SERVICE DE TRANSACTION ... 55
6. SERVICCE CYCLE DE VIE ... 55
7. SERVICE RELATIONS ... 56

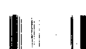
8. SERVICE EXTERNALISATION ... 56
9. SERVICE DE PERSISTANCE ... 56
10. SERVICE DE PROPRIÉTÉS ... 56
11. SERVICE DE TEMPS ... 56
12. SERVICE DE COLLECTION ... 56
14. SERVICE DE CONTROLRE DE LICENCE ... 57
CONCLUSION ... 57

CHAPITRE V L'IMPLEMENTATION

INTRODUCTION ... 58
1. CONSTRUCTION UNE APPLICATION REPARTIE AVEC CORBA ... 58
1.1 Choisir un langage d'implantation ... 58
1.2 Le Serveur d'application utilisé : ... 59
1.3 Implémentation de CORBA utilisée : ... 59
2. LES ETAPES DE DEVELLOPEMENT D'UNE APPLICATION CORBA ... 60
3. DESCRIPTION DE NOTRE APLICATION ... 60
CONCLUSION ... 65

CONCLUSION GENERALE ... 67

REFERENCES BIBLIOGRAPHIQUES



LISTE DES FIGURES

Figure 1 : *Exemple d'architecture centralisée* 3

Figure I.1: *Les trois niveaux d'une application informatique* 8
Figure I.2 : *Architecture d'une application sur site central* 9
Figure I.3 : *Les différents types de client-serveur selon la classification du Gartner Group* 10
Figure I.4 : *la base du dialogue, un besoin du client* 12
Figure I.5 : *Accès aux données en mode deux tiers* 12
Figure I.6 : *Middleware* 13
Figure I.7 : *Le middleware par rapport au modèle OSI (Open Systems Interconnection)* 14
Figure I.8 : *Le découpage d'une application en pavés fonctionnels indépendants* 17
Figure I.9 : *L'architecture distribuée* 19
Figure I.10 : *Evolution des technologies utilisées pour la mise en œuvre d'applications Distribuée* 19

Figure II.1 : *Figure générale de corba* 23
Figure II.2 : *l'ensemble de l' OMA* 25
Figure II.3 : *les services offerts par la norme CORBA* 26
Figure II.4 : *noyau CORBA* 27
Figure II.5 : *le dialogue CORBA* 31

Figure III.1 : Le contrat *IDL entre un fournisseur et un client d'objets* CORBA 36
Figure III.2 : *les types de données de l'OMG-IDL* 38
Figure III.3 : *La communication entre des applications hétérogènes* 47

Figure IV.1 *: les services CORBA* 49
Figure IV.2 : *Scénario d'utilisation du service de noms* 50
Figure IV.3 *: Le fonctionnement du service Vendeur* 52
Figure IV.4 *: Scénario d'utilisation du service d'événements* 53
Figure IV.5 *: le modèle push* 54
Figure IV.6 *: le modèle pull* 54

Figure V.1 *: Relation entre client, serveur et Smart Agent* 59
Figure V.2 *: Page d'accueil* 61
Figure V.3 *: Onglet gérant* 61

Figure V.4 : *L'ajout d'un livre* 62
Figure V.5 : *Recherche ou suppression d'un livre* 62
Figure V.6 : *consultation générale ou individuelle* 63
Figure V.7 : *Liste des livres* 63
Figure V.8 : *Aide* 64
Figure V.9 : *Aide* 64
Figure V.10 : *Message d'exception* 65
Figure V.11 : *Trace des invocations* 65

INTRODUCTION GENERALE

INTRODUCTION GENERALE

Lors de ces dernières années, l'informatique à connu de grands progrès technologiques: augmentation de la puissance tirs micro-processeurs, forte diminution (le prix des micro-ordinateurs), déploiement de réseaux rapides à large échelle (régionaux, nationaux) et finalement fédération de ces réseaux dans des structures mondiales de type Internet.

Ces avancées technologiques et la démocratisation de l'outil informatique ont permis l'éclosion d'applications distribuées comme, par exemple, la messagerie électronique ou le WWW (World Wide Web). Ce dernier fournit, une grande variété de ressources - des informations et - des applications - accédées par de nombreux utilisateurs répartis sur un ensemble de machines hétérogènes. La croissance des réseaux et des applications distribuées est liée à l'augmentation des besoins de partage d'informations et de ressources entre les organisations du monde entier.

Les applications distribuées ont besoin, pour leur exécution, de plates-formes réparties. Ces plates-formes nécessitent la définition de normes prenant en compte la communication, l'hétérogénéité, l'intégration et l'interopérabilité des applications distribuées. CORBA s'inscrit, naturellement dans ce processus: elle fournit une plate-forme d'exécution à base de technologies orientées objet pour l'intégration et l'interopérabilité d'applications distribuées et hétérogènes.

Après une introduction sur les architectures centralisées, dans laquelle on présente une vision générale de ces architectures et ses limites, on passe ensuite au chapitre 1.

Le chapitre 1 détaille l'architecture C/S distribuées et l'apparition des objets distribués qui est L'évolution la plus récente dans ce domaine.

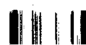
Le chapitre 2 présente le premier travail réalisé par l'OMG qui est la définition de l'OMA (*Object Management Architecture*), et ensuit le bus CORBA.

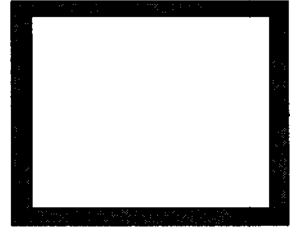
Le chapitre 3 illustre le langage IDL (Interface Definition Language), qui est en premier lieu un langage de description de services orientés objet pour l'environnement CORBA. Des exemples illustrent tout au long de ce chapitre les différentes fonctionnalités offertes par le langage IDL.

Le chapitre 4 présente les différents services de base qui ont été normalisés par OMG. Après avoir défini le cœur d'un système CORBA (l'ORB), l'OMG a spécifié un ensemble de services que l'on trouve dans bon nombre d'applications distribuées.

Le chapitre 5 qui est le dernier met en exergue notre mie en œuvre pratique des concepts théoriques sur un fragment d'application de gestion bibliothèque.

Enfin, le tout sera suivi d'une conclusion générale qui fera la synthèse de notre travail en ouvrant de nouvelle perspective.





PROBLEMATIQUE

PROBLEMATIQUE

1. ORIGINES ET HISTORIQUE

Dans un monde où la course à la productivité conduit les technologies à évoluer de plus en plus vite, le client-serveur s'est taillé une part de choix depuis le début des années 1990. En effet, il faut pouvoir disposer de systèmes d'information évolutifs permettant une coopération fructueuse entre les différentes entités de l'entreprise. Les systèmes des années 70 et 80 ne répondaient pas à ces exigences.

1.1 Avant 1980

Les architectures étaient centralisées autour de calculateurs centraux *(mainframe).* Les terminaux étaient passifs à interfaces caractères ou pages. Les applications étaient développées souvent en Cobol, parfois en PL1 autour de fichiers ou de grandes bases de données réseaux ou hiérarchiques. La productivité des développeurs restait faible. La maintenance des applications était des plus difficiles. Les utilisateurs restaient prisonniers de systèmes propriétaires. La figure1 illustre les architectures centralisées.

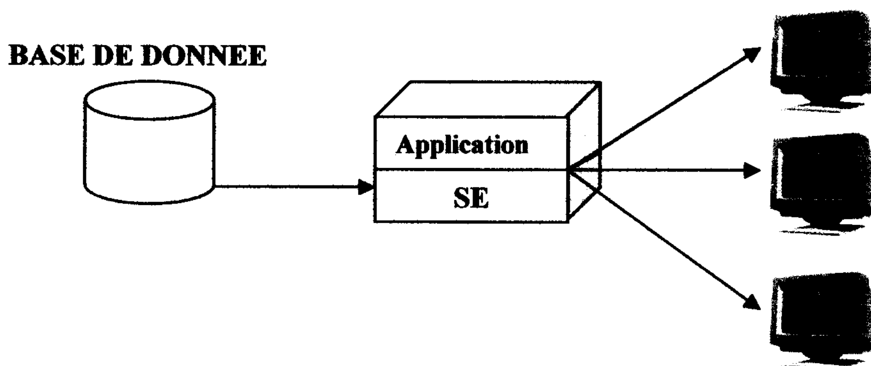


Figure 1 : *Exemple d'architecture centralisée* **[1].**

1.2 Les années 80

Les années 80 ont connu le développement du transactionnel et des bases de données. Les systèmes ont commencé à migrer depuis des systèmes propriétaires vers des systèmes plus ouverts type Unix. Les bases de données relationnelles ont vu le jour accompagnées de langages de développement construits autour des données. SQL s'est imposé comme la norme d'accès aux données.

2. CARACTERISTIQUE DES ARCHITECTURES CENTRALISÉES

- Système propriétaire, provenant d'un seul OS (Operating Systeme);
- Applications monolithiques ;
- Traitements au niveau du Serveur Central;
- Gros systèmes mêlant interfaces, règles métiers (logique applicative) et stockages de données;
- Terminaux passifs.

3. AVANTAGES ET INCONVÉNIENTS

Avantages :

- Facilité d'administration;
- Performance : la centralisation de la puissance sur une seule et même machine permet une utilisation optimale des ressources et se prête très bien aux traitements par lots (en *batch).*
- Sécurité et fiabilité.

Inconvénients:

- Interface utilisateur en mode caractère peu convivial ;
- Systèmes non ouverts vers d'autres, dépendance d'un fabricant particulier ;
- Maintenance logicielle peu aisée et délicate, et donc un coût très élevé pour la maintenance logicielle ;
- Problèmes de compatibilité: les Hosts , n'étaient parfois pas compatibles dans la gamme d'un même constructeur. Le simple passage d'un modèle à un autre au sein du même constructeur impliquait souvent une réécriture des applications.

SOLUTION

Les réseaux occupent depuis les années 90 une place centrale dans l'entreprise. Les vitesses de calcul des micros deviennent impressionnantes. Le graphique est partout au niveau des interfaces. Le besoin de partage des données est essentiel aussi bien pour l'accès transactionnel caractérisé par des mises à jour rapides en temps réel que pour l'accès décisionnel marqué par le besoin de requêtes complexes sur de gros volumes de données.

Aprés l'apparition des architectures centralisées vient l'architecture distribuée, qui sera l'objet du chapitre suivant, pour répondre aux limites et inconvénients de première architecture.

CHAPITRE 1

L'ARCHITECTURE CLIENT-SERVEUR

L'ARCHITECTURE CLIENT-SERVEUR

INTRODUCTION

L'informatique répartie est un des enjeux principaux de l'informatique actuelle et à venir. De l'architecture de terminaux centralisée qui servait encore de modèle il y a peu de temps, nous somme passé à l'architecture client/serveur. L'information est maintenant répartie entre différents serveurs du système et non plus centralisée sur un seul et même serveur : elle est répartie et distribuée aux clients de façon transparente.

1. CARACTÉRISTIQUES DE L'APPROCHE *C/S*

Ou'est-ce-que le client-serveur? Voilà une question difficile car le mot est souvent très galvaudé. C'est de fait un modèle d'architecture à la mode et assez large pour recouvrir des réalités distinctes, comme nous allons le voir dans la suite.

1.1 Une répartition hiérarchique des fonctions

L'architecture client-serveur s'articule en général autour d'un réseau. Deux types d'ordinateurs sont interconnectés au réseau. Le serveur assure la gestion des données partagées entre les utilisateurs. Le client gère L'interface graphique de la station de travail personnelle d'un utilisateur. Les deux communiquent par des protocoles plus ou moins standardisés.

1.2 Définition

En fait, l'architecture client-serveur est plus large. Un réseau n'est pas toujours nécessaire. Il est possible de la réaliser sur une même machine en dégageant deux processus, l'un - **le client** - qui envoie des requêtes à l'autre - **le serveur-**, ce dernier traitant les requêtes et renvoyant des réponses.

1.3 Une grande diversité d'outils

Le type de client-serveur que nous étudions est celui aujourd'hui mis en œuvre pour réaliser les systèmes d'information des entreprises. Il comporte les composants suivants:

. **Un système ouvert** plus ou moins basé sur des standards publiés ou de fait pour réaliser les fonctions de base du serveur.

. **Un SGBD** s'exécutant sur le serveur.

. **Des stations de travail personnelles** avec interface graphique connectées au réseau.

. **Des outils de développement d'applications** construits autour d'une approche objet.

. **Des logiciels de transport de requêtes et réponses.**

. **Des outils de conception, de déploiement et de maintenance.**

En clair, et c'est sans doute une force mais aussi une source de complexité.

2. LES AVANTAGES DE *C/S*

Outre l'avantage de pouvoir fédérer une panoplie de vendeurs d'outils, le client-serveur répond aux vrais besoins actuels des entreprises.

2.1 Les contraintes de l'entreprise

Les entreprises sont soumises à des contraintes de plus en plus fortes, aussi bien du monde extérieur que de l'intérieur de l'entreprise.

- Les **contraintes externes** sont imposées par les clients de plus en plus exigeants, la régulation de plus en plus complexe, la compétition de plus en plus dure qui conduit à réduire le temps de passage d'un produit de la conception à la vente.
- Les **contraintes internes** se traduisent par des pressions sur les budgets, la difficulté à absorber les technologies nouvelles, et un manque général de temps et de moyens.

2.2 Mieux maîtriser le système d'information

Une approche de solution aux problèmes mentionnés passe par une meilleure organisation du système d'information qui doit devenir plus intégré, mais aussi plus évolutif. Ceci nécessite tout d'abord l'adoption de systèmes ouverts, obéissant à des standards permettant le choix d'un grand nombre de produits sur le marché. Il faut donc à tout prix éviter les solutions s'enfermant sur un constructeur ou des développements « maison » ignorant les standards, solutions propriétaires qui risquent à terme de limiter les choix et la concurrence des fournisseurs, et donc d'augmenter les coûts.

2.3 Prendre en compte les évolutions des technologies

Au-delà de la maîtrise du système d'information qui passe par un serveur relationnel éventuellement étendu à l'objet, le client-serveur apporte une modularité des composants matériels et logiciels. Ceci permet d'intégrer plus facilement les évolutions technologiques.

2.4 Réduire les coûts

Le client-serveur améliore l'ouverture du système d'information et la productivité des développeurs. Il permet un déploiement plus rapide sur des architectures réparties hétérogènes. Oui mais à quel prix? Les coûts du client-serveur sont discutés. Par rapport à une architecture centralisée autour de terminaux passifs, des surcoûts sont à prévoir:

- poste de travail local type PC ;
- réseau local;
- formation des développeurs ;
- techniciens de maintenance réseaux et PC.

3. LES TROIS NIVEAUX D'ABSTRACTION D'UNE APPLICATION

En règle générale, une application informatique peut être découpée en trois niveaux d'abstraction Distincts :

- **La couche de présentation**, encore appelée IHM, permet l'interaction de l'application Avec l'utilisateur. Cette couche gère les saisies au clavier, à la souris et la présentation des Informations à l'écran. Dans la mesure du possible, elle doit être conviviale et ergonomique.

- **La logique applicative, les traitements**, décrivant les travaux à réaliser par l'application. Ils Peuvent être découpés en deux familles :

  - ✓ **Les traitements locaux**, regroupant les contrôles effectués au niveau du dialogue avec l'IHM, visant essentiellement le contrôle et l'aide à la saisie.

  - ✓ **Les traitements globaux**, constituant l'application elle-même. Cette couche, appelée *Business Logic* ou couche métier, contient les règles internes qui régissent une entreprise donnée.

- **Les données**, ou plus exactement l'accès aux données, regroupant l'ensemble des Mécanismes permettant la gestion des informations stockées par l'application.

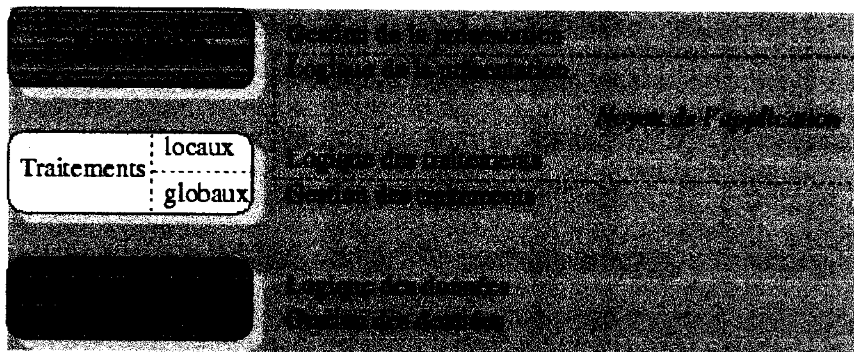


Figure 1: *Les trois niveaux d'une application informatique* **[4].**

Ces trois niveaux peuvent être imbriqués ou répartis de différentes manières entre plusieurs machines physiques.

Le noyau de l'application est composé de la logique de l'affichage et la logique des traitements. Le Découpage et la répartition de ce noyau permettent de distinguer les architectures applicatives Suivantes :

L'architecture 1-tiers, L'architecture 2-tiers, L'architecture 3-tiers, et Les architectures n-tiers.

4. L'ARCHITECTURE UN TIERS

4.1. Présentation

Dans une application un tiers, les trois couches applicatives sont intimement liées et s'exécutent sur le même ordinateur. On ne parle pas ici d'architecture client serveur, mais d'informatique centralisée .Dans un contexte Multi-Utilisateurs, on peut rencontrer deux types d'architecture mettant en oeuvre des applications un tiers :

- Des applications sur site central.
- Des applications réparties sur des machines indépendantes communiquant par partage de fichiers.

4.2. Les solutions sur site central (mainframe)

Historiquement, les applications sur site central furent les premières à proposer un accès multi utilisateurs. Dans ce contexte, les utilisateurs se connectent aux applications exécutées par le serveur central (le *mainframe*) à l'aide de terminaux passifs se comportant en esclaves. C'est le serveur central qui prend en charge l'intégralité des traitements, y compris l'affichage qui est simplement déporté sur des terminaux passifs. La figure 2 illustre l'architecture d'une application sur cite central.

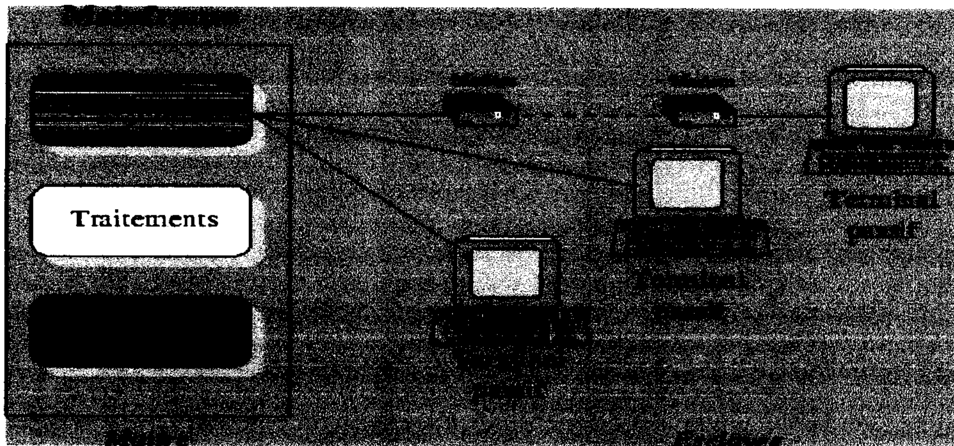


Figure 2 : *Architecture d'une application sur site central* **[5].**

4.3. Les applications un tiers déployées

Avec l'arrivée dans l'entreprise des premiers PC en réseau, il est devenu possible de déployer une application un tiers sur plusieurs ordinateurs indépendants. Ce type de programme est simple à concevoir et à mettre en oeuvre. L'ergonomie des applications mises en oeuvre, basée sur celle des outils bureautiques, est très riche. Ce type d'application peut être très satisfaisant pour répondre aux besoins.

Dans ce contexte, plusieurs utilisateurs se partagent des fichiers de données stockés sur un serveur commun. Le moteur de base de données est exécuté indépendamment sur chaque poste client. La gestion des conflits d'accès aux données doit être prise en charge par chaque programme de façon indépendante, ce qui n'est pas toujours évident.

Ce type de solution est donc à réserver à des applications non critiques exploitées par de petits groupes de travail (une dizaine de personnes au maximum).

4.4. Limites du model client-serveur un tiers

On le voit, les applications sur site central souffrent d'une interface utilisateur en mode caractères et la cohabitation d'applications micro exploitant des données communes n'est pas fiable au delà d'un certain nombre d'utilisateurs.

Il a donc fallu trouver une solution conciliant les avantages des deux premières :

- ✓ La fiabilité des solutions sur site central, qui gèrent les données de façon centralisée.
- ✓ L'interface utilisateur moderne des applications sur micro-ordinateurs.

Pour obtenir cette synthèse, il a fallu scinder les applications en plusieurs parties distinctes et Coopérantes:

- ✓ Gestion centralisée des données.
- ✓ Gestion locale de l'interface utilisateur.

Ainsi est né le concept du client-serveur.

5. LE SCHEMA DU GERTNER GROUP

Le Gartner Group, un cabinet de consultants américain, a publié un schéma des différents types de Client-serveur existants.

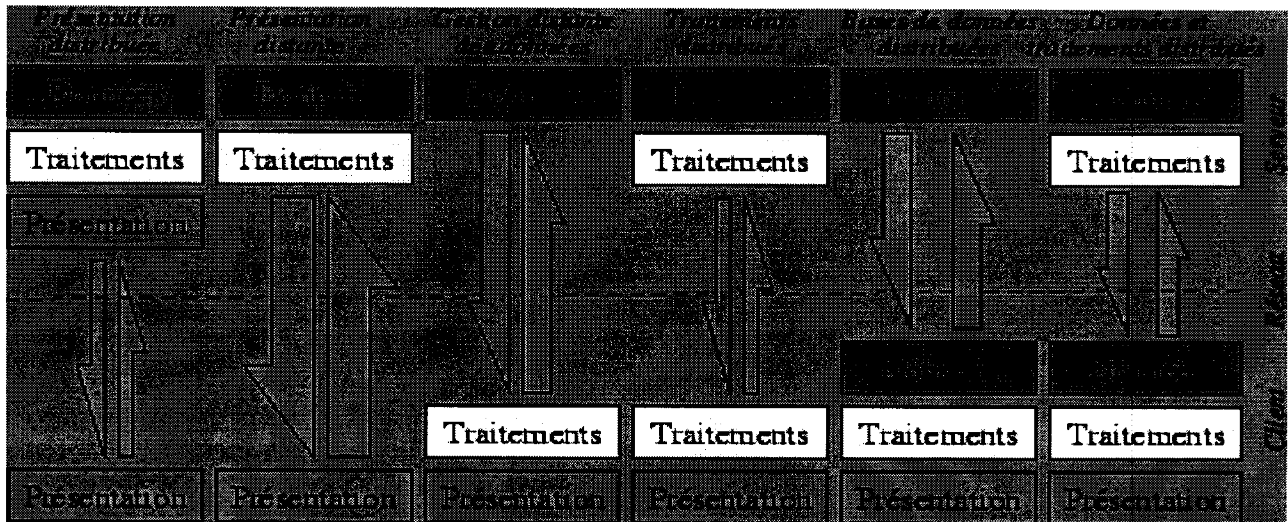


Figure 3 : *Les différents types de client-serveur selon la classification du Gartner Group* **[4]**.

Sur ce schéma, le trait horizontal représente le réseau et les flèches entre client et serveur le trafic réseau généré par la conversation entre client et serveur .Nous verrons par la suite que la vision du Gartner Group, en ne prenant en compte qu'un découpage en deux niveaux, est quelque peu limitative.

Le Gartner Group distingue les types de client-serveur suivants, en fonction du type de service déporté du coeur de l'application :

5.1. Présentation distribuée

Correspond à l'habillage ``graphique" de l'affichage en mode caractères d'applications fonctionnant sur site central. La classification ``client-serveur" du revamping est souvent jugée abusive, du fait que l'intégralité des traitements originaux est conservée et que le poste client conserve une position d'esclave par rapport au serveur.

5.2. Présentation distante

Encore appelée client-serveur de présentation. L'ensemble des traitements est exécuté par le serveur, le client ne prend en charge que l'affichage. Ce type d'application présentait jusqu'à présent

l'inconvénient de générer un fort trafic réseau et de ne permettre aucune répartition de la charge entre client et serveur.

5.3. Gestion distante des données

Correspond au client-serveur de données, sans doute le type de client-serveur le plus répandu. L'application fonctionne dans sa totalité sur le client, la gestion des données et le contrôle de leur intégrité sont assurés par un SGBD centralisé.

Cette architecture génère toutefois un trafic réseau assez important et ne soulage pas énormément le poste client, qui réalise encore la grande majorité des traitements.

5.4. Traitement distribué

Correspond au client-serveur de traitements. Le découpage de l'application se fait ici au plus près de son noyau et les traitements sont distribués entre le client et le(s) serveur(s).

Cette architecture permet d'optimiser la répartition de la charge de traitement entre machines et limite le trafic réseau. Par contre il n'offre pas la même souplesse que le client-serveur de données puisque les traitements doivent être connus du serveur à l'avance.

5.5. Données et traitements distribués

Ce modèle est très puissant et tire partie de la notion de composants réutilisables et distribuables pour répartir au mieux la charge entre client et serveur.

Prises individuellement, on peut dire que les deux premières solutions proposées ne correspondent pas à des applications client-serveur. En effet, la présentation distribuée comme l'affichage distant.

La première solution à mettre en oeuvre une relation client-serveur se retrouve au troisième niveau et correspond au client-serveur de données. Ce type d'application, encore appelé client-serveur de Première génération, met en oeuvre une architecture deux-tiers.

Nous allons maintenant nous pencher plus en détails sur ce type d'application

6. L'ARCHITECTURE DEUX TIERS

6.1. Présentation

Dans une architecture deux tiers, encore appelée client-serveur de première génération ou client-serveur de données, le poste client se contente de déléguer la gestion des données à un service spécialisé. Le cas typique de cette architecture est l'application de gestion fonctionnant sous Ms-Windows et exploitant un SGBD centralisé.

Ce type d'application permet de tirer partie de la puissance des ordinateurs déployés en réseau pour fournir à l'utilisateur une interface riche, tout en garantissant la cohérence des données, qui restent gérées de façon Centralisée.

La gestion des données est prise en charge par un SGBD centralisé, s'exécutant le plus souvent sur un serveur dédié. Ce dernier est interrogé en utilisant un langage de requête qui, le plus souvent, est SQL.

6.2. Le dialogue client-serveur

Le modèle client-serveur met en oeuvre une conversation entre deux programmes que l'on peut opposer à l'échange ``maître-esclave" qu'entretiennent les applications sur site central avec leurs terminaux passifs.

Dans une conversation client-serveur, on distingue donc les deux parties suivantes :

- ✓ **Le client**, c'est le programme qui provoque le dialogue.
- ✓ **Le serveur**, c'est le programme qui se contente de répondre au client.

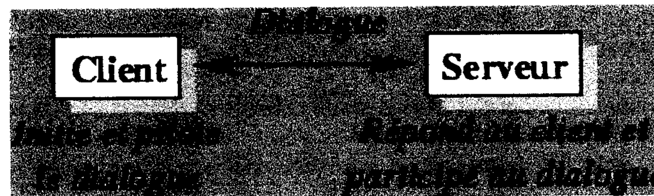


Figure 4 : *la base du dialogue, un besoin du client* **[5]**.

Le client provoque l'établissement d'une conversation afin de d'obtenir des données ou un résultat de la part du serveur. Cet échange de messages transite à travers le réseau reliant les deux machines. Il met en oeuvre des mécanismes relativement complexes qui sont, en général, pris en charge par un *middleware*.

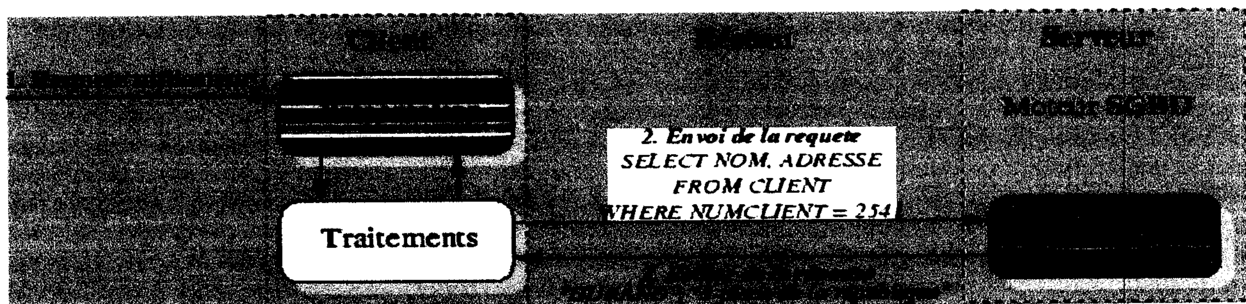


Figure 5 : *Accès aux données en mode deux tiers* **[5]**.

6.3. Le middleware

Le middleware permet de résoudre les problèmes liés à l'intégration des différentes plates-formes utilisées dans vos développements. Il se charge de la communication entre applications ou parties d'applications différentes, éventuellement localisées sur des sites distants et, bien souvent, écrites dans des langages distincts. Il peut s'agir, par exemple, de faire dialoguer une applet Java avec une base de données DB2, de permettre à une feuille de calcul Excel d'obtenir des données à partir d'une base Oracle ou encore de réutiliser un vieux module COBOL sur un site central en affichant les données à partir d'une application graphique C++.

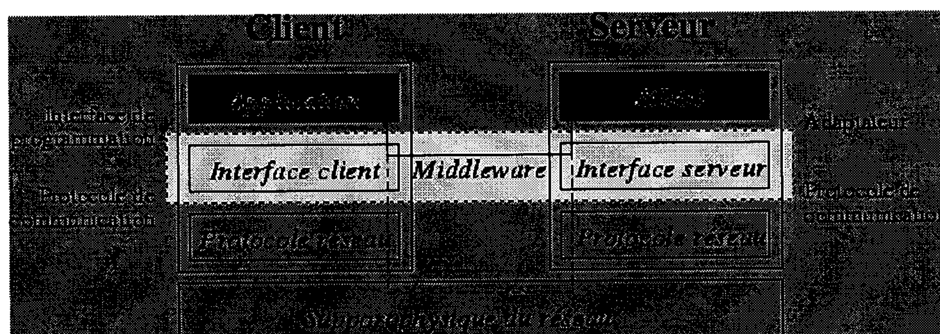


Figure 6 : *Middleware* **[5]**.

6.4. Les services rendus

Un middleware est susceptible de rendre les services suivants :

- ✓ **Conversion** : Service utilisé pour la communication entre machines mettant en oeuvre des formats de données différents.
- ✓ **Adressage** : Permet d'identifier la machine serveur sur laquelle est localisé le service demandé afin d'en déduire le chemin d'accès. Dans la mesure du possible, cette fonction doit faire appel aux services d'un annuaire.
- ✓ **Sécurité** : Permet de garantir la confidentialité et la sécurité des données à l'aide de mécanismes d'authentification et de cryptage des informations.
- ✓ **Communication** : Permet la transmission des messages entre les deux systèmes sans altération. Ce service doit gérer la connexion au serveur, la préparation de l'exécution des requêtes, la récupération des résultats et la dé-connexion de l'utilisateur.

Le middleware masque la complexité des échanges inter-applications et permet ainsi d'élever le niveau des API utilisées par les programmes. Sans ce mécanisme, la programmation d'une application client-serveur serait extrêmement complexe et rigide.

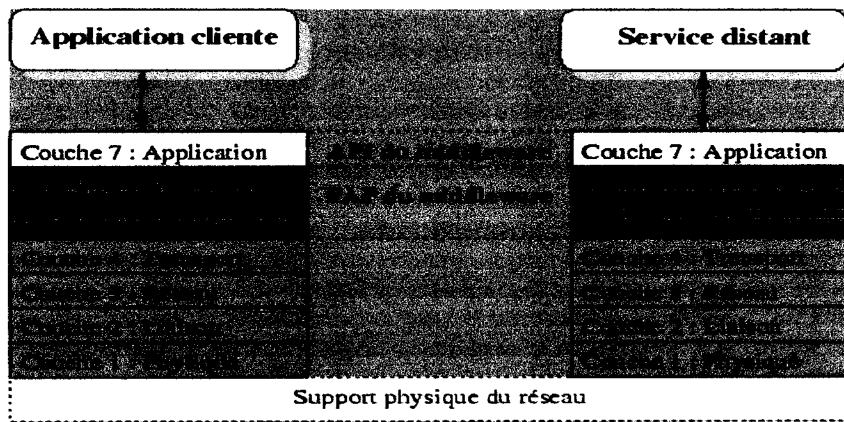


Figure 7 : *Le middleware par rapport au modèle OSI (Open Systems Interconnection)* **[5]**.

6.5. Limites du model client-serveur deux tiers : Le client lourd

L'expérience a démontré qu'il était coûteux et contraignant de vouloir faire porter l'ensemble des traitements applicatifs par le poste client. On en arrive aujourd'hui à ce que l'on appelle le client lourd, ou *fat client*.

L'architecture client-serveur de première génération s'est heurtée à ce constat à l'heure des premiers bilans :

- ✓ on ne peut pas soulager la charge du poste client, qui supporte la grande majorité des traitements applicatifs,
- ✓ le poste client est fortement sollicité, il devient de plus en plus complexe et doit être mis à jour régulièrement pour répondre aux besoins des utilisateurs,
- ✓ la conversation entre client et serveur est assez bruyante et s'adapte mal à des bandes passantes étroites. De ce fait, ce type d'application est souvent cantonné au réseau local de l'entreprise,
- ✓ les applications se prêtent assez mal aux fortes montées en charge car il est difficile de modifier l'architecture initiale,
- ✓ la relation étroite qui existe entre le programme client et l'organisation de la partie serveur complique les évolutions de cette dernière,
- ✓ ce type d'architecture est grandement rigidifié par les coûts et la complexité de sa maintenance.

Malgré tout, l'architecture deux tiers présente de nombreux avantages qui lui permettent de présenter un bilan globalement positif :

- ✓ elle permet l'utilisation d'une interface utilisateur riche,
- ✓ elle a permis l'appropriation des applications par l'utilisateur,
- ✓ elle a introduit la notion d'interopérabilité.

Pour résoudre les limitations du client-serveur deux tiers tout en conservant ses avantages, on a cherché une architecture plus évoluée, facilitant les forts déploiements à moindre coût. La réponse est apportée par les architectures distribuées.

LES ARCHITECTURES DISTRIBUÉES

7. L'ARCHITECTURE TROIS TIERS

7.1. Présentation

Les limites de l'architecture deux tiers proviennent en grande partie de la nature du client utilisé :

- ✓ le frontal est complexe et non standard (même s'il s'agit presque toujours d'un PC sous Windows),
- ✓ le middleware entre client et serveur n'est pas standard.

La solution résiderait donc dans l'utilisation d'un poste client simple communicant avec le serveur par le biais d'un protocole standard.

Dans ce but, l'architecture trois tiers applique les principes suivants :

- ✓ les données sont toujours gérées de façon centralisée;
- ✓ la présentation est toujours prise en charge par le poste client;
- ✓ la logique applicative est prise en charge par un serveur intermédiaire.

7.1. Les premières tentatives

Les premières tentatives de mise en œuvre d'architecture trois tiers proposaient l'introduction d'un serveur d'application centralisé exploité par les postes clients à l'aide dialogue RPC propriétaire.

Les mécanismes nécessaires à la mise en place de telles solutions existent depuis longtemps :

- UNIX propose un mécanisme de RPC intégré au système NFS ;
- DCOM de Microsoft permettent l'instauration d'un dialogue RPC entre client et serveur ;
- certains environnements de développement facilitent la répartition des traitements entre le poste client et le serveur ;

- des solutions propriétaires permettent l'exploitation d'un serveur d'application par des clients simplement équipés d'un environnement d'exécution.

Cependant, l'application de ces technologies dans une architecture trois tiers est complexe et demande des compétences très pointues, du fait du manque de standards malgré ses avantages techniques évidents, ce type d'application fut souvent jugé trop coûteux à mettre en place.

7.2. La révolution Internet

7.2.1. Introduction

S'il est un phénomène qui a marqué le monde de l'informatique ces dernières années, c'est bien celui d'Internet. Ce réseau mondial, créé en 1969 par l'armée américaine, puis utilisé par les chercheurs et autres scientifiques, a connu une croissance phénoménale auprès du grand public avec l'introduction du *World Wide Web* en 1989. Ce dernier permet de publier simplement des informations richement mises en forme et pouvant même, par la suite, contenir des données multimédia.

La véritable révolution du WWW réside dans son caractère universel, rendu possible par l'utilisation de standards reconnus.

7.2.2. Les standards d'Internet

L'universalité du Web repose sur des standards simples et admis par tous :

- ✓ HTML, pour la description des pages disponibles sur le Web ;
- ✓ HTTP, pour la communication entre navigateur et serveur Web ;
- ✓ TCP/IP, le protocole réseau largement utilisé par les systèmes Unix ;
- ✓ CGI, l'interface qui permet de déclencher à distance des traitements sur les serveurs Web.

7.2.3. Adaptation à l'entreprise (Intranet)

Aucun des mécanismes mis en oeuvre par Internet n'est exempt de défaut et il est relativement simple de trouver plus performant. En fait, la force de l'ensemble repose essentiellement dans son universalité.

La notion d'Intranet est née de l'intégration des principes d'Internet et des technologies déployées dans L'entreprise :

- on utilise le réseau local de l'entreprise,
- les données sont toujours gérées par un SGBD,
- les mécanismes utilisés pour interroger le SGBD sont toujours les mêmes.

7.3. Le client léger

Dans l'architecture trois tiers, le poste client est communément appelé client léger, par Opposition au client lourd des architectures deux tiers. Il ne prend en charge que la présentation de l'application avec, éventuellement, une partie de logique applicative permettant une vérification immédiate de la saisie et la mise en forme des données. Il est souvent constitué d'un simple navigateur Internet.

7.4. Répartition des traitements

L'architecture trois tiers, encore appelée client-serveur de deuxième génération ou client-serveur distribué, sépare l'application en trois niveaux de service distincts :

- **premier niveau :** l'affichage et les traitements locaux (contrôles de saisie, mise en forme de données...) sont pris en charge par le poste client.
- **deuxième niveau :** les traitements applicatifs globaux sont pris en charge par le service applicatif.
- **troisième niveau :** les services de base de données sont pris en charge par un SGBD.

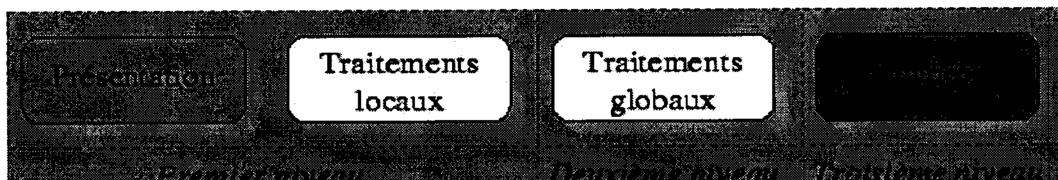


Figure 8 : *Le découpage d'une application en pavés fonctionnels indépendants* **[5]**.

7.5. Exemples de clients légers

Le client léger peut prendre plusieurs formes :

- ✓ un poste Windows équipé d'un navigateur HTML;
- ✓ un terminal Windows (NET PC) correspondant à un PC minimal.

Le client léger a souvent été présenté comme le successeur du PC sous Windows. En fait, le client léger ne prétend pas remplacer tous les PC dans l'entreprise, il se présente plutôt comme une alternative à ce dernier pour certains besoins particuliers et cohabite très bien avec l'existant.

7.6. Limites du model client-serveur trois tiers

L'architecture trois tiers a corrigé les excès du client lourd en centralisant une grande partie de la logique applicative sur un serveur HTTP. Le poste client, qui ne prend à sa charge que la présentation et les contrôles de saisie, s'est trouvé ainsi soulagé et plus simple à gérer.

Par contre, le serveur HTTP constitue la pierre angulaire de l'architecture et se trouve souvent fortement sollicité et il est difficile de répartir la charge entre client et serveur. On se retrouve confronté aux épineux problèmes de dimensionnement serveur et de gestion de la montée en charge rappelant l'époque des mainframes.

De plus, les solutions mises en oeuvre sont relativement complexes à maintenir et la gestion des sessions est compliquée.

Les contraintes semblent inversées par rapport à celles rencontrées avec les architectures deux tiers : le client est soulagé, mais le serveur est fortement sollicité. Le phénomène fait penser à un retour de balancier.

8. LES ARCHITECTURES N-TIERS

8.1. Présentation

L'architecture n-tiers a été pensée pour pallier aux limitations des architectures trois tiers et concevoir des applications puissantes et simples à maintenir. Ce type d'architecture permet de distribuer plus librement la logique applicative, ce qui facilite la répartition de la charge entre tous les niveaux.

Théoriquement, ce type d'architecture supprime tous les inconvénients des architectures précédentes :

- elle permet l'utilisation d'interfaces utilisateurs riches;
- elle sépare nettement tous les niveaux de l'application;
- elle offre de grandes capacités d'extension;
- elle facilite la gestion des sessions.

L'appellation ``n-tiers" pourrait faire penser que cette architecture met en oeuvre un nombre indéterminé de niveaux de service, alors que ces derniers sont au maximum trois (les trois niveaux d'une application informatique). En fait, l'architecture n-tiers qualifie la distribution d'application entre de multiples services et non la multiplication des niveaux de service.

Cette distribution est facilitée par l'utilisation de composants ``métier", spécialisés et indépendants, introduits par les concepts orientés objets (langages de programmation et middleware). Elle permet de tirer pleinement partie de la notion de composants métiers réutilisables.

Ces composants rendent un service si possible générique et clairement identifié. Ils sont capables de communiquer entre eux et peuvent donc coopérer en étant implantés sur des machines distinctes.

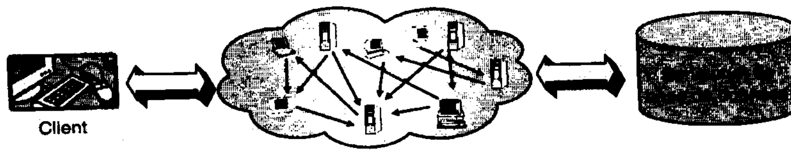


Figure 9 : *L'architecture distribuée* **[2]**.

8.2. L'approche objet

Les évolutions successives de l'informatique permettent de masquer la complexité des mécanismes mis en oeuvre derrière une approche de plus en plus conceptuelle.

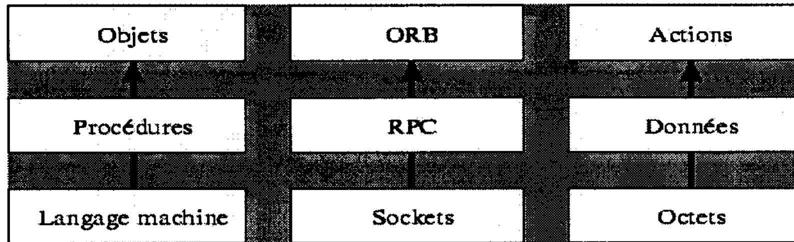


Figure 10 : *Evolution des technologies utilisées pour la mise en oeuvre d'applications Distribuée* **[5]**.

Ainsi, les langages de programmation ont tout d'abord été très proches de la machine (langage machine de bas niveau), puis procéduraux et, enfin, orientés objets. Le succès du langage Java a véritablement popularisé ce mode de programmation.

Les protocoles réseau ont suivi le même type d'évolution. Ils furent d'abord très proches de la couche physique, avec les mécanismes de sockets orientés octets. Ensuite, la notion de RPC a permis de faire abstraction des protocoles de communication et, ainsi, a facilité la mise en place d'application client-serveur. Aujourd'hui, l'utilisation d'ORB permet une totale transparence des appels distants et permet de manipuler un objet distant comme s'il était local. Le flux d'informations fut donc initialement constitué d'octets, puis de données et, enfin, de messages.

Les méthodes de conception orientées objet telles qu'UML ou OMT permettent une modélisation plus concrète des besoins et facilitent le passage de la conception à la réalisation.

8.3. Les objets métier

8.3.1. Les Java Beans

Selon les spécifications de Sun, un Java Beans est un composant logiciel réutilisable qui peut être manipulé par un outil d'assemblage. Cette notion assez large englobe aussi bien un simple bouton qu'une application complète. Concrètement, les Java Beans sont des classes Java utilisant des interfaces particulières.

Un Java Beans peut être utilisé indépendamment, sous la forme d'une simple applet, ou intégré à un développement Java. Il peut dévoiler son comportement à ses futurs utilisateurs à l'aide :

- ✓ des propriétés qu'il expose et rend accessible à l'aide d'accesseurs ;
- ✓ des méthodes qu'il permet d'invoquer, comme tout objet Java ;
- ✓ des événements qu'il peut générer pour avertir d'autres composants.

8.2.2. Les EJB (Entreprise Java Beans)

Les Enterprise Java Beans sont des Java Beans destinés à s'exécuter côté serveur :

- ✓ ils ne comportent pas forcément de partie visible;
- ✓ ils prennent en charge des fonctions de sécurité, de gestion des transactions et d'état ;
- ✓ ils peuvent communiquer avec des Java Beans côté client de façon indépendante du protocole (IIOP, RMI, DCOM),

8.3. La communication entre objets

8.3.1 Principe

Pour permettre la répartition d'objets entre machines et l'intégration des systèmes non objets, il doit être possible d'instaurer une communication entre tous ces éléments. Ainsi est né le concept de middleware objet qui a donné naissance à plusieurs spécifications, dont l'architecture CORBA préconisée par l'OMG et DCOM développée par Microsoft.

Ces middlewares sont constitués d'une série de mécanismes permettant à un ensemble de programmes d'interopérer de façon transparente. Les services offerts par les applications serveurs sont présentés aux clients sous la forme d'objets. La localisation et les mécanismes mis en œuvre pour cette interaction sont cachés par le middleware.

La communication entre objets gomme la différence entre ce qui est local ou distant. Les appels de méthodes d'objet à objet sont traités par un ORB, se chargeant d'aiguiller les messages vers les objets (locaux ou distants).

8.3.2 Le modèle CORBA

Le système CORBA permet, au travers du protocole IIOP, l'utilisation d'objets structurés dans un environnement hétérogène. Cette communication, orchestrée par l'ORB, est indépendante des contraintes systèmes des différentes plates-formes matérielles.

Une application accède à un objet distant en utilisant une télécommande locale, appelée *proxy*. Ce proxy lui permet de déclencher les méthodes de l'objet distant à l'aide de primitives décrites avec le langage IDL.

L'OMG effectue toute une série de recommandations connues sous le nom de CORBA services visant à proposer des interfaces génériques pour chaque type de service usuel (nommage, transaction, cycle de vie, ...).

8.3.3 L'appel de procédure distante (RMI)

RMI permet à une application Java, s'exécutant sur une machine virtuelle, d'invoquer les méthodes d'un objet hébergé par une machine distante. Pour cela, le client utilise une représentation locale de l'interface de l'objet serveur. Cette représentation locale est appelée *stub* et représente l'interface de l'objet serveur, appelée *skeleton*.

Un objet distribué se caractérise par son interface et son adresse (URL). La mise en relation des objets est assurée par un serveur de noms.

CONCLUSION

Avec les applications n-tiers, on dispose enfin d'une vision cohérente du système d'information. Tous les services sont représentés sous la forme d'objets interchangeables qu'il est possible d'implanter librement, en fonction des besoins.

Le potentiel de ce type d'application est très important et permettrait enfin l'utilisation d'objets métiers réutilisables.

Si, sur le papier, on pourrait répondre par l'affirmative, l'expérience des précédentes ``révolutions" de l'informatique nous pousse à plus de modération. L'architecture n-tiers propose effectivement un potentiel énorme, reste à voir comment il sera utilisé, à quel coût et, surtout, comment sera gérée la transition depuis les environnements actuels.

CHAPITRE 2

LE MODELE CORBA

LE MODÈLE CORBA

INTRODUCTION GENÉRALE

Les réseaux d'information voient leur développement explosé depuis l'apparition d'interment. En effet celui-ci a standardisé la notion d'informatique répartie, c'est à dire les modèles client/serveur Conjointement à cette évolution, les applications sont de plus en plus gourmandes en ressources système, et elles demandent un haut niveau de tolérance.

Voila les bases sont posés , l'OMG (Object Management Group) , qui est un consortium rassemblant plusieurs entreprises à dominante informatique qui recherche et développe des solutions à base de modèle objet , avec un haut niveau de normalisation .

Le but est« très simple »plutôt que décentralisé toutes les informations,les informations sont disséminées sur plusieurs serveurs Il y a plusieurs avantage avec cette topologie réseaux, le premier est la tolérance aux pannes en effet si une unité (serveur) vient à être inaccessible un autres serveur est la pour assurer le bon fonctionnent du processus, ce qui implique hélas une redondance des informations Le deuxième avantage est une maintenance plus aisée, un autres avantages est qu'il est toujours plus simple de segmenter un problème en plusieurs autres sous problème plus simple .

1. INTRODUCTION

L'intérêt pour CORBA va grandissant, et sa mise en œuvre dans des applications industrielles de grande envergure est désormais une réalité. Cette technologie qui était, il y a peu, réservée aux universitaires et autres gourous de l'informatique, se démocratise. Ces dernières années, CORBA est passée du stade de technologie avant-gardiste a celle de solution efficace et durable. Elle est ainsi devenue l'un des atouts majeurs d'un développement et une solution d'avenir portée par les plus grands éditeurs de logiciels.

Ainsi, lors du lancement de tout nouveau projet d'envergure se pose la question essentielle suivante : quel middleware utiliser dans le cas d'une architecture distribuée ? CORBA, RMI, PC/DCE, DCOM... ? Où faut-il plus simplement se contenter d'une architecture client/serveur traditionnelle?

2. PRESENTATION DE CORBA

A l'inverse d'autres architectures logicielles, CORBA n'est pas un outil ou un ensemble d'outils crée par un seul éditeur de logiciels. CORBA est une norme, un ensemble de spécifications et de recommandations rédigées par un groupe de travail nomme OMG *(Objet Management Group)*, auquel appartiennent la plupart des acteurs informatiques majeurs.

- ✓ Des services supplémentaires (15) utilisables par l'application Service de résolution De noms, de transaction, de persistance… ;
- ✓ Des fonctionnalités additionnelles par rapport à un simple RPC.

3. L'ARCHITECTURE CORBA

Après cette présentation de CORBA, examinons cette architecture de construction d'application d'un point de vue plus technique. Commençons par nous intéresser à l'OMG et à son modèle global. Nous analyserons ensuite les différents éléments qui composent cette architecture.

3.1. L'OMG

L'OMG *(Object Management Group)* a été crée en 1989 par huit sociétés:

- 3Com.
- American Airlines.
- Canon.
- Data General.
- Hewlett-Packard.
- Philips Telecommunications.
- Sun Microsystems.
- Unisys.

Le rôle de ce consortium est de standardiser et de promouvoir sa vision de l'architecture distribuée. Une fois réalisés et imposés, les standards doivent permettre de réaliser des applications morcelées, reparties entre plusieurs ordinateurs, totalement interopérables en termes de plates-formes matérielles, de systèmes d'exploitation ou encore de langages de développement .Le nombre de sociétés membres de augmente très rapidement pour dépasser aujourd'hui les huit cents membres. Les plus grands acteurs du marche en font actuellement partie. Ils viennent de milieux très disparates puisqu'il s'agit de constructeurs (IBM, Compaq, NCR, etc.), d'éditeurs (Inprise, Microsoft, Netscape, Oracle, Rational, etc.), d'universitaires (CERN, CNET, etc.) ou encore d'utilisateurs (Air France, Alcatel, France Telecom, etc.).

3.2. L'OMA

L'OMA *(Object Management Architecture)* est une architecture globale dans laquelle sont incluses toutes les technologies préconisées par l'OMG. Elle ne fait qu'identifier les composants essentiels d'une architecture distribuée, mais n'en précise ni le mode de fonctionnement, ni les caractéristiques. Ce travail est laisse a la charge des différents groupes d'étude internes a l'OMG ; ils doivent spécifier et normaliser ces éléments.

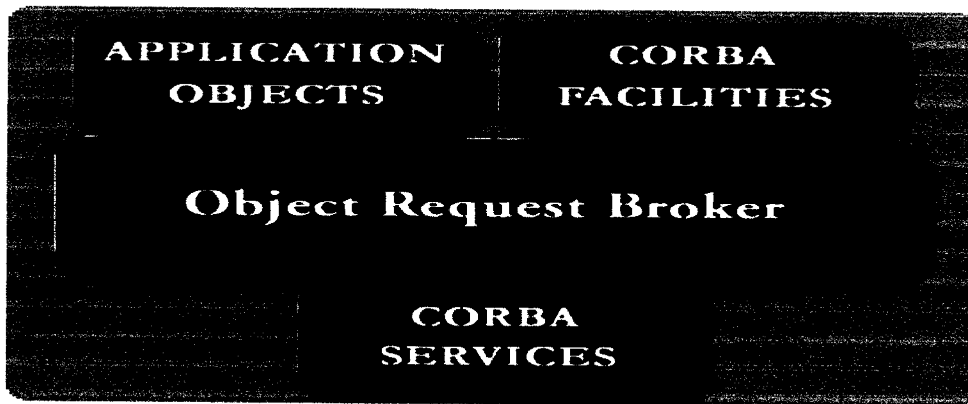


Figure 2 *: l'ensemble de l' OMA* **[6].**

3.3. Les composants de l'architecture CORBA

3.3.1. L'ORB

L'ORB (Object request broker) Sauvent appelé *bus logiciel,* l'ORB est au coeur de l'architecture définie par l'OMG. iI est responsable de la mise en relation et de la communication entre tous les éléments présents dans cette architecture distribuée. Il prend en charge le dialogue entre les objets serveurs et les différents clients qui s'y connectent. Il rend ce dialogue transparent - ou presque - quels que soient les plates-formes, systèmes d'exploitation ou langages utilises. C'est un coordinateur qui gère non seulement Ie fruit de développements, mais aussi tous les outils et composants du système qui vont en permettre le fonctionnement.

Lors du développement d'une application CORBA, le dialogue entre ses différents constituants est rendu totalement transparent pour le développeur. Le serveur déclare mettre a disposition ses objets et le client se contente de demander une connexion a un ou a certains de ces objets, sans pour autant en connaître obligatoirement la localisation ou le format. Dans ces deux cas, l'ORB se charge de localiser les objets, de les charger en mémoire et de transmettre au serveur les demandes du client. Il assure Ensuite des opérations de gestion ou de maintenance, comme la gestion des erreurs ou leur destruction.

Dans cette architecture, l'application cliente ne se préoccupe pas des détails d'implémentation des objets serveur, elle se contente de s'y connecter et de les utiliser.

L'ORB prend en charge la communication entre les divers composants du système distribué.

3.3.2. Objet applicatif

Vos objets - ceux que Vous allez bientôt réaliser - s'inséreront aussi dans l'architecture prévue par l'OMG. Ils seront vus comme des objets spécialises qui utilisent tous les services, utilitaires et interfaces que nous venons de décrire. Le travail sera localisé dans ce sous-ensemble de l'OMA, et sera utilisé toute l'infrastructure de l'architecture CORBA pour donne l'aider dans la réalisation des taches.

3.3.3. Utilitaires communs

Les services que nous venons de décrire s'adressent aux objets. Les CORBAFacilities se placent a un niveau d'abstraction supérieur; ce sont des outils utilises par les applications. Citons par exemple des gestionnaires d'impression, des outils de gestion documentaire, un module de messagerie ou encore diverses boites a outils spécialisées dans des domaines techniques précis.

3.3.4. Interface de domaine

Les interfaces de domaines constituent des spécialisations métiers. Ce sont des objets utilitaires destines a un secteur d'activité donné. Si, par exemple, un objet de gestion des cartes bancaires est crée, il sera utilise par l'ensemble des entreprises du secteur financier. Les applications qui en résulteront pourront communiquer plus facilement par le biais de ce module commun.

3.3.5. Service objet

Les services objets sont des outils destinés au développeur d'application distribuée .ils ont été définis par l'OMG dans le but de simplifier les développements puisqu'ils normalisent et ainsi centralisent des sous-ensembles répétitifs que l'on retrouve dans la plupart des développements. Par exemple, une grande partie des applications réparties ont besoin de règles de sécurité et de protection pour les données transmises sur le réseau. Pour éviter le développement d'une kyrielle de modules similaires, l'OMG a défini un service *Sécurité,* valable quel que soit le fournisseur de l'ORB. Ainsi, les développeurs CORBA peuvent recourir à ce service générique pour diminuer les temps de développement et fiabiliser leurs applications. Voir la figure 3.

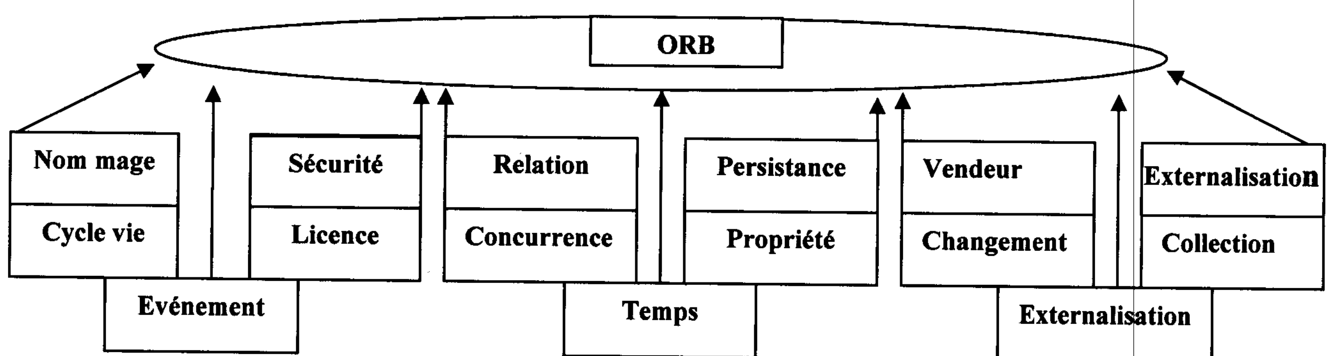


Figure 3 : *les services offerts par la norme CORBA.*

Services disponibles :

- nommage, courtage, événements & notification, transactions ;
- cycle de vie : gestion de l'évolution des objets (déplacement, copie, ...) ;
- persistance : sauvegarde de l'état des objets ;
- concurrence : gestion de verrous ;
- relation : gestion d'associations entre objets ;

- externalisation : mécanisme de «mise en flux» pour des objets ;
- requête : envoi de requête «à la SQL» vers des objets ;
- licence : contrôle de l'utilisation des objets ;
- propriétés : gestion d'attributs dynamiques pour des objets ;
- sécurité : gestion sécurisée de l'accès aux objets ;
- temps : serveur de temps et synchronisation d'horloges ;
- collection : gestion de groupes d'objets.

Remarque : peu d'ORBs offrent tous ces services.

A ce stade, nous avons appris que l'OMG a crée l'OMA qui contient plusieurs éléments dont l'ORB. Cependant, quel est le lien précis entre l'OMA et CORBA ? Le rôle de CORBA est d'implémenter les fonctionnalités décrites sous le nom d'ORB. Cependant, par abus de langage, OMA et CORBA sont généralement confondues et on parlera de CORBA pour tout système distribue s'appuyant sur l'architecture définie par l'OMG. Intéressons-nous maintenant à la norme CORBA et à son évolution. Deux années après sa création en 1991, l'OMG a publié la première version, notée 1.0, de son projet d'architecture distribuée. Il s'agissait alors de poser les premiers jalons de ce modèle. Celui-ci permettait déjà l'exécution de systèmes composés d'objets distribués, repartis sur des plates-formes différentes et écrits dans de multiples langages de programmation. Cependant le défaut majeur de CORBA 1.0 était de ne pas proposer un protocole de communication standard, identique à tous les ORB du marche, ce qui entraînait une incompréhension totale entre des objets bases sur des ORB différents.

4. LES COMPOSANTES DE L'ORB

Nous allons maintenant détailler les différents éléments au coeur de la norme CORBA : POA, BOA, OSI, IIOP, IOL, etc. Tous ces sigles - pour l'instant ésotériques- vous seront bientôt familiers. La Figure 4 présente un schéma qui synthétise toutes les notions que nous allons examiner ensuite.

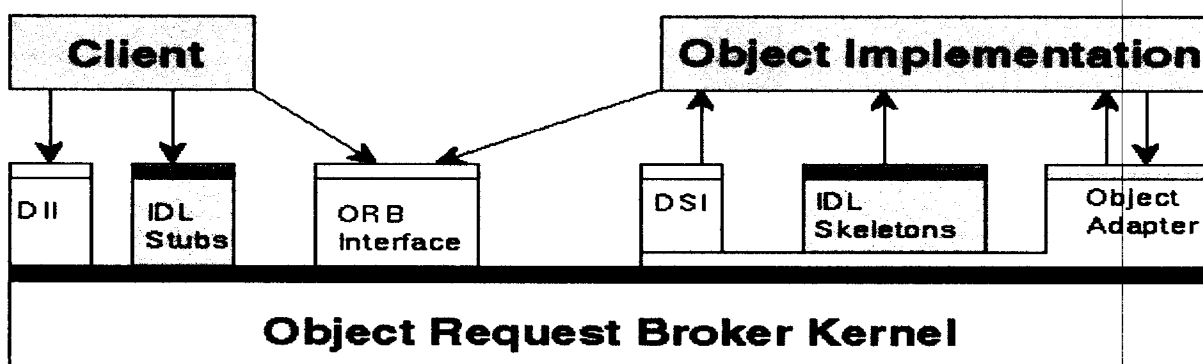


Figure 4 : *noyau CORBA* **[7]**.

4.1. Langage OMG/IDL

➢ **La notion de contrat**

Le langage OMG-IDL (Interface Definition Language) permet d'exprimer, sous la forme de contrats IDL.

La coopération entre les fournisseurs et les utilisateurs de services, en séparant l'interface de L'implantation des objets et en masquant les divers problèmes liés à l'interopérabilité, l'hétérogénéité et la Localisation de ceux-ci. Un contrat IDL spécifie les types manipulés par un ensemble d'applications réparties, C'est-à-dire les types d'objets (ou interfaces IDL) et les types de données échangés entre les objets. Le contrat IDL isole ainsi les clients et fournisseurs de l'infrastructure logicielle et matérielle les mettant en relation à travers le bus CORBA.

Les contrats IDL sont projetés en souches IDL (ou interface d'invocations statiques SII) dans L'environnement de programmation du client et en squelettes IDL (ou interface de squelettes statiques SSI) dans L'environnement de programmation du fournisseur. Le client invoque localement les souches pour accéder aux Objets. Les souches IDL construisent des requêtes, qui vont être transportées par le bus, puis délivrées par celui-ci aux squelettes IDL qui les délégueront aux objets. Ainsi le langage OMG-IDL est la clé de voûte du bus d'objets répartis CORBA.

4.2. Le langage de définition d'interface

Pour permettre la communication entre les applications clientes et les objets CORBA, il est obligatoire de définir les interfaces des objets auxquels vous désirez accéder. Ainsi, l'interface est le moyen idéal pour définir le rôle d'un objet sans entrer dans les considérations propres à son implémentation. Pour cela, l'OMG a spécifié un langage de définition d'interface, complet et simple, mais qui tient compte de tous les concepts objets. Il a été baptisé IDL *(Inteiface Definition Language).*

Nous l'utiliserons pour définir les opérations supportées par les objets (méthodes), les attributs associés (données), ou encore les exceptions qu'ils peuvent produire. Ce langage est très riche malgré une simplicité apparente.

Le rôle de l'OMG est aussi de définir les règles de transformation *(mapping)* d'une interface écrite en IDL dans les langages qui sont ensuite exploites pour implémenter les objets correspondants. Actuellement, un *mapping* officiel a été publie pour les langages suivants : C, C++, Java, Smalltalk, COBOL et Ada. Vous en trouverez la description complète dans les spécifications de la norme CORBA.

4.2.1. Modules stub et skeleton

Dans l'architecture CORBA, les communications entre un client et un serveur sont prises en charge par l'ORB avec deux modules spécifiques, appelées *stub* (souche).pour la partie cliente et *skeleton* (squelette) au niveau du serveur.

Nous avons pas écrire ces deux modules puisque des outils sent fournis par l'éditeur de l'ORB que nous allons utiliser pour générer ce code a partir du fichier IDL. Avec VisiBroker, il s'agit de IDL2JAVA et IDL2CPP. Le code généré est relativement complexe; il est difficile de le produire sans ces outils. De plus, il doit impérativement correspondre a la version courante de l'objet, sinon le client ne pourra jamais se connecter au serveur. En fait, si vous n'aviez pas ces outils de conversion (IDL vers stub et skeleton) à votre disposition, vous ne feriez probablement pas du CORBA !

4.2.2. Adaptateur d'objets

Dans l'architecture CORBA, l'adaptateur d'objets assure la gestion des différents objets qui se situent à l'intérieur de l'application serveur. Il constitue en quelque sorte la clé de voûte de la partie serveur d'une application CORBA.

En réponse aux demandes transmises par l'ORB, l'adaptateur est responsable de :

- L'activation des objets, en fonction des demandes qui lui sont faites ;
- La transmission des demandes émises par les applications clientes vers un objet particulier ;
- L'appel de méthodes standard pour créer et détruire les objets ;
- La détermination de l'état d'un objet.
- Etc....

Actuellement, deux types d'adaptateurs coexistent dans les applications et outils de développements CORBA :

- Le BOA, issu des anciennes versions de la norme, présent dans la totalité des ORB disponibles sur Ie marché.
- Le POA, défini a I' occasion de CORBA 2.2 pour accroître la portabilité des applications CORBA, qui est intègre dans les dernières versions des outils de développements.

L'adaptateur d'objets n'est présent que côté serveur. Si vous devez migrer une application depuis une architecture BOA pour utiliser le POA, les différents modules clients ne nécessiteront pas de modifications.

a. L'ancien adaptateur d'objets : *le BOA*

Le BOA *(Basic Object Adapter)* est, en français, l'adaptateur d'objets de base.

Il s'inscrit dans une politique large capable d'accéder à différents types d'objets. Des adaptateurs spécialises peuvent être crées, par dérivation, pour correspondre aux besoins d'un type d'objet particulier: par exemple, des OODA *(Object Oriented Database Adapter)* pour l'accès a des objets stockes dans des bases de données orientées objet, ou des LOA *(Library Object Adapter)* si les objets sont stockes dans des bibliothèques.

Le principal reproche fait au BOA est d'être trop simple et insuffisamment spécifie par l'OMG. En effet, seul un jeu très restreint de fonctionnalités a été décrit dans la norme CORBA, ce qui a incite les différents éditeurs d'ORB à ajouter des extensions propriétaires au BOA, afin d'accroître ses possibilités. Ainsi, chaque BOA, pourtant base sur les mêmes spécifications, est devenu particulier et incompatible avec son voisin.

Cette incompatibilité ne se situe pas au niveau du fonctionnement de serveurs et de leur interfaçage, mais au niveau de leur code: il est ainsi impossible de compiler, sans modification, le code d'un serveur avec deux ORB provenant d'éditeurs différemment.

b. Le "nouvel" adaptateur d'objets : *le POA*

Avant la norme CORBA 2.2, le BOA était l'adaptateur standard, implémente par tous les ORB. Cependant, si la norme imposait la présence d'un BOA, elle ne précisait pas comment l'implémenter. Le résultat de cette imprécision est simple: les BOA actuels sont tous incompatibles puisqu'ils ont été directement définis par les éditeurs.

La norme CORBA 2.2 impose désormais la définition d'un adaptateur particulier, noté POA *(Portable Object Adapter)*. Celui-ci est parfaitement décrit dans la norme, afin de permettre une compatibilité à ce niveau entre les différents ORB. Son architecture interne:

- Rend compatible entre elles les différentes implémentations d'objets ;
- Propose de nouvelles fonctionnalistes aux serveurs CORBA ;
- Définit de façon homogène les actions demandées à un objet serveur (activations, terminaison).

En accompagnement du POA est apparue une nouvelle terminologie qui regroupe les éléments suivants :

- **Servant.** Anciennement appelé "implémentation de l'objet", il représente l'objet que vous souhaitez réaliser.

- **Serveur**. Il s'agit de l'application qui s'exécute en proposant différents objets, les servants, aux applications clients.
- **Client**. L'application client utilise les services des objets servants distants.
- **Référence d'objet**. L'application client utilise une référence d'objet pour accéder a un servant qui permet de masquer la non-proximité de l'objet réel.

4.2.3. Identification des objets (IOR)

L'étape de connexion entre un client et un objet serveur est appelée *binding* ou *bind.* Il s'agit de l'une des opérations les plus importantes puisque, sans elle, le dialogue ne peut pas être instauré entre les différentes parties d'une application.

En quoi consiste réellement cette étape ? A obtenir une référence sur l'objet auquel on désire accéder. Avant de parler d'interopérabilité ou de distribution d'objets, il a fallu trouver un moyen d'identifier sans ambiguïté un objet CORBA précis, en en indiquant non seulement le nom et la classe, mais aussi la machine a partir de la quelle on peut y accéder.

Ce moyen d'identification est dénommé IOR *(Interoperable Object Reference).* Lors du bind, cette référence d'objet est retournée au client, ou plus précisément au stub du client, qui le stocke dans l'une de ses variables internes. Ensuite, lors de chaque accès à l'objet serveur, qu'il s'agisse d'appels de méthodes ou d'accès à des données, l'IOR est utilisée de sorte à toujours se connecter au même objet servant.

Sur la Figure 5, vous pouvez constater que IOR est placée dans le stub du client sans aucune interaction avec la partie applicative. Cependant, elle est utilisée ensuite pour accéder directement à l'objet servant.

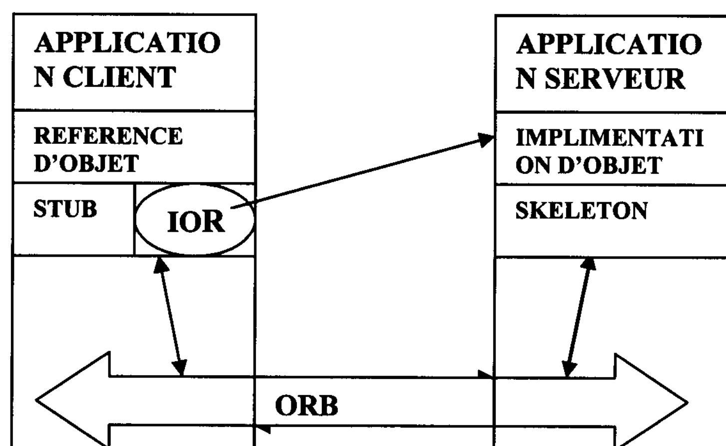


Figure 5 : *le dialogue CORBA* **[2]**.

4.2.4. Protocole IIOP

A la base de CORBA 2.0 se trouve un protocole de communication commun : IIOP. Pour être précis, signalons que ce dernier est la spécialisation d'un protocole plus général, baptisé GIOP *(General Inter-ORB Protocol).* Charge de faire communiquer des ORB, ce dernier est conçu pour fonctionner avec tous les protocoles réseau existants par le biais de spécialisations. Par exemple, IIOP est la spécialisation de GIOP adaptée au protocole TCP/IP ; c'est donc lui qui prend en charge toutes les communications entre les ORB dans le monde internet.

4.2.5. Les référentiels

L'interface Repository est un référentiel qui contient des informations sur les objets utilises durant l'exécution du système ; ce sont généralement celles que vous définissez dans l'IDL de l'objet. Toute fois, d'autres données peuvent figurer dans ce référentiel ; il s'agit par exemple d'informations de débogage ou de routines particulières.

Pour sa part, le référentiel d'implémentation contient des informations qui permettent à l'ORB de localiser et d'activer les différentes implémentations des objets références dans l'Interface Repository. Les diverses politiques d'activation des objets sont définies à ce niveau, tout comme des opérations liées, par exemple, à la sécurité ou au suivi de l'utilisation des objets.

a. Interfaçage dynamique : DII/DSI

La norme CORBA spécifie un moyen d'accéder a des objets lors de l'exécution d'une application sans pour autant, avant cette exécution, avoir détermine quels seront ces objets. Ainsi, les possibilités de l'objet distant ne sont détermines que lors de l'accès réel a cet objet.en utilisant :

DII *(Dynamic Invocation Interface),* on construit dynamiquement des requêtes vers des objets CORBA pour les quels on ne possède pas de stub. Avec **DSI** *(Dynamic Skeleton Interface),* on intercepte les requêtes en provenance de clients, sans avoir pour cela génère de skeleton. **DSI** est la réciproque de **DII,** coté serveur.

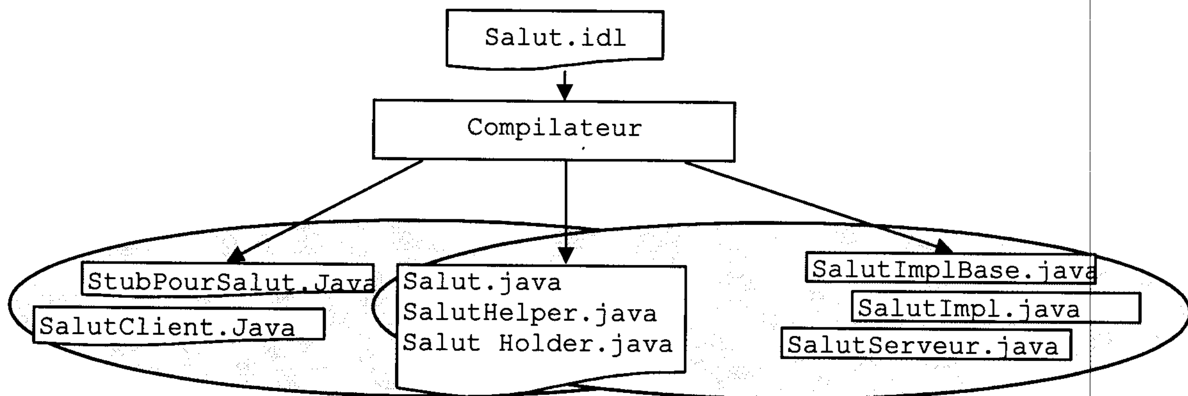
Ces deux techniques servent lors de la réalisation d'applications qui doivent s'adapter automatiquement à l'évolution dynamique des interfaces des objets, à leurs descriptions **IDL** (ce qui est utile notamment dans le cas de langages de type script).

Lors de l'utilisation de ces deux interfaces, le référentiel (**IR**) est utilisé pour résoudre correctement les appels. Rappelons que l'Interface Repository contient des informations sur les méthodes, les types de paramètres, et toutes les informations utiles relatives aux objets CORBA.

5. EXEMPLE : Salut en Java

Après le tour d'horizon des concepts proposés par l'OMG dans ce chapitre, nous allons passer à la mise en pratique du bus CORBA. Nous avons choisi pour cela de présenter la réalisation d'un exemple simple (« Salut mes Collègues ») illustre l'utilisation du bus CORBA.

Cette application nous permet de voir concrètement l'utilisation du langage OMG-IDL et l'implantation d'objets CORBA, de serveurs et de clients avec les langages Java.



- **Interface OMG-IDL Salut**

```
Interface Salut {
  Void doSalut() ;
} ;
```

- **Interface Java Salut**

```
Public Interface Salut extends org.omg.CORBA.Object {
  public Void doSalut() ;
} ;
```

- **Classes Java SalutHelper, SalutHoldre**

- **Classe StubPourSalut : implantation de la souche**
  - Sert à envoyer les requêtes
  - Utilisé en lieu et place de l'objet Salut dans le client
  - Invisible pour le programmeur

- **Classe _SalutImplBase**
  - Recoit et décode les requêtes
  - Doit être hérité par l'implantation

```
                                                  <SalutImpl.java>
public class SalutImpl extends _SalutImplBase {
  public SalutImpl() {}
  public void doSalut() {
    System.out.println(" Salut mes Collègues !");
  }}
```

- **Code Source du Serveur**

```
                                                        <SalutServeur.java>
Public class SalutServeur {
  Public static void main (String args[]) {
   try {
       // Initialisation du Bus Corba
       org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
       org.omg.CORBA.BOA boa = orb.BOA_init(args, null);
       // Creation de l'objet serveur
       SalutImpl salut = new SalutImpl();
       // Diffusion de la référence de l'objet
    System.out.println("IOR(Salut)=" +orb.object_to_string(Salut));
       // Mise en attente du serveur
       boa.impl_is_ready(null);
       System.exit(0);
      } catch (org.omg.CORBA.SystemException exc) {
       exc.printStackTrace();
       System.exit(1);
    }
  } }
```

- **Code Source du Client**

```
                                                        <SalutClient.java>
Public class SalutClient {
    Public static void main (String args[]) {
      try {
      // Initialisation du Bus Corba
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
      // Creation de la souche Java referencant l'objet distant
      org.omg.CORBA.Object obj = orb.string_to_object(args[0]);
      Salut salut = SalutHelper.narrow(obj);
      // Invocation de l'objet distant
      salut.doSalut();
      System.exit(0);
     } catch (org.omg.CORBA.SystemException exc) {
      exc.printStackTrace();
      System.exit(1);
    }
  } }
```

CONCLUSION

Les modèles de programmation distribuée, à l'instar de CORBA, possèdent de nombreux avantages pour le couplage de codes. Tout d'abord, ils offrent une bonne structuration des codes et des communications grâce notamment à l'utilisation d'interfaces spécifiées des différents codes et au mécanisme d'encapsulation offert par les objets. Par ailleurs, ils supportent naturellement la dynamicité et l'hétérogénéité entre les codes. Ainsi de nombreux travaux on choisi d'utiliser des *middlewares* pour coupler des codes et plus particulièrement la technologie CORBA.

CHAPITRE 3

LE LANGAGE IDL

LE LANGAGE IDL

INTRODUCTION

Apres ce tour d'horizon de l'environnement CORBA, ce chapitre présente le langage de définition d'interfaces défini par l'OMG, appelé aussi *OMG IDL* pour *Object Management Group Interface Definition Language.* Ce langage est un des éléments-clés du bus a objets repartis CORBA : il est utilise pour décrire toutes ses composantes, comme nous le verrons dans les chapitres suivants, mais il est aussi le langage de référence pour définir les contrats entre les fournisseurs d'objets et leurs utilisateurs.

Pour présenter ce langage, nous passons en revue ses différentes constructions syntaxiques et leur sémantique en les illustrant par de nombreux fragments de code IDL. La fin de ce chapitre aborde la projection du langage IDL vers les langages de programmation. Cette projection permet d'utiliser les définitions IDL depuis les langages de programmation, afin de pouvoir développer des applications clientes ou d'implanter des objets CORBA. Nous illustrons alors les grandes lignes des règles de projection vers le langage Java.

1. UN LANGAGE DE DESCRIPTION

Le langage IDL est en premier lieu un langage de description de services orientés objet pour l'environnement CORBA. Cette partie explique les intérêts de ce langage de description et en présente les notions de base.

Ce langage peut être défini comme suit : **le langage IDL permet d'exprimer, sous la forme de contrats IDL, la coopération entre les fournisseurs et les utilisateurs de services, en séparant l'interface de l'implantation des objets et en masquant les divers problèmes lies a l'interopérabilité, l'hétérogénéité et la localisation de ceux-ci.**

1.1 Le contrat IDL

L'intérêt majeur du langage IDL est de permettre la définition des contrats passes entre les fournisseurs d'objets CORBA et leurs utilisateurs. Les fournisseurs décrivent, grâce au langage IDL, l'ensemble des interfaces des objets qu'ils veulent fournir à leurs clients; ensuite, les clients utilisent ces interfaces IDL pour exploiter a travers leurs programmes les objets ainsi fournis. Les architectes de systèmes d'informations et de services peuvent quant a eux spécifier clairement et simplement les interfaces de programmation (ou API pour *Application Programming Interface)* de leurs divers composants a l'aide du langage standard IDL. Cette description est alors utilisée par les développeurs

pour implanter les objets ou encore pour réaliser des programmes utilisant ces objets.

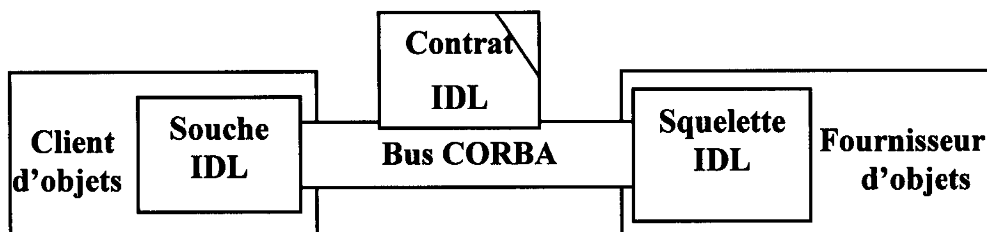


Figure 1 : Le contrat *IDL entre un fournisseur et un client d'objets CORBA* **[3]**

Pour réaliser cette isolation, les contrats IDL sont projetés dans les environnements de programmation pour donner des souches IDL pour les programmes clients et des squelettes *IDL pour les programmes fournisseurs d'obj*ets CORBA (figure 1). Les clients invoquent localement les souches IDL pour accéder aux objets. Les souches construisent alors des requêtes transportées par le bus CORBA en isolant les programmes clients des spécificités du bus employé. Les requêtes sont ensuite délivrées par le bus aux squelettes IDL qui les interprètent et les délèguent aux objets. Les réponses suivent le chemin inverse pour retourner vers les programmes clients. Ainsi, la réunion de ces trois mécanismes (souche IDL - bus CORBA - squelette IDL) implante et permet de mettre en œuvre, à l'exécution, le contrat liant les fournisseurs et les clients.

Chaque interface décrit l'ensemble des opérations fournies par un type d'objets CORBA. Le langage OMG-IDL permet de décrire les interfaces et leurs opérations, de préciser les types des paramètres ainsi que leur *mode* de passage finalement, de spécifier les exceptions pouvant être retournées par les objets.

1.2 La grammaire du langage IDL

Comme le paragraphe précédent le montre, ce langage est relativement simple et syntaxiquement très proche du langage C++. Si la syntaxe est similaire en ce qui concerne la définition de nouveaux types de données, quelques constructions nouvelles apparaissent tout de même pour définir les interfaces des objets, le mode de passage des paramètres et la liste des exceptions d'une opération.

1.3 Le précompilateur IDL

Les descriptions IDL sont stockées dans des fichiers texte. Ces fichiers sont ensuite analyses par un précompilateur IDL qui génère les souches de communication pour le bus CORBA présentées précédemment. Ces souches sont ensuite exploitées *via* des langages de programmation pour réaliser les programmes clients des objets et les programmes implantant ces objets en respectant les règles de projection.

1.4 Les commentaires

Comme dans la plupart des langages, il est possible d'utiliser des commentaires pour documenter les sources IDL. De la même manière qu'en *C++*, les commentaires sont soit précédés par les caractères '//' et se terminent à la fin de la ligne, soit débutent par les caractères '/*' et se terminent par les caractères '* /' comme ci-après.

```
/****************************************************
   ceci est un commentaire sur plusieurs lignes
 ****************************************************/
 //ceci est un commentaire jusqu'a la fin de la ligne
```

1.5 Les espaces de définition

La définition d'un module IDL débute par le mot-clé ***module*** suivi d'un identificateur et contient un nombre quelconque de définitions encadrées par des accolades ({et}). Ces définitions doivent avoir un identificateur unique au sein du module mais peuvent utiliser un identificateur déjà contenu dans un autre module. Tout type de définition peut être contenu dans cet espace: des constantes, des types de données, des interfaces, des exceptions mais aussi d'autres modules. L'exemple suivant illustre un module Math contenant des déclarations d'utilité commune comme la constante **PI** et la définition de type **Positif.**

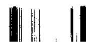
```
// regroupement de déclarations
module Math {
const double PI = 3.14159;
typedef unsigned short Positif;
//autres declarations
 };
```

De plus, les modules sont des espaces de noms ouverts dans lesquels il est possible d'ajouter plus tard de nouvelles définitions. L'exemple de code IDL suivant ouvre à nouveau le module Math pour ajouter la nouvelle déclaration desPositifs.

```
// réouverture du module précédent
module Math {
typedef sequence<Positif> desPositifs;
 };
```

1.6 Les pragmas

Les pragmas IDL sont des directives permettant au précompilateur IDL de générer et créer ces identifiants globaux. La norme CORBA 2.0 définit les trois pragmas suivants:



- Le pragma **prefix** définit le préfixe courant utilisé dans la génération des identifiants. Ce préfixe est valide jusqu'a la fin de l'espace de définitions courant ou jusqu'au prochain pragma *prefix.*

- Le pragma **version** permet d'associer un numéro de version à une définition IDL. II est composé de deux valeurs numériques: le *major* et le *minor.*
- Le pragma **ID** sert, quant a lui, à fixer précisément l'identifiant associé à une définition IDL.

```
#pragma prefix "omg.org"
 module ServiceDates {
     typedef unsigned short Mois;
    #pragma version Mois 1.2
    typedef unsigned short Annee;
   #pragma ID Annee "IDL:monde/Annee:2.3"
};
```

2. UN LANGAGE FORTEMENT TYPÉ

Les spécifications définies avec le langage IDL sont fortement typées en ce qui concerne le type de retour des opérations, les paramètres et le type des attributs des interfaces.

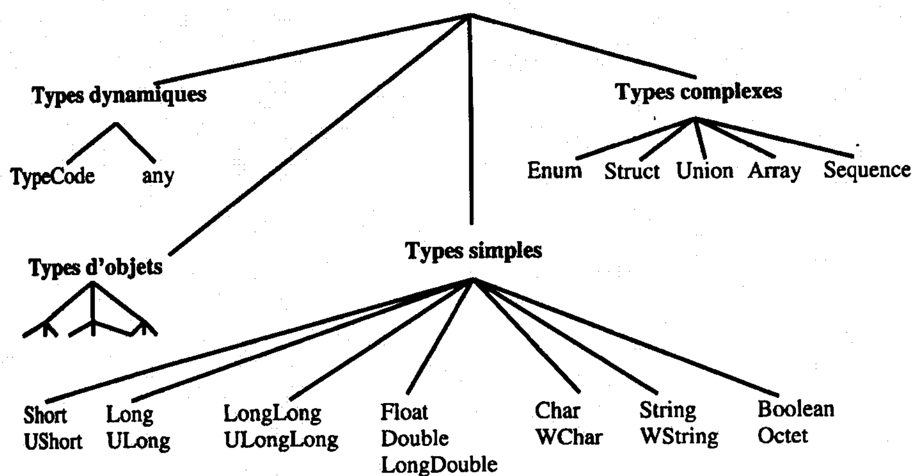


Figure 2 : *les types de données de l'OMG-IDL* **[3].**

Pour exprimer ce typage, le langage IDL fournit quinze types scalaires de base (entier, réel, booléen, caractère, chaîne, octet), des types complexes (énumération, structure, union, tableau et sequence), deux types dynamiques de données *(TypeCode, any)* et finalement des types d'objets *(* figure 3.3). Dans les sections suivantes, nous allons poursuivre cette visite du langage IDL en étudiant comment exprimer ce typage.

2.1. Les types simples

Avant tout, il est nécessaire de savoir que le langage IDL ne contient pas de notion de pointeurs mais seulement des types de valeurs. Les pointeurs sont un concept que nous trouvons uniquement dans certains langages d'implantation des objets CORBA. Le tableau suivant (table 1) énumère ces types de base et les valeurs qu'ils autorisent.

Types de base	Valeurs
short	entiers 16 bits signes
unsigned short	entiers 16 bits non signes
long	entiers 32 bits signes
unsigned long	entiers 32 bits non signes
long long	entiers 64 bits signes
unsigned long long	entiers 64 bits non signes
float	nombres flottants 32 bits au format standard IEEE
double	nombres flottants 64 bits au format standard IEEE
long double	nombres flottants 128 bits
boolean	valeurs booléennes *FALSE* ou *TRUE*
octet	octets 8 bits
char	caractères 8 bits ISO Latin-1
wchar	caractères au format international
string	chaines de caractères
wstring	chaines de caractères au format international

Tableau 1 : *Les types IDL élémentaires et leurs valeurs* **[8].**

2.2 Les constantes

Une description IDL peut contenir des constantes qui seront utilisables depuis les programmes CORBA. Une constante peut être de n'importe quel type simple que nous venons de voir à l'exception du type *octet.* La syntaxe de déclaration d'une constante est similaire a celle du *C++*.

```
<declaration_constante> ::= const <identificateur> '='
<expression>
```

Une constante est déclarée par le mot-clé ***const*** et elle est désignée par un identificateur. Le langage IDL fournit les operateurs arithmétiques classiques pour construire ces expressions (+, -, *, / ...).

```
canst string UneChaineConstante = "une chaine de caracteres";
canst double PI = 3.14159;
canst unsigned short TAILLE = 100;
```

```
canst long TAILLE2 = TAILLE * 2;
```

2.3 Les définitions de type

le langage *C++* a fortement influence les premières phases de conception du langage ID1. Ainsi, ce dernier offre la possibilité de définir de nouveaux types à partir de types existants.

```
<definition_de_type> ::= typedef <type> <identificateur>
```

Pour définir un nouveau type, il suffit d'employer le mot-de ***typedef*** suivi d'un type existant et du nom du nouveau type. les types Jour et Annee de finis ci-dessous illustrent parfaitement cette construction.

```
typedef unsigned short Jour;
typedef unsigned short Annee;
```

2.4 Les types complexes

les types simples sont indispensables pour exprimer les contrats IDL mais sont souvent trop élémentaires pour exprimer des contrats sophistiqués. Ainsi, le langage IDL offre des constructions pour créer à partir des types de base de nouveaux types de données complexes: les énumérations, les structures, les unions, les tableaux et les séquences.

2.4.1 Les énumérations

Les énumérations permettent de définir un ensemble de constantes symboliques regroupées sous un même type. La syntaxe de déclaration d'une énumération est similaire à celle du langage *C++*.

```
<une_enumeration> ::= enum <identificateur> '{'
                      <identificateur> { ',' <identificateur> }*
,},
```

Le code IDL suivant illustre la déclaration des deux types énumérés Mois et JourDeLaSemaine. Dans un programme CORBA, une variable d'un de ces types énumérés ne pourra prendre comme valeur que Janvier a Decembre pour le type Mois, et Lundi à Dimanche pour le type JourDeLaSemaine.

```
enum Mois {
Janvier, Fevrier, Mars, Avril, Mai, Juin,
    Juillet, Aout, Septembre, Octobre, Novembre, Decembre
```

```
};
enum JourDeLaSemaine {
   Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi, Dimanche };
```

2.4.2 Les structures

Le langage IDL fournit la notion de structure ou d'enregistrement que nous pouvons rencontrer dans la plupart des langages de programmation structurée comme le C, Pascal ou Ada.

```
<une_structure> ::= struct <identificateur> '{'
                        { <un_champ> }+
                    '}'
<un_champ>  ::= <type> <identificateur> ';'
```

Une structure IDL se déclaré simplement par le mot-clé ***struct*** suivi d'un identificateur et de la liste des champs de celle-ci. Chaque champ possède un nom le désignant et un type fixant les valeurs possibles.

```
struct Date {
Jour le_jour;
Mois le_mois;
 Annee l_annee;
};
struct Complexe {
double partie_reelle; double partie_imaginaire; };
```

La structure `Date` est composée des trois champs le_jour, le_mois et l_annee respectivement des types Jour, Mois et Annee. La structure Complexe est, Quant à elle, composée de deux nombres flottants double: la partie_reelle le et la partie_imaginaire.

2.4.3 Les unions

Une union IDL contient un discriminant et un ensemble de champs. La valeur du discriminant fixe le champ accessible a une instante donné.

```
<une_union> ::= union <identificateur> switch ( <type_scalaire> )
                ,{, <un_choix> { <un_choix>}* '}'
<type_scalaire> ::= short | unsigned short | long | unsigned long |
            boolean | char | <identificateur_d_une_enumeration>
```

```
<un_choix> ::= <un_label> { <un_label> }* <un_champ>
<un_label> ::= case <valeur_constante> ':' | default':'
<un_champ> ::= <type> <identificateur> ';'
```

L'exemple suivant décrit l'union DateMultiFormat qui est composée d'un discriminant de type ***long*** et de trois champs.

```
union DateMultiFormat switch (long) {
   case 1:
     Date format_date;
   case 2:
   case 3:
     string format_chaine;
   default :
     Jour nombre_de_jours;
};
```

2.4.4 Les tableaux

Le langage IDL autorise la définition de tableaux de taille fixe pour regrouper un ensemble de valeurs d'un même type. La taille et le nombre de dimensions de ces tableaux sont fixes à la description du type.

```
<un_tableau>  ::=  <type>  <una_dimension>  {  <una_dimension>  }*
<une_dimension> ::= '[' <une_constante_de_type_entier_positif> ']'
```

Le code IDL suivant illustre la déclaration d'un tableau de 10 éléments de type Date.

```
typedef Date[10] TableauDates;
```

2.4.5 Les séquences

Le nombre d'éléments d'une séquence est fixé seulement à l'exécution et non pas a la définition du type séquence. Ce qui permet de transporter des ensembles non bornés de valeurs des clients vers les objets et vice versa. II est tout de même possible de fixer une taille maximale pour certaines séquences.

```
<une_sequence> ::=  sequence '<' <type> '>'
                  | sequence '<' <type>, <taille_maximale> '>'
```

Les déclarations de séquences suivantes desJours et desDates permettent de transporter des ensembles de valeurs entre les clients et notre service sans devoir fixer leur taille à la compilation.

```
typedef sequence<Jour> desJours;
typedef sequence<Date> desDates;
```

2.5 Les métatypes de données

Le langage **IDL** définit deux types de métadonnées: les ***TypeCode*** et les ***any.*** Un ***TypeCode*** est un type de données permettant de stocker la description de tout autre type de données IDL.

Le type ***any*** sert à stocker n'importe quelle valeur IDL et son ***TypeCode*** associé. Le type ***any*** est employé dans tous les contextes ou il est nécessaire de transporter des valeurs dont le type n'est pas encore connu au moment de la phase de conception IDL ou encore pour définir des opérations dont l'un des paramètres est polymorphe. L'exemple IDL suivant illustre l'utilité de ce type.

```
interface PileDeChaines {
   readonly attribute string sommet;
   void poser (in string valeur);
   string retirer ();
};
interface PileGenerique {
   readonly attribute any sommet;
   void poser (in any valeur);
   any retirer ();
};
```

3. LES INTERFACES

Une interface IDL décrit l'ensemble des opérations et des attributs offerts par un type d'objets. Il est aussi important de noter que cette description ne présente que la partie publique des objets et ne précise aucun élément sur l'implantation de ceux-ci.

```
<prédéclaration> ::= interface <identificateur>
<une_interface>  ::= interface <identificateur> '{'
                                 <ensemble_de_déclaration>
                 '}'
```

L'exemple ci-dessous illustre la prédéclaration de l'interface **MonInterface** et la définition de

l'interface **Calendrier.**

```
interface MonInterface;
interface Calendrier {
    // déclaration des opérations et des attributs
  };
```

4. LES OPERATIONS

Les opérations IDL représentent les traitements que nous pouvons appliquer sur les objets. Chaque opération IDL est caractérisée par line signature composée des informations suivantes :

1. **Le nom** de l'opération est un identificateur unique.

2. **La liste des paramètres** indique le mode de passage, le type et le nom formel de chacun des paramètres.

Le tableau 2 indique les différents modes de passage et la nature de l'échange des paramètres entre le client et l'objet. Lorsqu'une opération déclenche une exception, tous les paramètres en mode ***out*** et ***inout*** ont une valeur indéfinie.

Mode	Nature de l'échange de la valeur d'un paramètre
`in`	Transport du client vers l'objet.
`out`	Transport en retour de l'objet vers le client
`inout`	Transport dans un sens puis dans l'autre.

Tableau 2 : *Les modes IDL de passage des paramètres* **[8]**.

3. **Le type de retour** peut être un type quelconque IDL pour les opérations retournant un résultat, ou le type ***void*** lorsqu'aucune valeur n'est retournée.

4. **Le mode d'invocation** des opérations est par défaut synchrone, c'est-a dire que le client attend la fin de l'exécution de l'opération pour reprendre la suite de son activité. II est aussi possible de déclarer des opérations asynchrones grâce au mot-clé ***oneway***.

Dans ce cas, Tous les paramètres doivent alors être en mode ***in*** et e type de retour est forcement ***void***.

5. **La liste des exceptions** indique optionnellement les exceptions pouvant être déclenchées par cette

opération. Une opération en mode ***oneway*** ne peut pas déclencher d'exception.

6. **La liste des contextes** permet de transmettre l'équivalent de variables d'environnement depuis le client vers les objets. Ces variables peuvent servir à l'implantation des opérations pour connaitre des informations sur le contexte d'exécution du client.

4.1 Les exceptions

L'implantation d'une opération peut signaler un problème d'exécution à ses appelants à l'aide du mécanisme d'exception. L'opération convertir _chaine renvoie une exception DateErronnee lorsque la chaine passée en paramètre est incohérente. Les exceptions peuvent avoir zéro ou plusieurs champs de valeurs typés.

```
exception DateErronnee {
string raison;
};
Date convertir_chaine (in string une_chaine)
           raises(DateErronnee);
```

5. LES ATTRIBUETS

Afin de simplifier la conception d'interfaces pour des objets exportant une partie de leur état, il fournit un mécanisme d'attributs d'interfaces.

```
<un_attribut> ::= <mode> attribute <type> <identificateur>
 <mode>     ::=  [readonly]
```

Un attribut n'est qu'une spécification décrivant une propriété d'un objet clairement définie dans son interface. Cette déclaration est composée du mot-clé ***attribute*** suivi du type de la propriété et de son nom. Les attributs peuvent être consultés et modifiés (c'est le cas par défaut) ou seulement consultes (le mot-clé ***readonly*** devant l'attribut).

```
interface CarteIdentité {
            attribute string nom;
            attribute string prénom;
            attribute short jours;
            attribute short mois;
            attribute long annee;
            readonly attribute short age;   };
```

6. L'HÉRITAGE

Le langage IDL permet d'encapsuler les objets grâce à la définition d'interfaces. Celles-ci décrivent de manière abstraite la partie publique des objets et masquent tous les détails d'implantation. De plus, certaines interfaces peuvent être conclues par extension d'interfaces déjà existantes. Pour cela, le langage IDL permet de décrire des relations de sous-typage entre les interfaces *via* un mécanisme d'héritage.

```
<une_interface> ::= interface <identificateur> [ <heritage> ] '{'
                    <ensemble_de_declarations>
'}'
  <heritage>   ::=':' <une_interface> { ',' <une_interface> }*
```

Toutes les opérations et les attributs des interfaces héritées font partie de l'interface en cours de conception.

7. LA PROJECTION DU LANGAGE IDL

Une fois les spécifications décrites en IDL, il est nécessaire de les *projeter* vers des langages de programmation pour pouvoir implanter les objets et réaliser les applications utilisant ces objets. Cette projection, appelée aussi liaison, respecte des règles définies par l'OMG pour chaque langage de programmation.

7.1 Le mécanisme de projection

A partir d'une description IDL, deux types de projections sont réalisés:

La projection pour implanter les objets (appelée squelettes IDL) et celle pour les applications utilisant les objets (appelée souches IDL). Ces deux éléments permettent de séparer les applications du substrat de communication, c'est-a-dire le bus CORBA (figure 3). Cette isolation permet ainsi de concevoir des applications reparties multi-Langages.

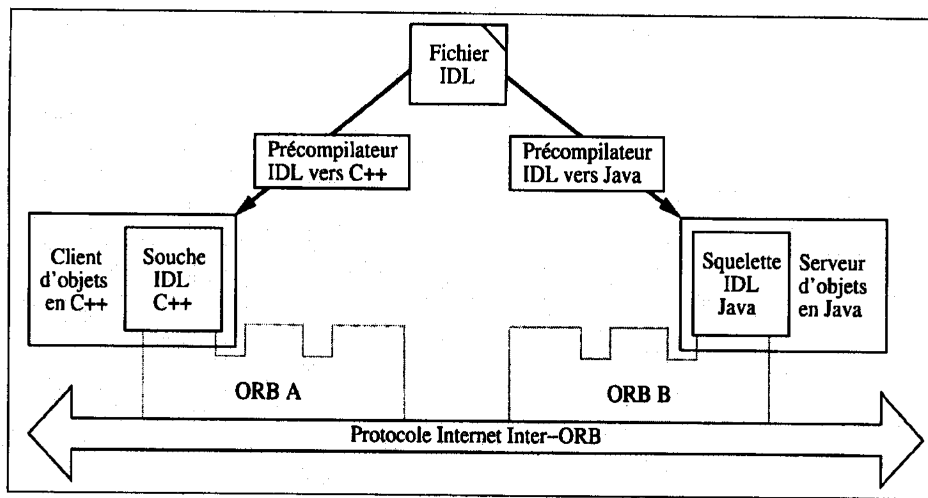


Figure 3 : *La communication entre des applications hétérogènes* **[3].**

Le modèle CORBA permet dans l'absolu la communication entre des applications écrites dans tout type de langage de programmation. Le langage IDL permet de décrire les objets indépendamment de tous les aspects d'implantation et les précompilateurs IDL génèrent le code nécessaire pour utiliser ou implanter ces objets.

7.2 Les règles des projections

A la fin de ce chapitre, nous allons brièvement présenter quelques exemples des projections CORBA en les présentant sous forme de tableau. Il s'agit particulièrement du mécanisme de projection en Java. Cependant, il est à remarquer au niveau du tableau III. , les constructions IDL qui n'ont pas d'équivalent en Java sont traduites par des classes.

CORBA/IDL	Java
short	short
unsigned short	short
long	int
unsigned long	int
long long	long
unsigned long long	long
float	float
double	double
long double	pas encore défini
boolean	boolean
octet	byte
char	char
wchar	char
string	String
wstring	String
fixed	java. Math.BigDecimal

Tableau 3 : *la projection des types élémentaires* **[8].**

CORBA/IDL	Java
module	***package***
type élémentaire	type élémentaire
constante	attribut constant
définition de type	pas de projection.
Enumération, structure, union	classe
Tableau, séquence	tableau
interface	interface
référence d'objet	instance
attribut	1 ou 2 opérations membres
opération	opérations membres
exception	classe
traitement des exceptions	***try*** { ... } ***catch***

Tableau 4 : *Résume des règles de projection vers les langages* Java **[8]**

CONCLUSION

Le langage IDL est donc le langage de définition des contrats entre les fournisseurs d'objets CORBA et leurs clients. Ce langage est totalement indépendant de tout langage de programmation. Cependant, de nouvelle construction ont été ajoutées pour le traitement distribue a base d'objets: les interfaces, les exceptions et les signatures d'opérations. Comme nous avons pu nous en rendre compte, ce langage est vraiment très simple a appréhender et il nous sera agréable pour décrire nos composants logiciels, que ce soit pour encapsuler du code de bas niveau dans des interfaces de programmation clairement spécifiées ou bien pour concevoir des objets distribues sur Internet.

Un précompilateur génère les souches et les squelettes nous permettant d'utiliser et d'implanter naturellement nos objets dans le langage de programmation de notre choix. Actuellement, les règles de projection ont été définies pour les langages de programmation les plus courants.

CHAPITRE 4

LES SERVICES CORBA

LES SERVICES CORBA

INTRODUCTION

Comme nous l'avons évoqué sommaire au deuxième chapitre, l'OMG a développé sous forme d'objets CORBA un ensemble de tâches qualifiées sous le vocable de « services » souvent récurrentes lors do mise on place d'application CORBA, et ce, dans un souci de réduire au mieux les charges des développeurs. Ces services sont regroupés sous l'appellation générique do **COS** (Common Object Services), et **COSS** (Common Object Services Specification) pour leur specification .

Cela va sans dire que ces services fournissent sans doute des APIs d'objets CORBA dotés des interfaces qui rendent leur manipulation facile et leur intégration au sein d'une nouvelle application.

Dans les lignes qui suivent, nous présentons les différentes fonctionnalités des ces services la façon de l'incorporer dans une application.

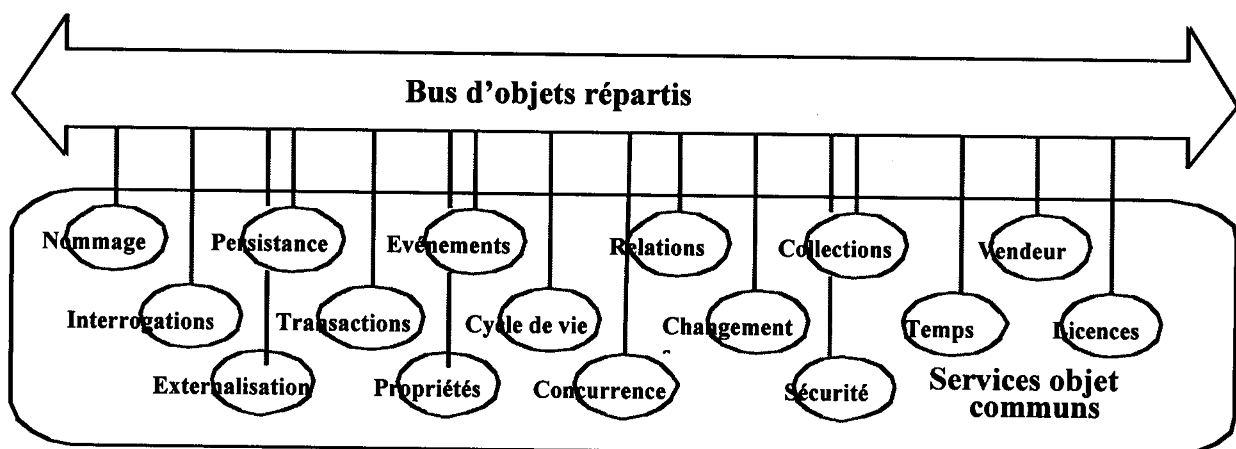


Figure 1 *: les services CORBA* **[8].**

Ces services fournissent un cadre standardisé permettant aux développeurs de répondre plus facilement à certaines problématiques techniques, récurrentes dans les développements a base d'objets distribués.

1. SERVICE DE NOMS

Le service de noms - appelé aussi CosNaming dans la terminologie CORBA - est aujourd'hui disponible dans de nombreuses implémentations CORBA. En adéquation avec les besoins élémentaires d'applications distribuées, le service de noms est l'un des premiers services à avoir été implémenté.

1.1 Présentation

Le service de noms propose l'association d'un ou plusieurs noms logiques aux objets CORBA d'une application et permet de constituer un véritable annuaire ou catalogue d'objets distribués. Le service de noms est constitué d'un graphe de noms généralement alimenté par les programmes serveurs, dans lequel les clients peuvent naviguer et retrouver une ou plusieurs instances d'objets CORBA. Ces instances d'objets sont référencées dans le service à travers le stockage de références d'objets.

La Figure 2 suivant illustre la position du service de noms dans une application CORBA.

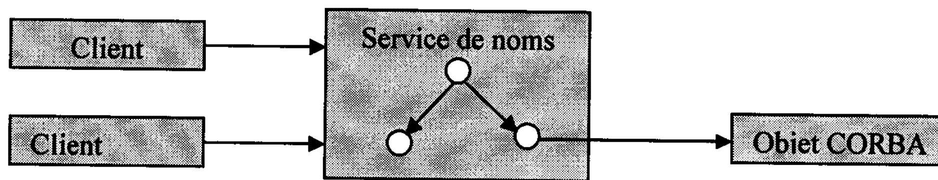


Figure 2 : *Scénario d'utilisation du service de noms* **[2].**

Les caractéristiques générales de ce service peuvent se résumer de la manière suivante :

- ✓ Le service permet de réaliser des associations entre des noms logiques et objets CORBA.
- ✓ Il hiérarchise les objets CORBA au sein d'un référentiel standard commun.
- ✓ Le service n'impose pas une structure de graphe particulière ou une sémantique quelconque sur les noms utilisés.
- ✓ Il offre un accès standard aux différents types de clients, pouvant être codés avec des fournisseurs d'ORB différents. .
- ✓ Il permet de créer des graphes distribués sur plusieurs processus ou machines.
- ✓ Et enfin il assure la persistance du graphe de noms. Le graphe est sauvegarde et rétabli entre les différentes exécutions du service.

Le service de noms est basé sur des spécifications officielles et est défini par des interfaces IDL : il se présente sous la forme d'un objet CORBA qui peut être manipulé par les applications comme n'importe quel autre objet de l'application. Une fois résolu le problème de l'attachement au service lui-même, il propose une interface connue de tous pour manipuler le graphe et retrouver les références d'objets.

Ce service est constitué d'un graphe de contextes de nommage interconnectés (ce graphe pouvant former un espace de nommage reparti) ; chaque contexte gère une liste d'associations nom-référence. A l'intérieur d'un contexte, chaque nom doit être unique mais un objet peut être désigné par plusieurs

noms dans un ou plusieurs contextes. Ce service est composé d'un ensemble de définitions OMG **IDL** regroupées dans le module CosNaming, spécifiant les concepts suivants :

La convention de nommage : CosNaming: : Name est la définition IDL pour représenter le nom des associations ou chemins d'accès dans le graphe des contextes. Un chemin est représenté par une ***séquence*** IDL de noms (CosNaming: : NameComponent).

```
module CosNaming {
 typedef string Istring;

 struct NameComponent {
 Istring id;
 Istring kind ;};
```

Les contextes de nommage : l'interface IDL CosNaming: : NamingContext modélise les contextes de nommage. Elle fournit les opérations de base pour ajouter (bind), modifier (rebind), retrouver (resolve) et détruire (unbind) des associations, pour créer de nouveaux contextes (new_context et bind_new_context), pour connecter des contextes existants (bind_context et rebind_context) et finalement pour les détruire (destroy).

La consultation des associations : l'opération `list` permet de consulter le contenu d'un contexte, représenté par la séquence CosNaming: : BindingList d'éléments structurés CosNaming: : Binding. Si ce contenu est trop important, une nouvelle instance de type CosNaming: : BindingIterator permet de consulter le reste de ce contenu (next_one et next_n).

Le traitement des erreurs : les types d'exception CosNaming:: NotFound, CosNaming:: CannotProceed, CosNaming: : InvalidName, CosNaming: : AlreadyBound et CosNaming: : NotEmpty permettent aux contextes de signaler à leurs appelants les problèmes liés à une mauvaise utilisation des opérations lorsqu'un nom ne correspond a aucune association lors d'une recherche.

2. LE SERVICE VENDEUR

Le service `Vendeur` est l'équivalent des pages jaunes des annuaires téléphoniques. Les fournisseurs enregistrent leurs objets auprès de ce service en les caractérisant par un ensemble de propriétés. Les clients indiquent le type du service désire et les critères le caractérisant. Le service résout alors les demandes des clients en fonction des offres des fournisseurs.

2.1 Présentation

Le service Vendeur est le second service, avec le service Nommage, qui offre la possibilité de découvrir des références d'objets à l'exécution. Le principe de fonctionnement du service Vendeur est illustré par la figure. Une application serveur va « exporter », vers le service Vendeur, une référence sur un de ses objets avec la description associée (1). Les applications clientes qui désirent utiliser cet

objet vont interroger le service Vendeur en fournissant des critères de sélection. Le service Vendeur retourne alors une liste d'objets qui peuvent répondre aux critères de la requête. L'application cliente « importe » ainsi la référence de l'objet applicatif choisi (2). Ensuite, l'application utilise l'objet choisi à travers le bus CORBA (3). Nous constatons ainsi que le scénario d'utilisation de ce service est similaire à celui du service Nommage présenté ci-avant.

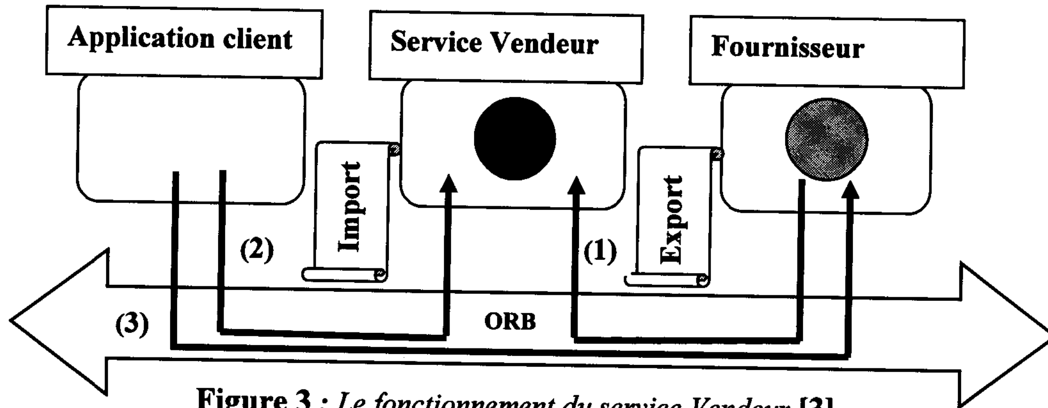


Figure 3 *: Le fonctionnement du service Vendeur* **[3].**

Pour chaque service applicatif enregistre dans le service Vendeur, les informations associées sont :

- **Une référence à un objet** qui implante le service.
- **Un nom d'interface IDL** qui définit les opérations et attributs qui peuvent être utilisés sur la référence d'objet associée.
- **Une liste de propriétés qui** décrivent ce que fait le service.

3. LE SERVICE D'EVENEMENTS

L'appel de méthodes est généralement réalise dans le sens client vers serveur CORBA. Il existe cependant de très nombreux cas de figure dans lesquels les serveurs doivent être capables de notifier une information aux clients. Il s'agit de cas classiques tels que celui des messageries, alertes, cotations en direct, données à rafraîchir, erreurs logiques...

Cette notification peut être considérée comme un événement ou un *callback* (rappel du client par le serveur).

3.1. Présentation

Le service d'événements *(CosEvent)* est spécifié par l'OMG sous la forme de deux fichiers IDL pressentant un ensemble d'interfaces .il est implémenté sous la forme d'un objet CORBA, tout comme le service de noms. Il offre la faculté de définir des canaux d'événements ainsi que des consommateurs et des fournisseurs d'événements.

Deux grands modèles sont exploitables dans le cadre de ce service: des programmes de type *push* notification des évènements sous l'impulsion du fournisseur - et des programmes de type *pull* interrogation du fournisseur. Ce deuxième mode est particulièrement utile pour la mise en place de systèmes travaillant à intervalles fixes.

Les événements sont délivrés de manière asynchrone et les deux modèles de communication peuvent être employés au sein d'un même canal d'événements.

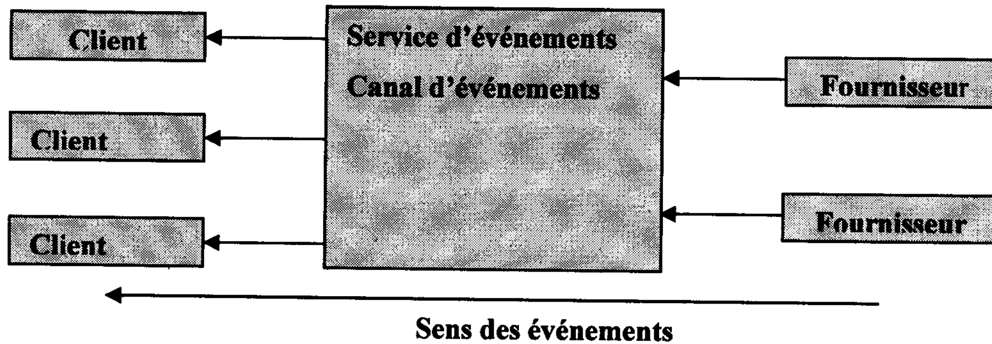


Figure 4 *: Scénario d'utilisation du service d'événements* **[2].**

Les éléments fondamentaux manipules par le service sont donc constitues par :

- ✓ les canaux d'événements ;
- ✓ les fournisseurs ;
- ✓ les consommateurs.

Canaux d'événement *(event chanels)* : sont des objets CORBA assurant le transport des informations entre les multiples consommateurs et fournisseurs.

Les fournisseurs *(suppliers)* : produisent les evenements.

Les consommateurs *(consumers)* : sont des objets traitant les evenements reçus.

Les fournisseurs et les consommateurs fonctionnent selon les deux grands modelés définis dans la section précédente. On distingue :

- ➢ Des fournisseurs push *(push suppliers)* qui contrôlent l'émission des données vers le canal d'evenements.
- ➢ Des clients push *(push consumers)* qui sont notifient par appel du canal d'evenements.
- ➢ Des fournisseurs pull *(pull suppliers)* qui sont interrogés par Ie canal d'evenements.
- ➢ Des consommateurs pull *(pull consumers)* qui contrôlent la réception des données par

appel sur le canal d'événements.

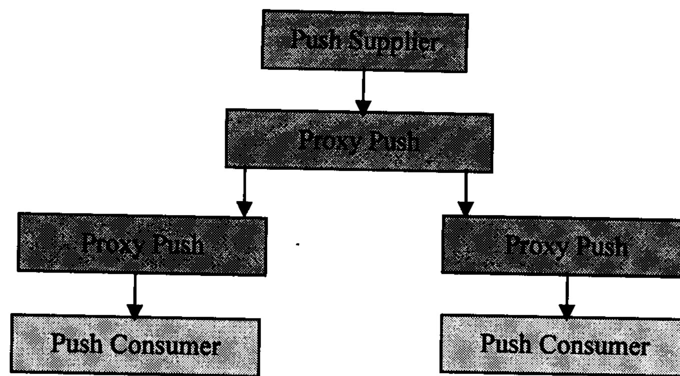


Figure 5 : *le modèle push.*

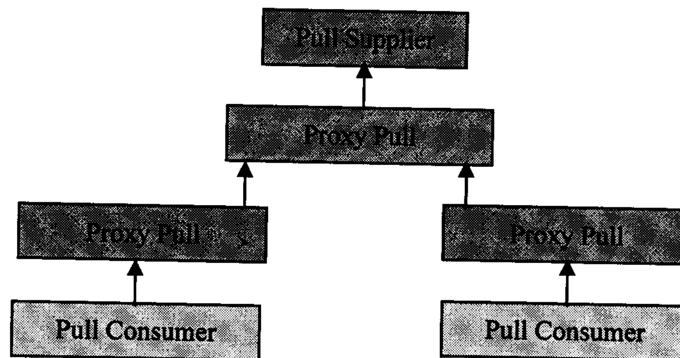


Figure 6 : *le modèle pull.*

4. SERVICE DE SÉCURITÉ

La sécurité constitue aujourd'hui un enjeu majeur des technologies de l'information en général et de l'Internet en particulier. Composé de nombreux maillons, l'Internet offre des capacités de diffusion d'informations sans précèdent. Les risques d'interception de données confidentielles sont toute fois élevés et les technologies de sécurité se sont développes afin de fournir des solutions permettant de fiabiliser des échanges de données.

4.1. Présentation

Le service de sécurité de l'OMG - CORBASec - définit un vaste ensemble d'interfaces et de protocoles afin de sécuriser la mise en œuvre d'applications CORBA.

Ce service Il se base sur des techniques cryptages des messages réseau afin de sécuriser une connexion entre un client et objet CORBA. De multiples domaines applicatifs nécessitent l'emploi de cette technologie : domaine bancaire, bien sur, notamment avec le développement prometteur du commerce électronique (achats par carte bancaire, consultation de soldes, opérations bancaires en ligne), mais

aussi les données touchant les domaines du médical, ayant trait au secret industriel, les applications militaires... D'une manière générale tout les applications manipulant des données sensible.

D'une manière plus précise, la vocation du service de sécurité CORBA est :

- D'obtenir le niveau de confidentialité souhaité : les données sont cryptées et plus difficilement déchiffrables qu'un message en clair (à travers différents protocole de communication).
- De garantir l'intégrité des données: le destinataire des données sera en mesure de détecter une modification des informations et vérifier, par exemple, que transfert de 200 € demande ne s'est pas transformé en 20 000 €.

5. SERVICE DE TRANSACTION

Le service de transaction CORBA est conçu pour fournir un support transactionnel aux applications distribuées. L'application du concept de transactions aux objets offre la possibilité de modifier l'état d'objets CORBA au sein de blocs de traitements, à la manière des opérations réalisées sur les bases de données relationnelles. Les transactions CORBA sont d' ailleurs souvent associées aux transactions supportées par les systèmes de base de données ou par les moniteurs transactionnels.

5.1. Présentation

Les transactions CORBA sont comparables, dans leurs comportements et caractéristiques, aux transactions SQL. Elles peuvent se définir comme une unité de travail supportant certaines propriétés, souvent résumes par la mnémonique "ACID" qui peut se décomposer comme suit :

- **L'Atomicité :** Garantit que l'ensemble des opérations est validé ou annulé.
- **La Consistance :** Implique que les effets d'une transaction sont invariants (une transaction est une transformation correcte d'un état vers un autre).
- **L'Isolation :** Garantit que les modifications non terminées d'une transaction sont invisibles des autres transactions.
- **La Durabilité :** Procure l'assurance qu'une fois terminée, les effets d'une transaction sont durables.

6. SERVICCE CYCLE DE VIE

Le service Cycle de vie décrit les interfaces et les opérations IDL pour la création, la copie, Ie déplacement et la destruction des objets du bus. Pour cela, il définit Ie concept de *fabrique* d'objets (ou

« object factory»). Ce service petit agir sur un objet ou sur un groupe d'objets. Les relations entre objets sont définies a l'aide du service Relations pressente.

7. SERVICE RELATIONS

Le service Relations gère des associations dynamiques reliant des objets sur Ie bus. Ces relations sont encapsulées dans des objets et permettent de parcourir les graphes d'objets et de supporter les contraintes d'intégrité référentielles entre ces objets. Ce service fournit une palette très variée de types de relations: l'appartenance, l'inclusion, la référence, l'auteur et l'emploi.

8. SERVICE EXTERNALISATION

Le service Externalisation permet de stocker la représentation d'un objet dans un flux. du service Externalisation peut être comparé aux flux que l'on trouve dans Ie langage *C++* pour les entrées/sorties.

L'externalisation est l'opération qui stocke un objet dans un flux, à l'inverse, l'internalisation est l'opération qui transforme le contenu d'un flux en un objet CORBA. Entre l'opération d'externalisation et celle d'internalisation, vous pouvez stocker cette représentation de l'objet Sur un support persistant ou la faire transiter sur Internet. Ce mécanisme peut être utilise pour faire de la migration d'objets ou du passage d'objets par valeur.

9. SERVICE DE PERSISTANCE

Le service Persistance offre un ensemble uniforme d'interfaces afin que les objets puissent être stockes de manière stable sur tout type de support rémanent: les systèmes de gestion de bases de données objet (SGBDO), les bases de données relationnelles (SGBDR) ou de simples fichiers.

10. SERVICE DE PROPRIÉTÉS

Il s'agit de permettre l'ajout dynamique de propriété a un objet CORBA .ces propriétés viennent en marge de la définition de l'objet pour le compléter et répondre a un besoin temporaire.

11. SERVICE DE TEMPS

Le but de ce service et de fournir un point de synchronisation entre client et serveur .ainsi, les client se règlent sur l'heur du serveur pour ensuit déclencher des actions communs simultanément.

12. SERVICE DE COLLECTION

Permet de manipuler d'une manière uniforme des objets sous la forme de collection d'itérateur. Les structures de données classiques (listes, piles, tas,..) sont construit par sous-classement. Ce service

est aussi conçu pour être utilisé avec le service d'interrogation pour stocker résultats des requêtes.

13. SERVICE DE CHANGEMENT

Permet de gérer et de suivre l'évolution des diff6rentcs versions des objets. Ce service maintient des informations sur les évolutions des interfaces et des implantations. Cependant, il n'est pas encore spécifié officiellement.

14. SERVICE DE CONTROLRE DE LICENCE

Fournit un mécanisme de gestion des accès concurrent à une ressource.

CONCLUSION

Dans ce chapitre, nous avons présenté de manière synthétique les defférents services objet communs de l'architecture OMA de gestion des objets de L'OMG.

L'intégration de ces services dans les applications peut se faire selon différents modes en fonction de la nature du service .par exemple, le service nommage se présent sur le bus CORBA comme un ensemble de serveurs de contextes de nommage invoques par les applications a travers les souches IDL ou l'invocation dynamique. D'autre services nécessitent qu'une partie de leurs fonctionnalités soit implantes dans les application comme par exemple lorsque les objets veulent offrir les fonction de cycle de vie .de plus, l'utilisation de certain services peut impliquer la coopération de plusieurs serveurs distribués sur le bus .cette coopération forme alors des fédérations de service comme celles des services interrogations ou vendeur.

CHAPITRE 5

IMPLEMENTATION

INTRODUCTION

Après le tour d'horizon des concepts proposés par l'OMG, nous allons passer à la mise en pratique du bus CORBA. Nous avons choisi pour cela de présenter la réalisation d'un exemple gestion bibliothèque illustre l'utilisation du bus CORBA.

Cette application nous permet de voir concrètement l'utilisation du langage OMG-IDL et

attention à la mémoire. En Jbuilder (java), il n'y a pas de difficultés majeures. En CorbaScript, les scripts expriment directement l'essentiel et l'interpréteur prend en charge les détails de mise en œuvre. Au vu des exemples, nous pouvons facilement classer le niveau de difficulté d'utilisation de ces langages avec CORBA. Cependant, cela ne veut pas dire que l'on ne doit utiliser qu'un seul langage (Jbuilder !). Dans une application complexe, il sera plus judicieux d'utiliser chacun de ces langages à l'endroit le mieux approprié : le langage C++ est plus adapté pour construire des serveurs fortement sollicités, le langage Jbuilder (java) est plus adapté pour construire les applications clientes avec les interfaces graphiques et les applets dans le WWW, le langage CorbaScript est plus adapté pour les tâches de mise au point et d'administration d'une application répartie.

1.2 Le Serveur d'application utilisé :

Pour mettre en place une application CORBA, il faut un serveur d'application. Dans ce sens, suite aux contraintes logiciels rencontrées au départ de notre projet, nous avons opté pour le développement de notre application à la version 4 de JBuilder.

En effet celle-ci inclut gratuitement l'IAS comme serveur d'application.

1.3 Implémentation de CORBA utilisée :

Comme nous l'avons souligné ci haut, notre serveur d'application IAS contient le produit visibroker pour java qui constitue l'implémentation CORBA actuelle de Borland.

En effet, visibroker constitue le principal ORB utilisé dans le développement des applications reparties avec la technologie CORBA et dont une des principales caractéristiques est:

l'architecture Smart Agent : le SmartAgent ou OSagent de visibroker est un service de répertoire dynamique utilisé tant par les applications clientes que par les objets CORBA Il enregistre tous les objets CORBA actifs et disponibles sur le réseau et les localise pour servir

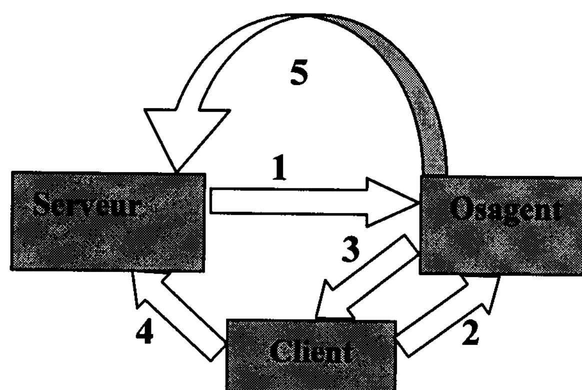


Figure 1 : *Relation entre client, serveur et Smart Agent.*

Dans la figure ci-dessus, on y trouve lors du lancement de l'application:

1. enregistrement de l'objet serveur sur l'agent visibroker
2. demande de connexion d'un client à un objet distant
3. en réponse retour de la référence à l'objet distant
4. dialogue direct entre le client et l'objet
5. accès périodiques (tolérance aux pannes)

2. LES ETAPES DE DEVELLOPEMENT D'UNE APPLICATION CORBA

Dans cette partie nous présentons les étapes à suivre pour développer une application distribuée avec CORBA.

Etape 1 : la première chose à faire est de créer l'interface IDL des objets en définissant les opérations fournies et qui seront accessibles aux applications distantes.

Etape 2 : compiler l'IDL pour générer les fichiers stub et skeleton.

Etape 3: Implémenter l'objet c'est-à-dire créer la classe de l'objet dont on souhaite mettre à dispositions des différents clients.

Etape 4 : à cette étape de développement, on passe à la réalisation du serveur qui contiendra l'objet. En effet celui-ci se charge de l'initialisation de l'ORB, du routage des requêtes et des résultats.

Etape 5 : enfin, il faut passer à la réalisation de l'application cliente et dont la logique de fonctionnement se résume:

- l'initialisation de l'ORB
- la connexion à l'objet distant
- et puis l'utilisation (appels des méthodes).

3. DESCRIPTION DE NOTRE APLICATION

Nous proposons dans cette partie de décrire notre application gestion bibliothèque mise en place. Elle est constituée des objets modélisant des livres dans une bibliothèque virtuelle accessible à distance, d'un serveur d'objets et de quelques programmes clients qui utilisent ce serveur. Ces deux parties de l'application (serveur d'objets et clients) s'exécutent au dessus du bus d'objets répartis visibroker, qui est une l'implémentation CORBA commercialisée par Borland.
Le serveur stocke les informations liée au livres (cote, titre, auteur, édition).
Pour cela notre serveur est capable de répondre à des invocations telles que:

La première fenêtre qui apparaît c'est la page d'accueil qui donne au client la possibilité d'accéder à l'application distribuée.

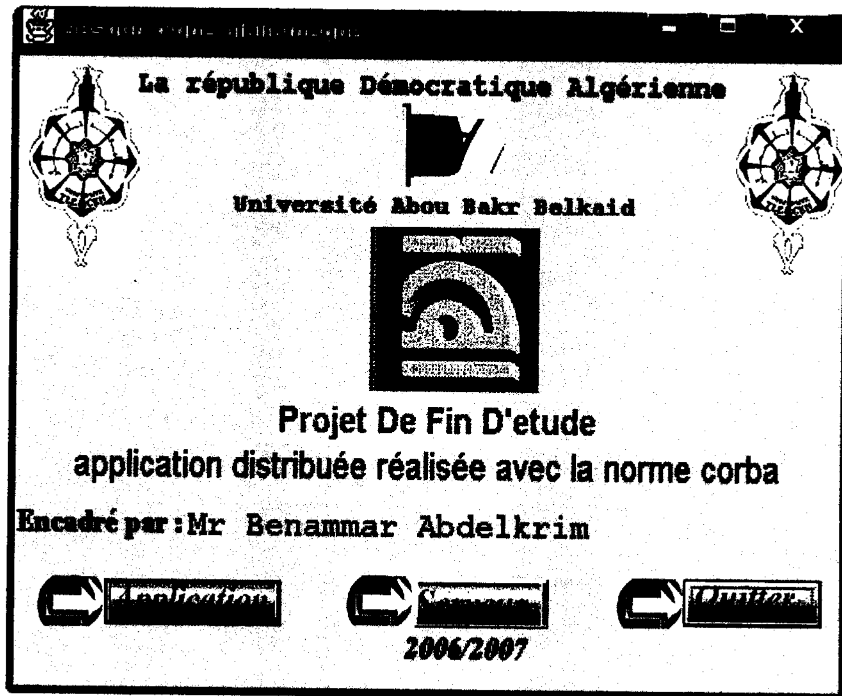


Figure 2 : *Page d'accueil*

La fenêtre suivante représente le menu de cette application, les éléments de ce menu sont accessibles par le client « le gérant de du bibliothèque ».

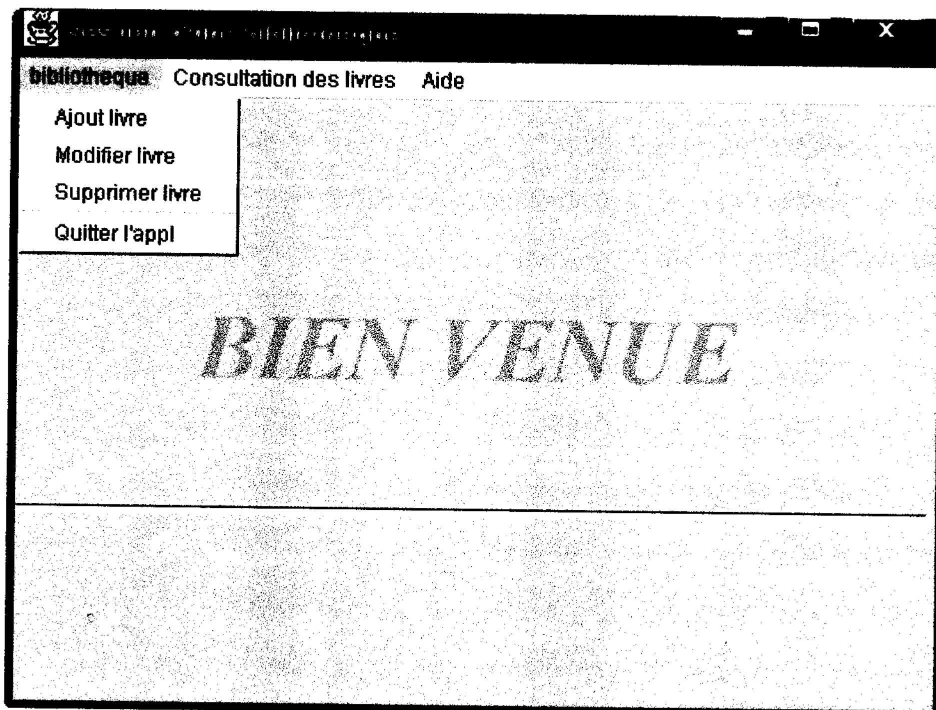


Figure 3 : *Onglet gérant.*

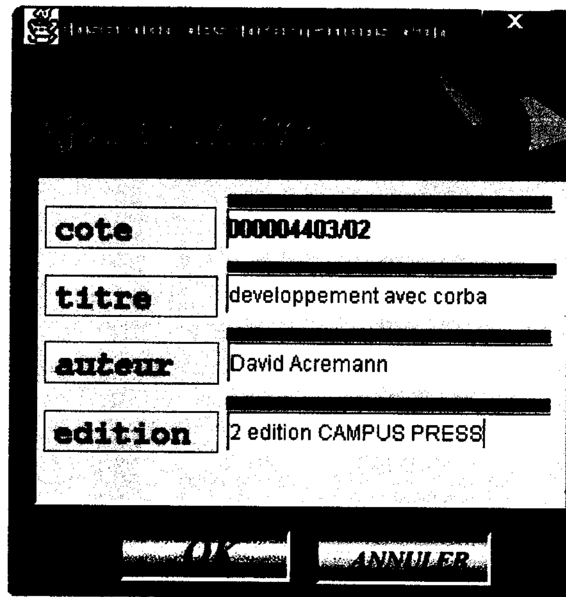


Figure 4 : *L'ajout d'un livre.*

Pour la recherche ou la suppression d'un livre il faut d'abord introduire la cote juste de ce livre et ensuite valider l'opération.

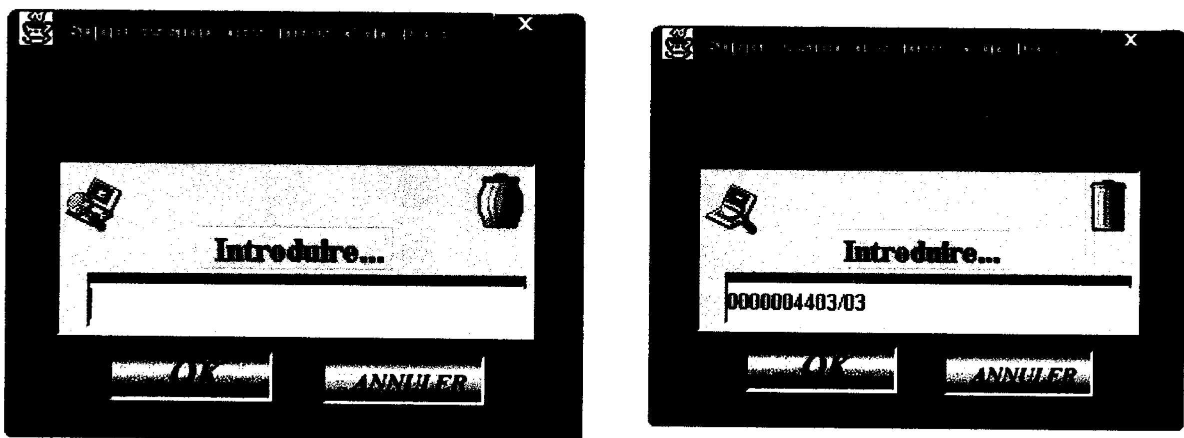


Figure 5 : *Recherche ou suppression d'un livre.*

La fenêtre suivante offre la possibilité de consultation générale ou individuelle.

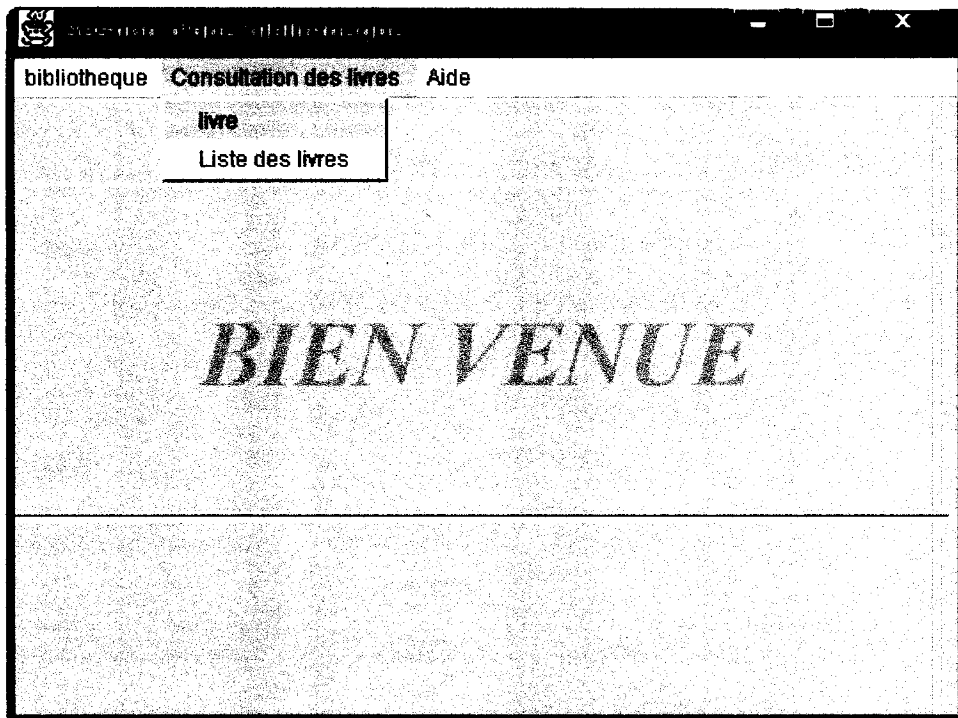


Figure 6 : *consultation générale ou individuelle.*

Si on choisi liste des livres la fenêtre suivante apparaît :

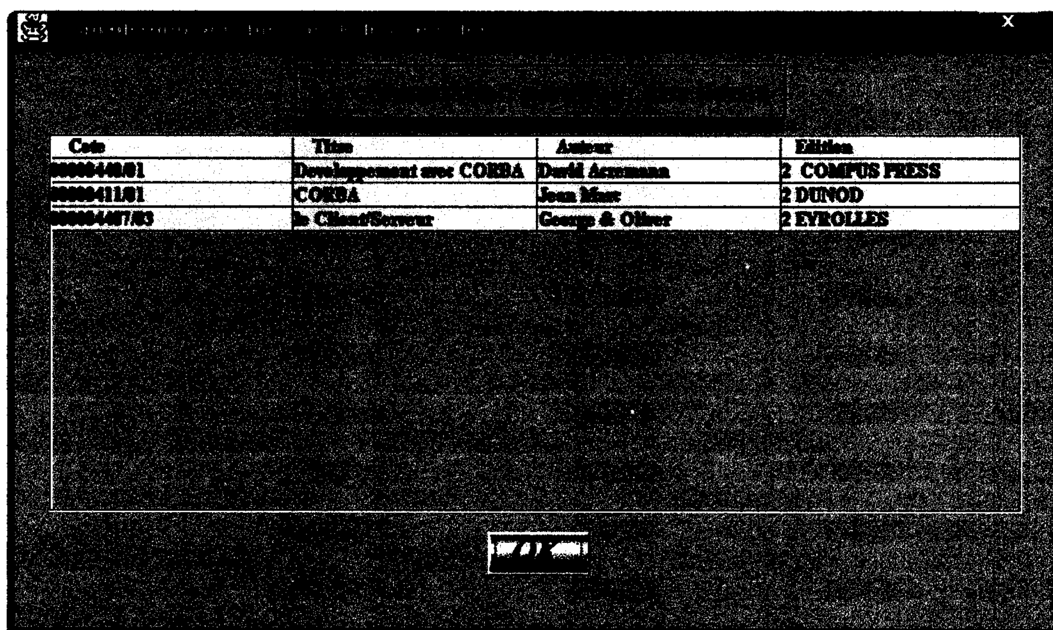
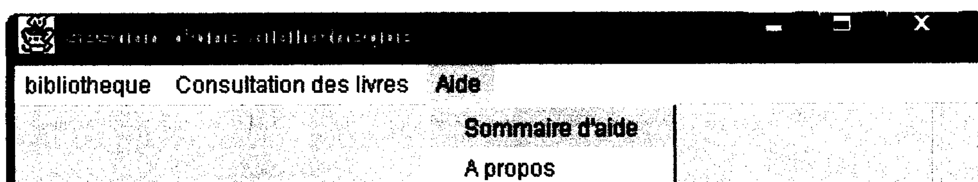


Figure 7 : *Liste des livres.*

On peut voir comme aide les fenêtres suivantes :



Il y a des messages d'exceptions qui peuvent être apparus de temps en temps dans cette application, et on peut citer comme exemple :

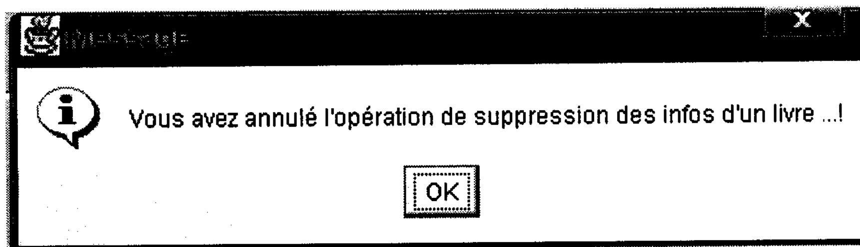


Figure 10: *Message d'exception.*

Dans tout ça, le plus important est de voir la trace laissée par des invocations au niveau du serveur distant et cela illustré par :

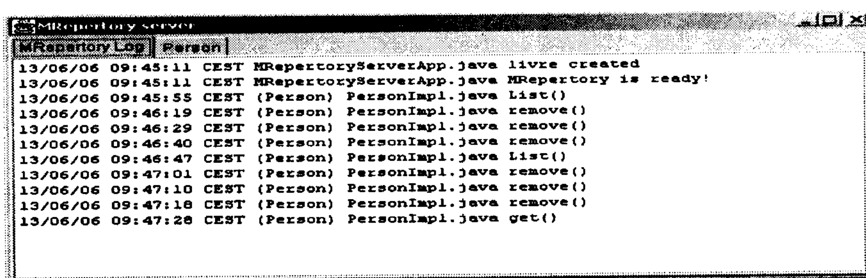


Figure 11: *Trace des invocations.*

CONCLUSION

De nombreuses autres solutions « middlewares » telles que DCE, Java et les propositions Microsoft pour la construction et l'exécution d'applications réparties existent. Cependant à travers l'application décrite, nous pouvons constater que la norme CORBA se distingue car elle offre :

· **Une solution ouverte et évolutive** : choisir CORBA revient à ne pas s'enfermer dans une solution propriétaire évoluant difficilement. L'OMG réagit très vite aux demandes du marché.

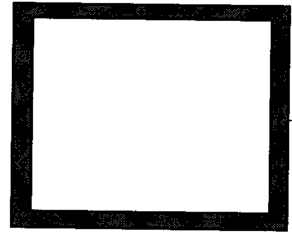
· **Une architecture modulaire** : grâce à l'OMA, une application répartie n'est pas construite à partir de zéro mais réutilise des composants logiciels offrant des fonctions orientées système, utilisateur et/ou métier.

· **L'interopérabilité entre composants hétérogènes** : CORBA fournit les abstractions et mécanismes offrant un bus orienté objet d'intégration de composants logiciels conçus avec des technologies hétérogènes (langages, systèmes d'exploitation, machines et réseaux).

· **Le libre choix des technologies d'implantation** : dans notre application, nous n'avons imposé aucune contrainte aussi bien au niveau des réseaux, des machines, des systèmes d'exploitation que des langages de programmation.

· **La portabilité du code** : les fragments de code présentés sont totalement portables d'une implantation du bus à une autre. Aucune supposition technique n'a été faite sur le bus utilisé.

· **La simplicité de mise en œuvre** : la mise en pratique de CORBA n'est pas foncièrement complexe, il suffit de se plier au modèle CORBA et à un ensemble de règles de programmation des applications.



CONCLUSION GENERALE

CONCLUSION GENERALE

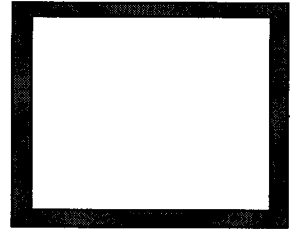
La norme CORBA ne s'est pas faite en un jour, elle est le produit d'un long travail de consensus industriel commencé en 1989. Au cours de son évolution, de nombreux ajouts et modifications lui ont été apportés. CORBA 1.0 avait pour objectif l'interopérabilité entre objets réparties. Cela fut atteint grâce aux briques de base telles que le langage OMG-IDL, l'ORB, l'adaptateur d'objets BOA et les mécanismes dynamiques (IFR, DII). Cependant, cette version ne prévoyait pas grande chose pour l'interopérabilité entre différentes implantations du bus. CORBA 2.0 avait donc comme objectif principal d'offrir cette interopérabilité via le protocole GIOP et IIOP.

En suit la version 2.2 qui corrige certaines imperfections observées et a ajouté de nouvelles composantes. En résumé, CORBA 2.2 définit :

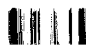
- le modèle abstrait orienté objet et client/serveur ;
- le langage OMG-IDL ;
- les briques de base telles que le module *CORBA*, les interfaces *ORB* et *Object* ;
- l'adaptateur d'objets portable ou POA ;
- les mécanismes dynamiques tels que le référentiel des interfaces (IFR), l'invocation dynamique DII, l'implantation dynamique DSI et la gestion dynamique des *DynAny* ;
- l'interopérabilité entre bus CORBA via le protocole générique GIOP et son instanciation IIOP ;
- l'interopérabilité avec le monde COM ;
- les intercepteurs pour placer des traitements globaux sur les requêtes ;

La norme CORBA 3.0 sera la prochaine révision majeure annoncée par l'OMG.

Malheureusement, ce travail ne peut pas aborder toutes ces composantes dans les moindres détails. Nous avons donc opéré une sélection sur ce qu'il est indispensable de connaître de la norme CORBA. A partir de cette information, le lecteur sera mieux armé pour étudier les composantes que nous avons laissées de côté. Cependant, il faut bien être conscient que l'OMG définit seulement des spécifications. Ces spécifications sont ensuite implantées dans des produits « à la norme CORBA ». Ainsi, actuellement, aucun produit n'implante complètement les spécifications CORBA 2.2 : chaque produit sélectionne les composantes qui lui paraissent le « plus vendeur ». Aussi, il ne faut pas hésiter à choisir le produit CORBA le plus approprié à chaque partie d'une application : un produit offrant le POA pour écrire des serveurs C++ portables et un produit offrant la projection IDL/Java pour les applications clientes utilisées à travers le WWW.



REFERENCE BIBLIOGRAPHIQUES



Références bibliographiques

[1] : Georges & Olivier Gardarin ; 1997- L e client /serveur

[2] : David, A ; Gilles, M & Laurent, R ; 1999, Développer avec CORBA en JAVA et en C++.

[3]: Jean-Marc Geib, Christophe Gransart et Philippe Merle ; Editions MASSON, 1997- CORBA : des concepts à la pratique

[4] : Robert Orfali, Dan Harkey et Jeri Edwards ; 2e édition - CLIENT / SERVEUR Guide de survie.

[5] : http://www.iona.com/products/orbacus_home.htm.

[6] : http://corbaweb.lifl.fr.

[7]: Philippe Merle; Modèle de composants CORBA.

[8] : Marc-Olivier Killijian ; Présentation de Corba , 19/02/2006.

[9] : Yamine Himeur ; Système d'exploitation (corba).

[10] Nicolas Duminil ; 2002 - J2EE avec JBuilder 7 Enterprise. Service de nommage, POA et IFR.

Richard Mark Soley et Christopher M. Stone; « Object Management Architecture Guide, revision 3.0 » OMG TC Document 97.

OMG TC Document formal/98-07-05; CORBAservices : Common Object Services Specification.

Jean-Marc Geib - Christophe Gransart - Philippe Merle Université des Sciences et Technologies de Lille Document formal 02, CORBA : des concepts à la pratique.

Quelques Site Web officiel

www.theadvisors.com/Client-Serveur.htm

www.info.uqam.ca/~obaid/obaid_ens.html

www.remi.leblond.free.fr/probatoire/node1.html

www.omg.org/corba/corbiiop.htm

ww.developpez.net/forums/archive/index.php/f-31-p-2.html

www.csee.umbc.edu/help/oracle8/java.815/a64683/appcorba.htm

http://itech.fgcu.edu/faculty/zalewski/CORBA/CORBA.html

http://www.ooc.com

Site Web officiel de la société Object-Oriented Concepts diffusant les sources de l'excellent ORBacus (ex OmniBroker) pour C++ et Java.

http://www.commentcamarche.com