MASTER EN SCIENCES INFORMATIQUES

Systèmes d'objets distribués, concepts, architectures et applications

Rymenants, Steve

Award date:
1998

Facultés Universitaires Notre-Dame de la Paix, Namur

Institut d'Informatique

Systèmes d'objets distribués
Concepts, architectures et applications

Steve Rymenants

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

Année académique 1997-1998

Table des matières

Remerciements

C'est un plaisir de remercier sincèrement toutes les personnes qui, de près ou de loin, ont contribué à l'élaboration de ce mémoire.

Tout d'abord, je tiens à remercier gracieusement Mr Jean Ramaekers, mon promoteur, grâce à qui j'ai pu effectuer mon stage chez Bull et travailler sur un sujet qui me passionnait. Ses conseils et sa patience ont été pour moi une aide précieuse.

Je remercie également François Leygues, qui m'a accueilli chez Bull et permis de vivre un stage enrichissant dont je garderai un excellent souvenir. Ses conseils et la confiance qu'il me portait, m'ont permis d'aborder ce stage d'une manière professionnelle.

Je tiens aussi à remercier du fond du cœur tous les membres de l'équipe ISM Monitor (Huan, Philippe, Pierre et Thierry) qui m'ont suivi tout au long de mon stage. Leur patience, leurs conseils, la confiance qu'ils ont su porter en moi et surtout leur bonne humeur ont largement contribué à la réussite de mon stage.

Merci à tous les membres de l'équipe ISM et en particulier à Christine, François, Olivier et Alain qui m'ont prêté une attention particulière lorsque j'en avais besoin.

Je remercie cordialement Hugues Deghorain pour son aide précieuse avant, pendant, et même après mon stage.

Merci aussi à toutes les personnes qui ont eu la patience de corriger les fautes d'orthographe de ce mémoire (Pascal, Denise, Vanessa, Lindsay et tous les autres).

Mes derniers remerciements sont destinés à mes parents qui m'ont soutenu tout au long de mes études et de la rédaction de ce mémoire.

INTRODUCTION

Depuis quelques années, on ne cesse de parler de ces fameux systèmes d'objets distribués. Qui n'a jamais entendu les noms de Corba[1], d'OLE[2] ou des RMI[3] ? Cette technologie envahit le monde informatique à une vitesse effarante. Il ne se passe pas une semaine, sans que paraisse un article sur les objets distribués dans la presse spécialisée. De nombreux informaticiens ne jurent plus que par l'utilisation de ces systèmes pour le développement des applications et la plupart des entreprises entrent, en grand nombre, dans cette danse objet.

Même l'utilisateur d'Internet n'échappe pas à cette invasion en règle. En effet, grâce aux applets Java, des objets sont téléchargés et exécutés par notre navigateur presque à chaque fois que nous consultons une page Web.

Comment en sommes-nous arrivés là ? Il y a trente ans, l'orienté objet évoquait quelques obscurs sectaires scandinaves regroupés sous le nom de Simula. Ensuite, plusieurs langages ont été créés sur base de ce concept. Ceux-ci, bien que très prometteurs, ont subit une lente évolution : la résistance au changement est malheureusement présente dans l'informatique.

Comment, pourquoi et à quel moment cette évolution linéaire est passée au stade exponentiel que nous connaissons à l'heure actuelle ? Quels ont été les facteurs qui ont permis aux informaticiens de trouver de nouvelles potentialités à ce concept ? Internet a-t-il influencé d'une manière ou d'une autre ce changement ? Pourquoi les objets distribués ont, à l'heure actuelle, autant de succès ? Et surtout, qu'est ce qu'un système d'objets distribués ? Telles sont les nombreuses questions auxquelles nous tenterons de répondre dans le premier chapitre.

Après vous avoir introduit tous les enjeux et les principes des objets distribués, la question qu'on serait en droit de se poser est de savoir si cette technologie est réellement au point. Nous tenterons de vous le prouver en vous présentant, au cours du deuxième ou du troisième chapitre, les deux courants majeurs, Corba et OLE, qui mènent actuellement à l'adoption généralisée des objets distribués comme technique de développement d'applications.

Nous examinerons également, dans un quatrième chapitre, le système RMI proposé par la société américaine Sun. Bien qu'il constitue l'outsider par rapport aux deux « mastodontes » que nous avons cités précédemment, nous pensons qu'il est assez intéressant de vous le présenter.

---

[1] *Common Object Request Broker.*
[2] *Object Linking and Embedding.*
[3] *Remote Method Invocation.*

Pour illustrer l'importance toujours croissante au sein des entreprises de cette technologie, nous terminerons ce mémoire par un exemple de développement d'un système d'objets distribués au sein de la société française Bull. J'ai eu la chance d'y effectuer mon stage de fin d'étude. Mon rôle y a été d'utiliser les SML Wrappers. Un *middleware* objet permettant la distribution, notamment sur Internet, du logiciel d'administration de Bull connu sous le nom d'ISM[1].

[1] *Integrated System Management.*

Chapitre 1

Objets distribués: une nouvelle révolution?

0. Introduction

Il n'y a pas si longtemps que cela, les architectures informatiques étaient centralisées autour de calculateurs centraux[1] de type IBM ou Bull. Nous avions dès lors droit à des applications mainframes monolithiques, difficiles à entretenir et développées le plus souvent en Cobol.
Ensuite, dans les années 90, grâce à l'importance croissante des réseaux au sein des entreprises et à la puissance de calcul devenue impressionnante des micro-ordinateurs, éclate une révolution dans le développement des applications : le client/serveur. Un client, typiquement un PC, fournit l'interface graphique, tandis que le serveur fournit l'accès aux ressources partagées, typiquement une base de données. Le client accède au serveur à travers un réseau.

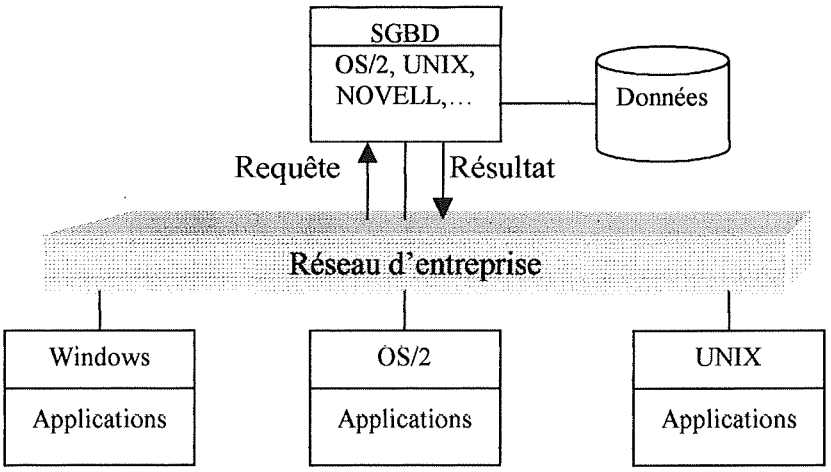


Figure 1.1. Exemple d'architecture client/serveur des années 90.

Actuellement, on assiste à un développement phénoménal des réseaux mondiaux (WAN[2]) et d'Internet, ainsi qu'à un accroissement important des performances dans le domaine des télécommunications. Cette évolution technologique permet à des milliers voire à des millions de machines dans le cas d'Internet de s'interconnecter entre elles. Dans cet environnement très ouvert et très distribué, l'architecture des applications client/serveur classiques ne suffit plus. Nous avons besoin d'une architecture plus souple, plus flexible.

[1] Appelés également Mainframes.
[2] Wide Area Network.

Ainsi, éclate une nouvelle révolution au sein même de la révolution client/serveur : les objets distribués. Ils répartissent clients et serveurs d'une application en composants « intelligents » pouvant interagir entre eux et se propager à travers les réseaux.
Les systèmes d'objets distribués constituent l'outil idéal pour nous permettre de tirer pleinement parti de cette nouvelle évolution technologique que connaît l'informatique d'aujourd'hui.

Nous tenterons tout au long de ce chapitre de vous le prouver. Mais avant cela, nous aborderons certains concepts de l'orienté objet nécessaires à la compréhension des systèmes d'objets distribués. Ensuite seulement , nous analyserons l'architecture générale de ces systèmes, ainsi que les avantages procurés par leur utilisation. Finalement, nous terminerons ce chapitre en introduisant un des domaines d'application incontournable des systèmes d'objets distribués : Internet.

1. Concepts de base de l'orienté objet

Une des premières barrières à la compréhension des systèmes d'objets distribués est constituée par le vocabulaire et les principes de l'orienté objet. Afin de franchir cette barrière, nous jugeons utile de vous les présenter .

Il faut bien noter que le but de cette section n'est pas de vous décrire un guide complet de l'orienté objet, mais simplement de vous permettre d'acquérir (si ce n'est déjà fait) les notions indispensables à la compréhension des technologies orientées objet. Nous allons tout d'abord examiner brièvement les concepts d'objet, de classe et de message. Ensuite, nous passerons en revue les principes telles que l'héritage, l'encapsulation ou le polymorphisme.

1.1. Objets, classes et messages

Un *objet* est une entité informatique, comportant un état représenté par des variables appelées *attributs* et un comportement composé de plusieurs services offerts au travers de fonctions appelées *méthodes*.
Quand on parle d'un objet, il n'est guère possible de ne pas penser à son type. Un *type d'objet* est une abstraction d'objets similaires par une interface offerte aux utilisateurs en termes de services et d'attributs.

Une *classe* est l'implémentation d'un type d'objet. Il s'agit simplement d'un modèle qui décrit et définit, dans un langage de programmation donné, les attributs et les méthodes communes à une collection d'objets similaires. Les objets appartenant à une même classe sont appelés les *instances* de la classe.

Une des notions fondamentales dans les technologies orientées objet est la notion de messages échangés entre objets. Ces messages sont de deux natures : ce sont des requêtes ou des résultats. Les méthodes d'un objet sont activées grâce à des requêtes

envoyées par d'autres objets. L'anatomie d'une requête est toujours la même : le nom de l'objet destinataire du message, le nom de la méthode à exécuter et une liste des paramètres éventuels de cette méthode.

Lorsqu'une requête est émise par un objet à destination d'un autre, le premier objet est dit client du second, lui-même qualifié de serveur. L'objet serveur renvoie le résultat de la requête au client[1].

1.2. Encapsulation

L'idée de regrouper attributs et méthodes et de séparer leur définition de leur implémentation porte le nom technique d'encapsulation. Un objet publie sous le nom d'*interface publique* la définition de ses attributs et de ses méthodes accessibles par le monde extérieur. L'implémentation[2] d'un objet n'est pas visible de l'extérieur, seule l'interface publique et, par conséquent, la définition des méthodes et des attributs est visible.
En d'autres termes, une interface d'objet est un contrat entre l'objet et le monde extérieur constitué d'autres objets.

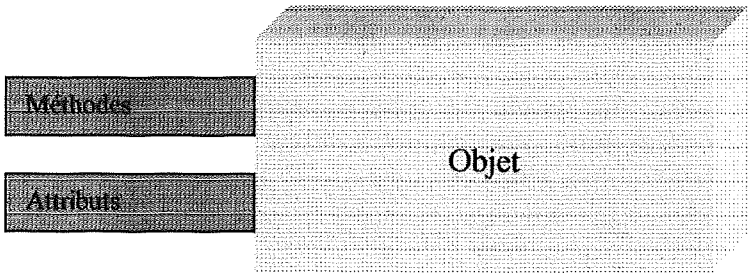


Figure 1.2. L'interface et l'implémentation d'un objet.

1.3. Héritage

L'héritage est le mécanisme permettant de créer une classe fille, appelée sous-classe à partir d'une classe parente. Les sous-classes ainsi créées héritent de tous les attributs et toutes les méthodes de la classe parente. Vous pouvez ajouter de nouvelles méthodes et de nouveaux attributs aux classes filles. Les méthodes de la classe parente ne sont pas affectées par ce mécanisme.
On peut voir l'héritage comme le fait de vouloir étendre une classe.

1.4. Polymorphisme

Le polymorphisme permet d'utiliser le même nom de méthode, et donc la même interface, pour des objets dont les implémentations diffèrent.

[1] Remarquez bien que certaines requêtes ne demandent pas l'émission d'un résultat en retour.
[2] Autrement dit, le code de la classe de cet objet.

Ces objets de classes différentes reçoivent le même message et réagissent différemment. L'expéditeur ne connaît pas la différence, le récepteur interprète le message et fournit le comportement approprié.
Le polymorphisme permet également à des sous-classes de réécrire les méthodes dont elles ont héritées et ce, sans affecter les méthodes des classes parentes.

1.5. L'invocation statique et dynamique

Il existe deux manières d'envoyer une requête à un objet, appelées respectivement *invocation statique* et *invocation dynamique*.

Dans l'invocation statique, le lien entre un objet client et l'interface d'un objet serveur est effectué à la compilation. Ceci implique que le programme client doit disposer des interfaces des objets qu'il utilise au moment de sa compilation. Il ne peut donc pas utiliser des classes d'objets qui n'existent pas au moment de son développement.

Par contre, dans l'invocation dynamique, le lien entre client et serveur s'effectue à l'exécution et non à la compilation. Ceci permet, par exemple, au client d'utiliser des classes d'objets qui ne sont pas encore implémentées au moment du développement ou de la compilation, mais qui existeront au moment de l'exécution.

2. Architecture des systèmes d'objets distribués

L'architecture des systèmes d'objets distribués est du type client/serveur mais contrairement à l'architecture client/serveur traditionnelle, les rôles n'y sont pas figés. Un objet serveur peut très bien jouer le rôle de client et inversement, un objet client peut être également un objet serveur.
Cette flexibilité est évidemment présente au niveau des applications « contenant » les objets. Elles peuvent être clientes, serveurs ou bien les deux. Cette caractéristique justifie pleinement les termes de révolution au sein de la révolution client/serveur que nous avons employés précédemment.

Dans un système d'objets distribués, les objets clients ou serveurs sont répartis dans une ou plusieurs applications sur une ou plusieurs machines. Ils ont été développés séparément, parfois par des équipes différentes et même des sociétés différentes. Dès lors, une question nous vient à l'esprit : comment les messages entre objets peuvent-ils traverser les « frontières » d'une application, circuler sur un réseau reliant des machines hétérogènes et, trouver sans erreur leur destinataire ? Pour résoudre ces problèmes, il faut mettre en place une infrastructure de distribution.

Au niveau de base, cette infrastructure doit fournir un bus à objets permettant la circulation des requêtes et des résultats entre objets locaux ou distants. Ce bus offre également des mécanismes pour permettre à des objets de se découvrir l'un l'autre.
A un niveau supérieur, l'infrastructure enrichit ce bus grâce à des services utilisables par le bus lui-même, les objets et encore les applications. Ces services sont, par exemple,

ceux de nommage, de gestion du cycle de vie, de persistance des objets, de gestion de la concurrence,...

3. Pourquoi utiliser les systèmes d'objets distribués ?

Nous vivons aujourd'hui dans un monde où la concurrence entre les entreprises est devenue impitoyable. Il est également bel et bien finit le temps où les problèmes que pouvait rencontrer une entreprise étaient stables et prévisibles.

Une des conséquences directes de cette concurrence féroce et de ce climat d'instabilité, est l'obligation pour les entreprises à une forte réactivité. Dès lors, elles ne peuvent plus admettre un temps de latence trop important entre la demande et la réalisation d'une application. Il ne s'agit plus de développer ponctuellement des applications monumentales, mais bien de mettre en place une véritable architecture permettant le développement rapide d'applications répondant aux besoins exprimés.
Ce besoin né des contraintes actuelles de l'entreprise, associé au développement des WAN et surtout d'Internet[1], peut être parfaitement satisfait par les systèmes d'objets distribués.

En effet, les objets distribués offrent de nombreux avantages dans le développement des applications et ce, notamment par rapport au client/serveur classique :

- Une fois qu'un objet aura été développé pour remplir certaines fonctions, il ne sera plus nécessaire de le réécrire. Le coût et le temps de développement des applications s'en retrouveront diminué.

- Le principe d'encapsulation des objets permettent à un programmeur d'utiliser les services offerts par un objet sans en connaître l'implémentation, ce qui permet de réduire la complexité du développement de l'application.

- L'utilisation d'objets standards pré-testés pourront rendre la maintenance des applications moins lourde et moins coûteuse.

- Les possibilités de répartition des fonctions entre clients et serveurs permettent de réduire la surcharge que pourrait subir certains serveurs comme c'est le cas dans le client/serveur classique. Cette caractéristique associée à une dispersion possible des objets à travers un ou plusieurs réseaux permet de développer des applications très flexibles et fortement distribuées.

- L'indépendance des objets distribués vis à vis des plates-formes et des systèmes d'exploitation permet le développement d'applications dont les objets seraient présents sur Windows, Unix, ...

[1] Voir section 4 de ce chapitre.

- La capacité d'abstraction et de modélisation des objets permet aux programmeurs de développer des applications d'une manière plus « naturelle », c'est-à-dire plus proche de notre manière de percevoir les choses du monde qui nous entourent.

Les objets distribués modifient fortement la manière dont nous développons des applications. Celles-ci seront bientôt reparties en une multitude d'objets situés sur différentes machines et interagissant à travers les réseaux.
Nous pensons que les objets distribués représentent la forme ultime de la distribution client/serveur et nous préparent à un probable future où des millions de machines seront à la fois client et serveur.

4. Les objets distribués et Internet

Internet est sans aucun doute le terrain le plus fertile pour les systèmes d'objets distribués. On peut dors et déjà affirmer qu'il est à la base de l'engouement pour cette technologie.

En quelques années, Internet est passé d'une aire de travail pour chercheurs universitaires à un véritable phénomène de société. Tout le monde ne parle plus que de ce réseau des réseaux, de cette autoroute de l'information. On est passé de quelques milliers d'utilisateurs à plusieurs millions en très peu de temps.
Ce succès est largement imputable à cette toile d'araignée mondiale des documents hypermédia qu'est le *World Wide Web*.

4.1. L'ère de l'hypertexte

Le Web est l'application client/serveur la plus largement déployée au monde. Dans sa première incarnation, il s'agit tout simplement d'un système d'hypertexte global. L'hypertexte est un mécanisme logiciel qui lie les documents à d'autres documents, au sein de la même machine ou à travers un réseau. Un document lié peut lui aussi contenir des liens vers d'autres documents, mais un lien peut aussi pointer vers des fichiers images, des clips vidéos, ...

Le principe du Web reste simple et repose sur quatre technologies :

- Le langage HTML[1]. Un document Web est un fichier texte dans lequel sont imbriquées des commandes HTML. Ces commandes servent à décrire la structure d'un document, définissent les liens vers d'autres pages Web ou d'autres ressources, ...

[1] *HyperText Mark-up Language.*

Les documents sont stockés sur des serveurs HTTP[1], appelés aussi serveurs Web.

- Les adresses appelées URL[2] permettent d'identifier toutes les ressources du Web : les documents, les clips, ...
  Plus précisément, une adresse URL indique la localisation d'un serveur Web et celle d'un fichier sur ce serveur. Il peut s'agir d'un texte HTML, d'une image, etc...

- Le protocole HTTP est utilisé pour accéder aux ressources Web désignées par leur URL. Il est construit au dessus de la couche TCP/IP d'Internet.

- Un navigateur Web est un client minimum qui interprète les informations qu'il reçoit d'un serveur via HTTP et les affiche sous forme graphique.

L'interaction client/serveur avec le Web est plus que limité. Voici un scénario courant:

1. Vous sélectionnez une URL cible dans votre navigateur[3].
2. Le navigateur envoie une requête HTTP au serveur.
3. Le serveur HTTP reçoit la requête, la traite et expédie le fichier HTML demandé.
4. Le navigateur interprète le fichier HTML reçu et affiche le contenu de la page.

Mais, au début des années 90, est né un langage de programmation qui allait révolutionner le Web : Java.

4.2. L'ère des objets Java

Java est un langage orienté objet proche du C++, qui a été développé par la société Sun. Tous les concepts orientés objet que nous avons abordés dans ce premier chapitre sont présents dans ce langage.
Java dispose de nombreux atouts et avantages que nous n'allons pas examiner ici. Nous nous attarderons juste sur deux éléments qui ont, à notre avis, fortement contribué au succès de ce langage, à savoir sa portabilité et les applets.

4.2.1. La portabilité

Lorsqu'on demande à un développeur de chez Sun de nous parler du langage Java, vous entendrez sûrement : « *Write once, run everywhere* ».

---

[1] *HyperText Transfert Protocol.*
[2] *Uniform Ressource Locator.*
[3] Soit vous l'obtenez en cliquant sur un lien ou en entrant explicitement l'URL, ou bien encore en la sélectionant dans une liste.

Une fois qu'un programme java est écrit, il peut être exécuté sur presque n'importe quelle machine.

En fait, Java est un langage compilé et interprété. Le compilateur Java compile le code Java en *bytecode* universel. Ce *bytecode* est exécuté par ce que l'on nomme une machine virtuelle Java. Celle-ci peut être implémentée au sein d'un navigateur Web, sous Unix, sous Windows, ...
Le code java compilé peut être exécuté par n'importe quelle machine virtuelle Java quelque soit son implémentation.

4.2.2. Les applets

Une applet est un petit programme Java qui est téléchargé et exécuté par un navigateur Web à travers Internet.

Concrètement, si on ajoute un marqueur particulier dans un page HTML en indiquant le nom du fichier contenant le code de l'applet, un navigateur téléchargeant cette page téléchargera également l'applet et l'exécutera dans sa propre machine virtuelle Java.
Grâce aux applets, on passe de pages Web relativement passives à des documents contenant de véritables programmes écrits en Java exécutés par le navigateur client.

4.3. L'ère des objets distribués

Java et ses applets ont ouvert une brèche dans laquelle Microsoft avec OLE, et l'OMG[1] avec Corba, se sont engouffrés très rapidement. Cette brèche n'est bien évidemment rendue exploitable que par l'évolution rapide des navigateurs Web permettant le transfert d'objets et leurs exécutions sur une machine cliente.

On pourra bientôt développer une application en téléchargeant à partir du Web les objets standards ou préfabriqués dont on aura besoin. On développera également de véritables applications distribuées sur Internet ; on y téléchargera des objets et ceux-ci communiqueront avec d'autres objets présents sur le Web ou sur le réseau local de la machine cliente.

Internet est devenu en quelques années le meilleur terrain de développement pour les système d'objets distribués et il est dors et déjà prévisible que ces systèmes se trouveront à peu près partout où l'on trouve actuellement une porte d'accès à Internet. On comprend dès lors les raisons de la véritable bataille en règle ayant actuellement lieu sur Internet. Microsoft essaie non seulement d'imposer son navigateur Web[2], mais également son système d'objets distribués au travers de composants téléchargeables par *Internet Explorer* : les composants ActiveX[3]. Il en va de même pour l'OMG qui s'est associé avec Netscape pour promouvoir ses objets Corba.

---

[1] *Object Management Group.*
[2] A savoir *Internet Explorer.*
[3] Voir chapitre 3, section 4.

5. Conclusion

Maintenant que nous connaissons mieux les enjeux des systèmes d'objets distribués, ainsi que les concepts de base tant au niveau architectural qu'au niveau objet nécessaires à la compréhension de tels systèmes, nous allons pouvoir examiner au cours des prochains chapitres les principaux protagonistes de cette vague objet : Corba, OLE et RMI.

Chapitre 2

Corba : Common Object Request Broker Architecture

0. Introduction

Qu'est ce qu'exactement Corba ? Pour répondre à cette question, il nous faut remonter quelques années en arrière, plus précisément en avril 1989. Cette année-là, de grandes sociétés américaines (HP, Sun, Canon, American Airlines, ...) ont créé l'*Object Management Group* (OMG).

L'OMG est un consortium sans but lucratif dont l'objectif principal est de promouvoir les technologies orientées objet dans le développement d'applications distribuées. En particulier, il vise à réduire la complexité et les coûts de développement objet, ainsi que de permettre l'introduction rapide de nouveaux types d'applications distribuées.
Initialement formé par quelques sociétés, il comprend aujourd'hui plus de 400 membres[1].

Pour atteindre ces objectifs, l'OMG doit créer des standards permettant à des applications orientées objet, développées indépendamment sur des réseaux hétérogènes, d'interagir. C'est ainsi qu'il a définit, en 1990, l'*Object Management Architecture* (OMA), une architecture logicielle pour le développement d'applications distribuées à base d'objets.
Un des composants clé de cette architecture est une spécification normative appelée Corba. Cette norme décrit un système permettant à des objets d'interagir dans un environnement distribué hétérogène.
Elle recouvre les domaines constituant l'OMA :

- Un bus à objets, appelé *Object Request Broker*[2]. Il est définit dans le document de spécification de Corba.
  Ce document fournit également :
  - Le modèle objet supportant Corba.
  - Une syntaxe et une sémantique pour un langage de définition d'interfaces : le langage IDL[3].
  - Un ensemble de projections de l'IDL vers certains langages d'implémentation.
  - Une spécification pour l'interopérabilité entre ORB.

[1] Parmi lesquels des sociétés telles que Bull, Microsoft, IBM ou encore Borland.
[2] Nous utiliserons désormais l'abréviation ORB pour plus de faciliter.
[3] *Interface Definition Language*.

- Les spécifications, appelées *CorbaServices*, de services communs nécessaires aux objets applicatifs. Ces services sont, par exemple, ceux de nommage, de cycle de vie, de persistance des objets, ...

- Les spécifications, appelées *CorbaFacilities*, de services communs nécessaires non plus aux objets mais aux applications.
  Le but de ces spécifications est de définir des collections d'objets préfabriqués pour des applications récurrentes dans l'entreprise: conception de documents, administration de systèmes informatiques, ...

- Les spécifications, appelées *Domain Services*, visent à offrir une description des objets et des services communs nécessaires à une industrie donnée.

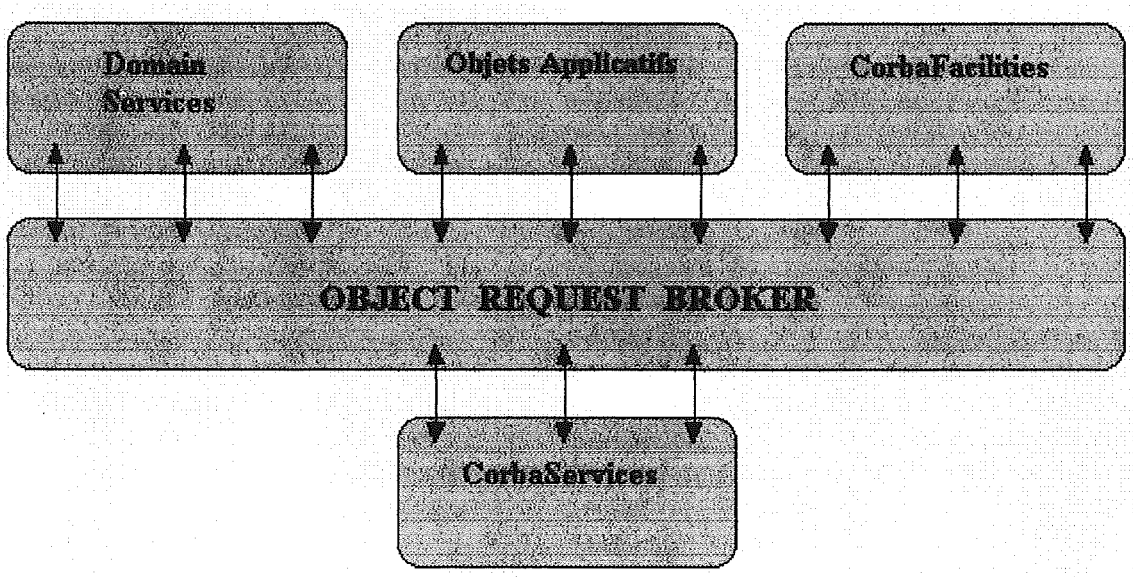


Figure 2.1. L'Object Management Architecture.

Il faut bien noter que ces spécifications décrivent uniquement la syntaxe, en IDL, des services et la description de leurs opérations.

Dans ce chapitre, nous commencerons par introduire quelques concepts de base avant d'analyser l'élément central de Corba :l'ORB. Ensuite, nous examinerons le langage IDL, ainsi que l'ensemble des services offerts par Corba (CorbaServices, CorbaFacilities et Domain Services). Nous terminerons ce chapitre par une conclusion sur Corba et son avenir.

1. Concepts de base

1.1. Les Objets CORBA

Un objet Corba est un objet pouvant se situer sur n'importe quelle machine d'un réseau supportant un ORB et pouvant être implémenté dans n'importe quel langage de programmation admis par Corba[1].
Un tel objet est identifié de manière unique au sein d'un système ORB distribué par une *référence d'objet*.

Des clients distants peuvent accéder aux méthodes d'un objet Corba. Ceux-ci ne connaissent ni la localisation de cet objet, ni son langage d'implémentation.
Cette notion de transparence est essentielle dans Corba.
La transparence vis-à-vis de la localisation d'un objet Corba est assurée par l'ORB, tandis que celle vis-à-vis du langage de programmation est assurée par un langage de description des interfaces d'objet, l'IDL.

Les définitions d'interfaces écrites en IDL informent le client des méthodes qu'un objet supporte, le type des paramètres et celui de la valeur de retour.
En connaissant la description IDL d'un objet (ou plutôt de sa classe), un programmeur client peut écrire le code de son application. Il doit pour cela utiliser les types définis en IDL au travers d'une projection vers le langage d'implémentation de son application. Cette projection définit les constructions de certains langages de programmation (types de données, classes, ...) par rapport aux constructions définies par l'IDL.

En conclusion, un objet Corba est un objet distribué décrit en IDL mais implémenté dans n'importe quel langage reconnu par Corba.

1.2. Les serveurs d'objets Corba.

Un serveur d'objet Corba est une entité informatique qui « contient » un objet Corba. Concrètement, un serveur peut être une application, une bibliothèque dynamique (DLL), ou encore même une base de données orientée objet.

Comme nous l'avons signalé auparavant, c'est l'ORB qui est chargé de garantir la transparence au niveau de la localisation de l'objet. Que celui-ci soit situé dans une application ou dans une base de données, active ou non, ne doit strictement rien changer pour le client.

[1] C'est-à-dire le C, C++, Ada, Smalltalk et Java.

2. L'Object Request Broker

L'ORB est un mécanisme permettant la circulation de messages entre objets. Ceux-ci peuvent être situés sur la même machine ou sur une machine distante accessible par l'ORB. Bref, il s'agit tout simplement du « middleware » qui établit les relations client/serveur entre les objets.

La figure 2.2 nous montre la structure d'un ORB.

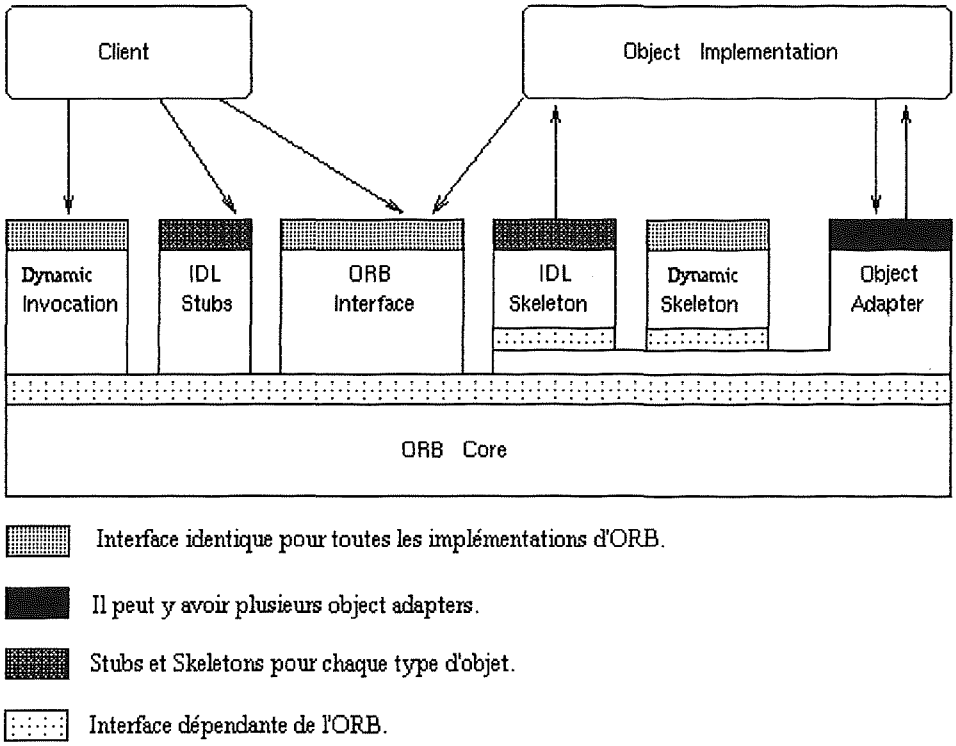


Figure 2.2. La structure d'un ORB.

Pour effectuer une requête sur un objet serveur, le client peut utiliser une interface d'invocation statique ou une interface d'invocation dynamique. Mais, il doit avant tout posséder la référence de cet objet.

2.1. Les références d'objets

Nous avons déjà vu qu'une référence d'objet est un identifiant permettant de spécifier un objet particulier dans un système d'objets distribués supportant un ORB. Autrement dit, il s'agit de l'identité d'un objet Corba qui est utilisée par les clients pour invoquer ses opérations.

La première question qui vient à l'esprit est de savoir comment on obtient une référence d'objet ? Corba offre plusieurs possibilités pour cela :

- Les clients peuvent recevoir une référence comme résultat de l'invocation d'une méthode sur un autre objet dont ils disposent de la référence.
  Dans Corba, les objets capables de créer d'autres objets, sont appelés des objets « usine ». Ils font évidemment appel à l'ORB pour ces créations.

- Les clients peuvent également faire appel à un service de nommage[1] offert par Corba. Ce service leur permet de retrouver une référence par rapport à un nom.

- En sachant qu'une référence d'objet peut être convertie en string pouvant être stocké, par exemple, sur un fichier, le client peut essayer d'obtenir un tel string et de le retransformer en référence. Ces transformations sont possible grâce à des fonctions offertes par l'ORB.

2.2. L'invocation statique

2.2.1. Du côté client

Une interface d'invocation statique n'est finalement qu'un ensemble d'opérations, fonctions ou procédures, accessibles dans le langage du client, permettant d'invoquer un objet distant de manière transparente.

Lorsqu'un client dispose d'une référence vers un objet serveur, il peut tenter d'effectuer une invocation statique sur cet objet serveur. Pour cela, il doit disposer d'un stub client.
Un stub client peut être considéré comme un objet local représentant un objet distant particulier. Il implémente les mêmes méthodes que celles présentes dans la description de l'interface en IDL de cet objet. Son but est de recevoir la requête du client, la transformer en messages pour émission vers le serveur via l'ORB et de restituer les réponses correspondantes reçues au client.

Concrètement, le client disposant de la référence d'un objet serveur, passe cette référence au stub et utilise le stub pour invoquer les méthodes voulues.

2.2.2. Du côté serveur

Du côté serveur, l'invocation statique fait intervenir les éléments suivants :

- **Le skeleton serveur.**
  Il peut également être considéré comme un objet chargé quant à lui de recevoir les requêtes du côté serveur sous forme de messages et d'effectuer la translation de ce

[1] Voir section 4.1 de ce chapitre.

format de transport en structures de données (appel de méthodes, paramètres, ...) dans le langage d'implémentation de l'objet serveur auquel il correspond. Il fournit le résultat de cette translation à l'objet serveur.
Un *skeleton* encode également les réponses sous forme de messages et les renvoient au client via l'ORB.

- **L'Object Adapter .**

L'*object adapter* est l'élément qu'une implémentation d'objet utilise pour avoir accès à un ORB et que l'ORB utilise pour gérer l'environnement d'exécution des implémentations d'objets.
Par gérer l'environnement, on entend instancier des objets, leur fournir les requêtes en collaboration avec le skeleton, activer les serveurs,...
Cet adaptateur enregistre également les classes qu'il supporte et leurs instanciations actuellement existantes dans une base de donnée appelée *référentiel d'implémentation*.
Corba spécifie que chaque ORB doit supporter un adaptateur standard appelé *Basic Object Adapter.*

2.2.3. Comment les stubs et les skeletons sont-ils générés ?

Ce sont des compilateurs IDL qui génèrent les stubs et les skeletons à partir des descriptions d'interfaces d'objet en IDL.
En effet, à partir du fichier contenant la description de l'interface d'un objet en IDL, le programmeur d'un objet serveur utilise un compilateur IDL pour générer un stub, un skeleton et une ossature de programme dans le langage d'implémentation de son choix. Il construit son programme autour de cette ossature.

Ce compilateur IDL ne génère pas que les stubs et les skeletons, il met à jour une base de données appelée *référentiel d'interfaces*[1] en ajoutant la description de l'interface en IDL de la classe nouvellement créée. Ce référentiel contient notamment les descriptions en IDL de toutes les interfaces des classes enregistrées.

2.2.4. Comment un programmeur client obtient-il un stub ?

Corba ne répond pas vraiment à cette question. Autrement dit, il ne spécifie pas la manière dont un client obtient un stub. En pratique, le programmeur serveur peut, par exemple, laisser les stubs accessibles dans un répertoire publique. Mais, la solution la plus utilisée est celle d'associer les stubs et les skeletons aux descriptions des interfaces présentes dans le référentiel d'interfaces.

[1] Voir section 2.5 de ce chapitre.

2.3. L'invocation dynamique

2.3.1. Du côté client

Pour pouvoir invoquer une méthode d'un objet serveur dont le client ne dispose pas du stub au moment de la compilation, le client doit utiliser l'interface d'invocation dynamique. Celle-ci peut se définir comme un ensemble de fonctions génériques permettant de composer des requêtes afin d'appeler des opérations dont le nom et les paramètres ne sont pas connus lors de la compilation.

Cette « *Dynamic Invocation Interface* » offre deux modes d'invocation dynamique, un mode synchrone dans lequel le client est bloqué en attendant la réponse du serveur et un mode asynchrone où le client n'attend pas la réponse du serveur et peut la demander plus tard.

Voici les fonctions de cette interface que le client doit appeler afin d'effectuer une invocation dynamique :

- Il doit obtenir la description de la méthode à partir du référentiel d'interfaces. Pour cela Corba spécifie quelques appels pour localiser les interfaces dans ce référentiel.

- Il doit ensuite créer une liste d'arguments à partir de l'appel *create_list*() et il ajoute les arguments de la méthode un à un à cette liste avec l'appel *add_arg*().

- La requête est créée en utilisant l'appel *create_request* (*Object Reference* , *Method*, *Argument List*).

- L'invocation de la requête peut se résoudre de trois manière c'est-à-dire que l'on peut utiliser 3 appels différents : 1) l'appel *invoke* pour envoyer la requête et recevoir les résultats ; 2) l'appel *send* renvoie le contrôle au programme client qui doit ensuite effectuer un *get_response* pour obtenir les résultats ; 3) l'appel *send* « *one way* » est définit lorsqu'aucune réponse n'est nécessaire.

2.3.2. Du côté serveur

L'*object adapter* intervient toujours et c'est l'interface d'invocation dynamique qui reprend, en quelque sorte, le rôle du skeleton statique. En effet, cette interface fournit le mécanisme permettant aux serveurs de pouvoir traiter les appels provenant d'objets ne disposant pas de stub.
La « *Dynamic Skeleton Interface* » regarde les valeurs des paramètres dans un message entrant et détermine l'objet et la méthode cible. Elle est également utilisée pour renvoyer les réponses au client.

2.4. L'Object Adapter

Il s'agit du mécanisme permettant à une implémentation d'objet d'accéder aux services d'un ORB. Voici quelques-uns des services fournit par l'*Object Adapter* :

- **Enregistrer les classes du serveur dans le référentiel d'implémentations.** Les classes implémentées par un serveur sont enregistrées dans ce référentiel d'implémentation. On peut d'ailleurs voir celui-ci comme une base de données persistante que l'*Object Adapter* gère. Ce référentiel contient également des informations permettant de savoir comment activer le serveur de manière correcte.

- **Instancier les nouveaux objets**. L'*Object Adapter* est responsable de la création des objets à partir de l'implémentation des classes. Le nombre d'instances créées dépend de la demande des clients.

- **Générer et gérer les références d'objets**. L'*Object Adapter* assigne des références aux nouveaux objets qu'il crée. Il est responsable du « *mapping* » entre les représentations des références spécifiques à l'implémentation et celles spécifiques à l'ORB.

- **Diffuser l'existence des objets serveurs**. Un *Object Adapter* peut diffuser les services qu'il fournit. Il est chargé de faire connaître au monde extérieur les services qu'il gère.

- **Traiter les appels entrants des clients.** L'*Object Adapter* est impliqué dans l'invocation des méthodes de l'objet serveur. Par exemple, il peut activer une implémentation et également authentifier les appels entrants.

Un *Object Adapter* définit comment un objet est activé. Il peut le faire en créant un nouveau processus, en créant un nouveau thread à l'intérieur d'un processus déjà existant, ou en réutilisant un processus ou thread déjà existant.
En fait, il faut savoir qu'un serveur peut supporter plusieurs *Object Adapter* pour pouvoir répondre à différents types de requête.

Mais, l'OMG préfère ne pas assister à une trop grande prolifération de types différents d'*Object Adapter*. C'est ainsi que Corba spécifie un *Basic Object Adapter*[1] qui peut être utilisé par la plupart des objets Corba ayant une implémentation conventionnelle.
Corba exige qu'un BOA soit disponible sur chaque ORB.

[1] Nous utiliserons dorénavant l'abréviation BOA.

L'implémentation d'un *Basic Object Adapter* doit fournir les fonctions suivantes :

- Un référentiel d'implémentations permettant d'enregistrer et d'installer les implémentations d'objets. Il fournit également des informations décrivant les classes.

- Les Mécanismes pour générer et interpréter les références d'objets, pour activer et désactiver les implémentations d'objets et les serveurs.

- Un mécanisme pour identifier le client effectuant l'appel.

- L'invocation des méthodes à travers les skeletons[1].

Le BOA peut administrer un processus serveur de plusieurs manières. Concrètement, il existe 4 politiques d'activation définissant comment les objets sont créés au sein d'un processus serveur.

1. Serveur partagé : il s'agit d'un processus serveur qui est partagé par plusieurs objets Corba. Cela signifie qu'il peut y avoir plus d'une instance d'un objet Corba particulier dans son espace d'adressage.

2. Serveur non partagé : chaque objet réside dans un processus serveur différent.

3. Serveur persistant : il s'agit d'un serveur partagé qui gère lui-même l'activation des objets.

4. Serveur par méthode : un processus serveur est lancé chaque fois qu'une requête est faite.

Les serveurs partagés et persistants sont les plus utilisés. Ils réduisent le temps d'initialisation et de chargement. Par contre, l'isolement en mémoire et la sécurité ne sont pas aussi présents que dans le cas de serveurs non partagé ou par méthode.

2.5. Le référentiel d'interfaces

Le référentiel d'interfaces[2] est en quelque sorte l'annuaire des interfaces des objets du système distribué. Toutes les interfaces, ainsi que les définitions sont sauvegardées et administrées par l'IR. Par exemple, l'ORB aura recours aux services de l'IR pour vérifier le type des paramètres passés dans les requêtes en les comparant aux définitions sauvegardées.

[1] que ce soit le skeleton statique ou dynamique.
[2] Nous utiliserons dans la suite du chapitre l'abréviation IR (*Interface Repository*).

L'IR peut également être utilisé par des programmeurs clients :

- Pendant le développement pour naviguer dans la liste des interfaces disponibles.
- Pour créer des outils de développement visuels qui aident à l'administration du code source IDL des objets.
- Pour faciliter l'installation et la distribution des objets dans une configuration donnée de réseau, ...
- Pour obtenir des informations sur l'interface d'un objet à partir de sa référence. En effet, un client peut invoquer la méthode *get_interface* sur la référence d'un objet Corba pour obtenir des informations sur son interface.

Un ORB peut posséder plusieurs IR. Chacun d'entre eux possède un identifiant unique et s'appuie sur un mécanisme de persistance et de sauvegarde des interfaces qui est laissé au choix des fournisseurs d'ORB. Par exemple, un IR peut être implémenté sous forme de base de données relationnelle ou orientée objet.

3. L'interface définition language (IDL)

Dans la norme Corba, les services sont décrits dans le langage de spécification IDL, qui est indépendant des langages de programmation choisis pour implémenter les comportements des objets.

L'IDL est purement déclaratif, il n'y a pas d'implémentation et il n'est utilisé que pour décrire les interfaces qu'implémentent les objets.

3.1. La syntaxe

Nous n'allons pas revoir toute la syntaxe de l'IDL, mais uniquement les principales caractéristiques du langage et ce, à l'aide d'un petit exemple.
Voici cet exemple :

```
Module  test {
        string Hello ;
        Interface Enter {
                        Void testmethod (in Hello item1) ;
                        Void testmethod2( ) ;
                         }
        Interface Out {
                        Void Outmethod( out long item2) ;
                        }
            }
```

Comme vous pouvez le constater, les interfaces IDL sont définies au sein d'un module. Un module détermine simplement un espace de nommage pour grouper différentes interfaces.

Les interfaces IDL quant à elles définissent un ensemble de méthodes qu'un client peut invoquer sur un objet. Une interface IDL peut également déclarer une ou plusieurs exceptions, ainsi que des attributs.

Pour la définition des méthodes, il faut spécifier le type de valeur renvoyé par la méthode, ainsi que les paramètres. Un paramètre a un mode qui est *in* ou *out* selon que sa valeur soit passée respectivement du client au serveur ou du serveur au client. Les paramètres ont également un type.
Les spécifications de ce langage contiennent une liste de types de base à partir desquels de nouveaux types peuvent être définis. (Exemple : le type Hello).

Une autre caractéristique importante de l'IDL est qu'il permet l'héritage des interfaces.

3.2. Le mapping vers les autres langages

Comme nous vous l'avons déjà dit précédemment, l'IDL ne sert qu'à décrire les méthodes d'une classe. Il faut disposer d'une projection de ce langage vers d'autres langages tels que le C++ ou Java. Cette projection permet à partir de la description IDL de construire une structure dans le langage cible. Il restera au programmeur à ajouter son code à cette structure.
Heureusement, cette projection est maintenant automatisée par des compilateurs IDL.

Une interface IDL sera projetée, par exemple, en tant que classe Java. Si une méthode en IDL est projetée en méthode Java, il en est autrement des attributs. Ceux-ci sont projetés en deux méthodes : *getAttr*() et *setAttr*(), respectivement pour consulter l'attribut et pour modifier sa valeur.

4. L'interopérabilité entre ORB

Lorsque sont apparues les premières versions des produits commerciaux implémentant les spécifications de l'IDL et de l'ORB, on se rendit très vite compte que même si elles étaient conformes aux mêmes spécifications, elles mettaient en œuvre des choix d'implémentation différents qui les rendaient souvent incompatibles.
Cette incompatibilité risquait de ruiner à terme les efforts de l'OMG pour créer une industrie où fournisseurs et consommateurs d'objets pourraient réellement se rencontrer.

C'est ainsi que l'OMG adopta un ensemble de spécifications portant sur l'interopérabilité entre ORB. Cet ensemble fut intégré dans la version 2.0 de la norme

Corba.

Dans ces spécifications, Corba définit deux manières d'unifier deux ORB, soit en leur imposant d'utiliser le même protocole de communication entre objets, soit en construisant des passerelles entre protocoles utilisés par chacun des ORB.

4.1. Les protocoles de communication

Corba définit différents protocoles de communication entre ORB : GIOP (General Inter-ORB Protocole), IIOP (Internet Inter-ORB Protocol) et des protocoles spécialisés dits ESIOP (Environment Specific Inter-ORB Protocol).

4.1.1. Le protocole GIOP

Il spécifie une syntaxe standard de transfert de données et un format de message pour la communication entre ORB. GIOP est indépendant du protocole réseau utilisé pour cette communication.
Il définit 7 formats de message qui recouvrent toutes les sémantiques des appels/réponses de l'ORB. Le format commun de représentation des données se nomme CDR[1].

GIOP spécifie également un format pour des références interopérables d'objets[2]. Un ORB doit créer un IOR dès lors qu'une référence d'objet est passée à travers un autre ORB.
Un IOR associe une collection de *profils annotés* avec la référence d'un objet. Ces profils contiennent l'information permettant de traduire un message d'un ORB à un autre.

4.1.2. Le protocole IIOP

Ce protocole spécifie simplement comment des messages GIOP sont échangés dans un réseau TCP/IP. Il définit notamment quelques primitives pour assister l'établissement des connections TCP/IP.

4.1.3. Le protocole ESIOP

Un protocole non-GIOP est dit ESIOP s'il suit les définitions de passerelles proposées dans l'architecture de communication de Corba 2.0.

Le protocole DCE[3] a été adopté comme faisant partie intégrante de la norme Corba 2.0. DCE est donc considéré comme un ESIOP particulier standardisé par la norme sous le nom de DCE-ESIOP.

[1] Common Data Representation.
[2] Nous utiliserons l'abréviation IOR (*Interoperable Object Reference*) pour plus de faciliter.
[3] *Distributed Computing Environment.*

4.2. Les passerelles inter-ORB

Corba fournit les mécanismes nécessaires pour créer des passerelles d'ORB à ORB. En fait, elle présente deux solutions : les passerelles complètes et les demi-passerelles.

4.2.1. Les passerelles complètes

Une passerelle complète relie directement deux ORB aux caractéristiques techniques différentes. La passerelle traduit les messages du premier ORB dans le format accepté par les protocoles de communication du second et vice-versa. Il y a alors dans le réseau autant de passerelles que de paires d'ORB à relier.

4.2.2. Les demi-passerelles

Dans cette solution, tous les ORB ont une demi-passerelle entre leur propre protocole de communication et un protocole de communication commun à toutes les demi-passerelles.
Lorsqu'un message part d'un ORB, il est transformé par la demi-passerelle au format commun, acheminé vers la demi-passerelle destinataire qui le transforme au format de l'ORB où se trouve l'objet destinataire.

5. CorbaServices

Pour constituer un système d'objets distribués, une infrastructure telle que l'ORB ne suffit évidemment pas. Il faut, par exemple, pouvoir sauvegarder l'état des objets en cas de désactivation du serveur ou gérer les requêtes concurrentes. Il est également nécessaire de fournir aux programmeurs la possibilité de manipuler les références d'objets sous la forme de noms facilement identifiables.

Bref, il existe de nombreux services servant de support à un système d'objets distribués et plus globalement à une architecture informatique distribuée. L'ensemble de ces services sont spécifiés par Corba sous le nom de *CorbaServices*.

Comme dans toute la norme Corba, ces services sont entièrement définis par des interfaces en IDL. En termes d'implémentation, ils peuvent être des serveurs locaux ou distants sur le réseau. De plus, ils sont généralement construits autours d plusieurs objets qui coopèrent.

Nous allons nous contenter dans cette section de décrire les services les plus importants.

5.1. La gestion des noms d'objets

Une référence rattachée à un objet n'étant pas très significative pour le programmeur, il est très vite apparu souhaitable de disposer de la possibilité de désigner

un objet au moyen d'un nom. C'est un des buts du service de nommage spécifié dans Corba. Comme nous l'avons signalé précédemment, l'autre objectif de ce service est de permettre au client d'obtenir des références d'objets.
Ce service simplifié à l'extrême permet essentiellement d'associer un nom à une référence d'objet.

Une association entre un nom et un objet est appelée lien (*name binding*). Un lien est toujours défini de manière unique dans un contexte particulier.

Un contexte est un objet qui contient un ensemble de couples <nom, référence> dans lequel chaque nom est unique.

Un contexte étant un objet, on peut le lier à un nom dans un contexte donné. On aboutit ainsi à la notion de graphe d'appellation. Il s'agit d'un graphe dont les nœuds intermédiaires sont des contextes, les nœuds terminaux des objets proprement dit, et dont les arcs représentent la relation d'appartenance.

La détermination de l'objet à partir du nom dans un contexte donné s'appelle la résolution du nom.

La gestion des noms est quand à elle assurée par un serveur de noms. Corba spécifie les fonctions offertes par cet élément :

- Lier les objets à un nom dans un contexte donné,
- Résoudre un nom dans un contexte donné,
- Détruire un lien dans un contexte donné,
- Créer et détruire des contextes,
- Parcourir un contexte.

5.2. La gestion des événements

Ce service offre les fonctionnalités nécessaires pour gérer des événements asynchrones dans un système d'objets distribués. En effet, il est parfois utile qu'un objet serveur puisse notifier à tous ses programmes clients qu'un changement d'état ou de valeur s'est produit.

Les événements sont délivrés entre objets producteurs et consommateurs par une connexion, appelée canal d'événement. Ce canal est un objet reliant un ou plusieurs objets producteurs à un ou plusieurs objets consommateurs. Il permet la circulation d'événements de producteur vers consommateur.

Il existe deux schémas de communication d'événements : le schéma *push* et le schéma *pull*. Dans le mode *push*, le producteur ou le canal émet une notification respectivement vers le canal ou vers le consommateur.
Dans le mode *pull*, c'est le canal ou le consommateur qui interroge respectivement le producteur ou le canal pour savoir s'il y a un événement en attente qui lui est destiné.

Les deux modes peuvent être mélangés pour un même canal.

Corba se contente de définir la sémantique de ces modes de communication et laisse aux implémentations le choix des mécanismes effectifs de transport des notifications. Il est bien évident que les performances de ce service sont alors entièrement dépendantes de ces choix d'implémentation.

5.3. La gestion du cycle de vie des objets

Ce service est principalement responsable de la création et de la destruction des objets. Il est également responsable de la duplication et du déplacement éventuel des objets entre leur création et leur destruction.

Dans Corba, un objet susceptible de créer un autre objet est appelé usine[1].
Cette usine est un objet associé à une classe existante afin de créer les instances de celle-ci. Elle est également responsable de la gestion des ressources nécessaires à l'objet créé, ainsi que des tâches plus administratives de notification au BOA de l'existence et de la disponibilité de l'objet, ainsi que de son enregistrement éventuel dans le service de nommage des objets.

Les objets usines sont généralement localisés dans le serveur des objets qu'il crée mais, ce n'est pas une obligation imposée par Corba.

Il existe d'autres objets permettant de copier, de déplacer et de détruire les objets.

5.4. La gestion d'objets persistants

La norme Corba précise que, du point de vue du client, les objets destinataires des messages sont toujours présents, prêts à répondre aux requêtes. Le maintien de cette illusion repose sur la possibilité de sauvegarder l'état complet des objets, entre deux requêtes du client, si l'interaction entre client et serveur est de longue durée.

Une référence à un objet doit, pour le client, toujours concerner le même objet dans l'état où il a été laissé par la dernière requête. C'est le principe de persistance.
Dans Corba, cette persistance est assurée par les services de persistance d'objets

Un objet persistant est stocké dans une base de données particulière appelé *Data Store*. Celle-ci obéit à un protocole d'accès permettant de s'y connecter, de stocker, et de retrouver un objet persistant, de se déconnecter, ... Tout objet persistant stocké dans un *Data Store* se voit attribuer un identifiant persistant afin de pouvoir le retrouver.

Deux composants permettent d'assurer les fonctions de persistance, l'un recevant les requêtes du client appelé gérant d'objets persistants, l'autre accédant à la base de

[1] *Factory* en anglais.

stockage sur le serveur appelé service de données persistantes. Ces composants peuvent être distribués entre le client et le serveur de sorte à optimiser les performances.

5.5. Les services d'administration des objets

Les deux services *Property* et *Query Services* permettent d'ordonner et d'administrer des systèmes comportant un très grand nombre d'objets.

Les *Property Services* permettent à un programme client d'associer des propriétés dynamiquement à un objet. Ces propriétés ne font pas partie de l'interface ou du type de l'objet, mais lui sont simplement associées et contiennent des informations utiles au client qui est à l'origine de cette association.

Les *Query Services* sont le cœur de l'administration de grands volumes d'objets. Il s'agit d'un langage de requêtes , qui vise à unifier les bases de données relationnelles, les bases de données orientées objet et les objets Corba. Ils s'appuient sur un nouvel objet, l'évaluateur de requête qui permet de traduire les requêtes d'un système à l'autre.

5.6. Les services de sécurité et d'authentification des objets

Bien que ces services soient assez récents dans les spécifications de Corba, ils représentent une des spécifications les plus importantes pour des systèmes d'objets distribués fonctionnant sur des réseaux.

La difficulté de la mise en œuvre de politiques de sécurité dans un système d'objets distribués est exacerbée par les caractéristiques mêmes d'ouverture de ces systèmes.
Les interactions entre objets y sont nombreuses et parfois complexes : les rôles de client et de serveur n'y sont généralement pas statiquement définis et changent au cours du temps, certains objets peuvent être plus ou moins persistants, ...

Les composants des services de sécurité et d'authentification de *CorbaServices* sont les suivants :

- Service d'identification et d'authentification pour les utilisateurs et les objets.
- Service d'autorisation et de contrôle d'accès, sur base de l'identification et de l'authentification des objets et des utilisateurs.
- Services d'audit.
- Services assurant la sécurité des communications entre objets.
- Services de cryptographie.

5.7. Autres services

Il existe de nombreux autres services tels que la gestion de la concurrence, des transactions, des relations entre objets, ...

Nous voilà arriver à la fin de ce rapide survol de la plupart des services décrits par la norme Corba. La plupart des fournisseurs de produit Corba ont tous déjà construit des ORB et des compilateurs IDL. Mais, tous n'ont pas mis sur le marché des produits qui implémentent la totalité de la norme *CorbaServices*.

6. CorbaFacilities

L'objectif de ces spécifications[1] est de mettre en œuvre les objets et les services de base décrits par CorbaServices en normalisant la conception de composants logiciels, de manière à former des solutions complètes à des problèmes d'entreprise tels que l'administration de systèmes, les interfaces utilisateurs, ...

Actuellement, CorbaServices couvre quatre grands domaines applicatifs :

- L'interface utilisateur et en particulier, les spécifications sur les documents composés.
- L'administration de l'information : Cette catégorie regroupe la modélisation, la définition, la sauvegarde et la récupération, ainsi que l'échange d'information.
- L'administration de systèmes.
- L'automatisation des flux de documents et de la circulation de l'information.

6.1. L'interface utilisateur

Dans Corba, l'interface utilisateur regroupe les notions essentielles des interfaces graphiques (systèmes de fenêtrage, émulation de terminal, ...), d'environnement de travail (administration des sessions,...), et des mécanismes d'automatisation (langages de scripts, macros, ...)

Ces services définissent la forme des interactions entre les utilisateurs et le système d'objets distribués. Ils couvrent les aspects suivants :

- Gestion de l'affichage à l'écran.
- Gestion de l'affichage des documents composés et des interactions avec les différentes parties du document.
- Aide à l'utilisateur, mécanisme d'aide en ligne ,...
- Gestion de l'environnement de travail.
- Langages de scripts pour l'automatisation des opérations.

[1] Elles sont toujours en cours d'élaboration.

6.2. L'administration de l'information

CorbaFacilities regroupe sous ce terme un nombre limité de services ou de domaines applicatifs que l'on retrouve systématiquement dans de nombreuses entreprises :

- Modélisation, création de modèles des systèmes d'information, génie logiciel.
- Sauvegarde et récupération de ces modèles.
- Echange de données et d'informations entre documents composés.
- Codage de données, définition des formats standards de codage et de représentation des données.
- Services de manipulation de dates.

6.3. L'administration de systèmes

L'administration des réseaux constituent un des domaines que les systèmes d'objets distribués tels que Corba risquent d'envahir très rapidement.
En effet, les applications de gestion des systèmes distribués utilisent généralement des protocoles de gestion de réseaux tel que SNMP[1] pour communiquer avec des agents se trouvant sur chaque machine gérée. Dès lors, pourquoi ne pas construire des applications de gestion autour d'un ORB plutôt que d'utiliser les protocoles de gestion.?

Nous n'allons pas dans ce mémoire effectuer une comparaison entre les différents protocoles de gestion et Corba. Cette comparaison nécessiterait l'écriture d'un ouvrage à part entière. Nous constatons simplement que Corba possède de nombreux avantages qui pourraient en faire à l'avenir le remplaçant des protocoles de gestion :

- Corba fournit un protocole moderne et naturel pour la représentation des entités gérées, ainsi que pour la définition de leurs services.

- Les objets gérés peuvent directement appeler la plate-forme de gestion quand ils ont quelques choses d'important à signaler.

- Le service d'événement de Corba est idéal pour la gestion des événements de gestion asynchrones distribués.

- Les autres services de Corba fournissent des fonctions très utiles qui n'existent pas pour la plupart dans les protocoles de gestion.

Certaines entreprises telles qu'IBM commencent déjà à remplacer les protocoles de gestion par un ORB. D'ailleurs, IBM développe, en ce moment même, une plate-forme d'administration de réseaux[2] reposant sur Corba.

[1] *Simple Network Management Protocol.*
[2] Connue sous le nom de *Tivoli Management environment.*

Cette plate-forme fournit des services de gestion de systèmes et de réseaux dans un environnement informatique distribué. Son architecture interne est construite autour d'un ORB.

Le produit est articulé autour de deux ensembles d'objets raccordés sur ce bus. Il y a, d'une part, les applications propres à l'administration, qui se situent sur un serveur, et d'autre part, les objets à administrer : les clients. Ceux-ci se situent sur les différentes entités à gérer.

6.4. L'automatisation des tâches

Cette partie de *CorbaFacilies* couvre tout ce qui a trait à l'exécution, au séquencement et à l'automatisation des tâches (*task management, workflow* ...).
Le document fournit par l'OMG divise ces services en un certain nombre de catégories :

- Organisation des tâches (*Workflow*) : il s'agit des services qui automatisent et formalisent la circulation des informations, des données et des documents pour structurer un ensemble de tâches.
- Systèmes d'agents qui automatisent certaines tâches répétitives.
- Administration de règles qui représentent les processus de l'entreprise.
- Services d'automatisation des opérations effectuées sur les objets.

7. Domain Services

Il s'agit des services spécialisés par grands domaines industriels. Leur élaboration se fait en collaboration avec les organismes représentant les acteurs des secteurs d'activité concernés, l'OMG apportant son bagage orienté objet et ses outils de formalisation.

Nous n'allons pas examiner ces services vu leurs spécificités par rapport aux domaines industriels.

8. Conclusion

Voilà, nous avons terminé ce chapitre sur Corba. Comme vous avez pu le constater, ce système est très complet et techniquement très complexe. Il existe maintenant de nombreuses implémentations qui commencent à fleurir un peu partout. Des sociétés telles qu'IBM ou Digital proposent déjà leur propre ORB.

Corba est certainement promis à une grande destinée mais cette destinée risque d'être compromise par un élément perturbateur : OLE.
Les arguments permettant de départager Corba et OLE, sont plus d'ordre politique et économique que d'ordre technique. Tout le monde connaît le poids de Microsoft dans l'informatique d'aujourd'hui, ainsi que ses tendances à profiter de son quasi monopole dans les systèmes d'exploitation pour micro.

Corba dispose cependant de nombreux atouts :

- Il a été développé par l'OMG, qui regroupe plus de 600 acteurs. Cette organisation dispose du bagage technique suffisant pour imposer Corba, notamment sur le Web.

- Netscape soutient Corba et rend possible le transfert d'objets Corba via les applets Java.

- Corba est une spécification ouverte et de nombreuses implémentations risquent de voir le jour.

Mais, est-ce que tous ces arguments suffiront à faire pencher la balance du côté de Corba. Nous le pensons mais seul l'avenir nous le dira.

Chapitre 3

OLE : Object Linking and Embedding

0.Introduction

OLE que l'on peut traduire par «intégrer et assembler les objets » est un standard qui n'était au départ qu'une solution ponctuelle à un problème spécifique de Microsoft (contrairement à Corba qui a été établie formellement par l'OMG).
Le problème était de pouvoir intégrer à des documents Powerpoint des graphes et des images produites avec une autre application de Microsoft, Graph.
Le mécanisme inventé fut généralisé, permettant d'intégrer dans un même document des éléments provenant des autres applications de Microsoft. Au fur et à mesure de cette généralisation, ce mécanisme devenait de plus en plus complexe et profond.

Actuellement, sous le terme OLE, Microsoft regroupe un ensemble de bibliothèques de programmation, d'outils de développement, d'applications et de conventions.

OLE est relativement complexe mais, ce n'est pas le même «genre » de complexité que dans Corba. En effet, la complexité de Corba résulte souvent de l'adoption d'un compromis entre plusieurs technologies soumises par les différents membres de l'OMG, tandis que celle d'OLE réside surtout dans l'agglomération de techniques conçues pour répondre à des difficultés ponctuelles découvertes au fur et à mesure que s'étendait le champ d'application d'OLE.

Avant de vous présenter les différentes fonctions d'OLE, deux définitions s'imposent :

- Un composant peut être perçu comme un objet typé. En effet, un composant a un type prédéfini, par exemple texte, vidéo, graphique,...
- Un document composé est un document contenant un ensemble de composants.

Voici les fonctions et les services qui constituent OLE :

- Le *Component Object Model*[1]. Il constitue le fondement d'OLE. C'est le modèle objet qui unifie l'architecture des services d'OLE.

- Les services concernant les documents composés. Ils se divisent en fonctions de stockage, d'archivage de documents composés,...

[1] Nous utiliserons l'abréviation COM pour plus de faciliter.

- Les services permettant à un programme client d'exécuter dynamiquement les méthodes d'objets résidant dans des serveurs. Ils sont regroupés sous le nom d'*OLE automation*.

- Les *OLE controls*[1], rebaptisés *ActiveX* par Microsoft en 1996. Il s'agit de l'architecture de composants de Microsoft, visant à fournir une infrastructure unique pour les documents composés et l'intégration d'objets dans des pages Web pour les applications Internet et intranet.

Dans ce chapitre, nous allons vous présenter COM et plus brièvement, sa version distribuée DCOM. Ensuite, nous vous présenterons OLE automation et ActiveX. Enfin, nous vous donnerons une description assez brève des autres services d'OLE.

1. The Component Object Model (COM)

Comme nous l'avons signalé précédemment, COM est le modèle objet de Microsoft qui unifie l'architecture des différents services d'OLE. Il définit un standard d'interopérabilité et des mécanismes qui permettent à des objets clients de se connecter à des objets serveurs, puis d'invoquer les méthodes offertes par ceux-ci.
Mais, ces objets clients et serveurs doivent obligatoirement se trouver sur la même machine. On peut donc voir COM comme un Object Request Broker pour un environnement monomachine.

Plus précisément, COM permet une interaction entre objets écrits dans des langages différents, sur des machines différentes utilisant des systèmes d'exploitation différents en définissant le format binaire des fichiers exécutables. Ces fichiers sont générés par un compilateur à partir d'un programme source écrit dans un langage de programmation donné.

Dans ce cas-ci, on distingue deux types de fichiers «exécutables». Il y a les fichiers étant effectivement directement exécutables (les .EXE) et il y a également les DLL[2]. Il s'agit d'une bibliothèque où sont définies des fonctions. Pour qu'une application puisse faire appel aux fonctions présentes dans une bibliothèque, elle doit d'abord charger celle-ci en mémoire.

Dans cette section, nous allons d'abord analyser les différents concepts de base de COM. Ensuite, nous expliquerons les services traditionnels de COM tels que la gestion du cycle de vie d'un objet, la création d'un objet,...

[1] également connus sous le nom d'OCX.
[2] *Dynamic Link Library*.

1.1. Concepts de base

1.1.1. Les Interfaces COM

Une interface COM est un ensemble de définitions d'attributs et de méthodes. Elle n'a pas d'implémentation visible et constitue une sorte de contrat entre fournisseur et consommateur de service. Les clients et les objets serveurs interagissent par l'intermédiaire d'interfaces.

En effet, les clients ne manipulent pas directement les objets: ils s'adressent à une interface. Ils n'ont dès lors pas accès à l'implémentation des méthodes offertes par un objet serveur.

Une interface COM est décrite dans un langage spécifique IDL (*Interface Definition Language*) qui est légèrement différent de l'IDL de Corba. Pour le programmeur, elle est définie comme une classe virtuelle en C++, dont il doit écrire l'implémentation de toutes les méthodes.

Le client accède à une interface COM via un pointeur. Un pointeur d'interface est un pointeur indirect sur une interface permettant au client d'accéder à celle-ci. Il fait référence indirectement sur une table de pointeurs de fonction, appelée *Virtual Table.* Ces pointeurs pointent quant à eux sur une implémentation des méthodes de l'interface.

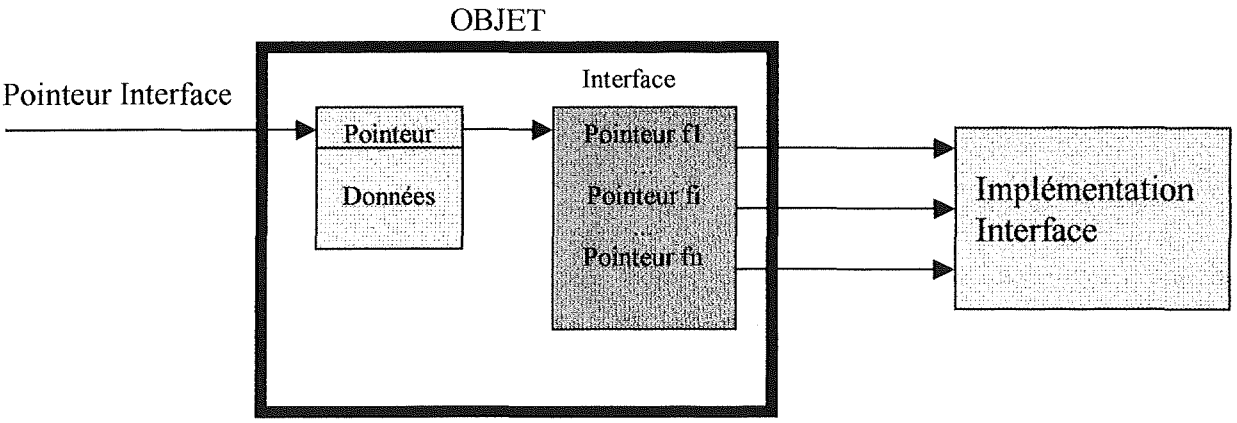


Figure 3.1. Un objet et son interface.

Contrairement à un objet Corba, un objet COM peut avoir plusieurs interfaces, chacune d'entre elles représente une vision ou un comportement différent de l'objet.

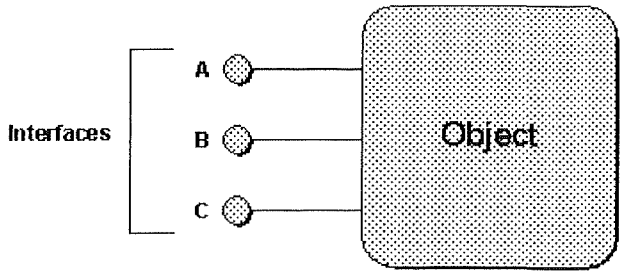


Figure 3.2. Un objet et ses interfaces.

La figure 3.3 est la représentation classique de l'accès d'une application cliente à un objet à travers une des interfaces de celui-ci.

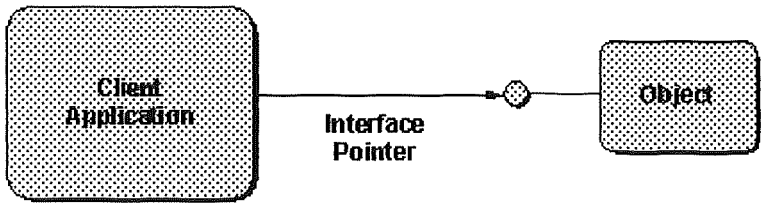


Figure 3.3. Interaction client-objet.

Chaque interface COM possède un nom commençant toujours par la lettre I et un identifiant. Cet identifiant appelé *Interface Identifier*[1] est unique au monde.
C'est la fonction COM *CoCreateGuid* qui permet de le générer. Cette fonction exécute un algorithme particulier, spécifié par l'OSF[2].

Cet identifiant d'interface permet au client de demander, de manière non ambiguë, à un objet s'il supporte l'interface désirée. Pour effectuer cette demande, le client interroge l'objet serveur en appelant la méthode *QueryInterface* de cet objet. Elle est définie dans l'interface *IUnknown*.
Tous les objets COM doivent implémenter cette interface.

Voici sa définition :

```
Interface IUnknown
{
        virtual HRESULT QueryInterface (<identifiant>, void** ppv));
        virtual ULONG AddRef ();
        virtual ULONG Release ();
}
```

Toutes ces fonctions virtuelles doivent être implémentées par le programmeur.

La méthode *QueryInterface* permet au programme client de découvrir si une interface donnée est supportée ou pas par un objet. Lorsqu'un client obtient un pointeur vers l'interface *Iunknown*, il appelle la méthode *QueryInterface* en fournissant à celle-ci l'IID de l'interface dont il a besoin. Si cette interface existe, la méthode retourne un pointeur vers celle-ci. La méthode *QueryInterface* permet ce qu'on appelle la négociation d'interface pendant l'exécution.

Les méthodes *AddRef* et *Release* permettent la gestion du cycle de vie d'un objet.

[1] Dans la suite du chapitre, nous utiliserons l'abréviation IID.
[2] *Open Foundation Software.*

1.1.2. Les Objets COM

Un objet COM est simplement une instance d'une classe d'objets COM. Une classe COM est l'implémentation d'une ou plusieurs interfaces. Elle est identifiée par un identifiant unique appelé CLSID.

Comme nous l'avons déjà signalé auparavant, un client n'interagit avec un objet COM qu'aux travers des pointeurs vers les interfaces que la classe de cet objet implémente. Remarquez qu'un objet n'a pas d'identifiant unique, vous ne pouvez pas demander à être connecter sur un objet particulier.

Tous les objets COM implémentent l'interface *Iunknown* que nous avons détaillée précédemment.

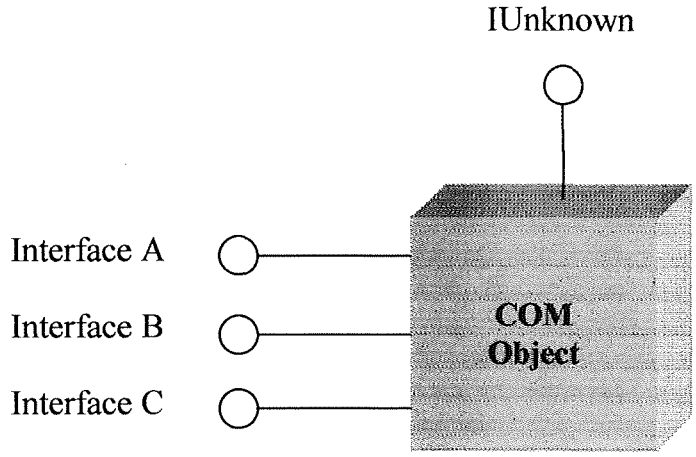


Figure 3.4. Un objet COM implémente toutes les interfaces que sa classe supporte.

1.1.3. Les serveurs COM

Un serveur COM est une DLL ou un fichier exécutable (.EXE) qui « abrite » une ou plusieurs classes COM, chacune ayant un CLSID.

Tout comme dans Corba, il faut bien faire la distinction entre serveur et objet serveur. Un objet serveur est une instance d'une classe dont les interfaces et par conséquent, les méthodes peuvent être appelées par d'autres objets. Cet objet est contenu dans un serveur.

Un serveur COM doit :

- Implémenter une classe « usine ». Pour chaque classe supportée dans le serveur, il doit y avoir une classe « usine ». Elle permet de créer les instances de la classe auxquelles elle correspond. Chacune de ces « *class factory* » implémente l'interface *IclassFactory*.

- Enregistrer les classes qu'il supporte. Le serveur doit enregistrer les CLSID de toutes les classes qu'il gère dans le registre Windows. Cet enregistrement s'effectue grâce à une API spéciale. Le serveur doit notamment fournir les CLSID, ainsi que son propre chemin d'accès (Exemple : c:\windows\test.exe). Il fournit également les IID des interfaces que la classe implémente.

  Ce registre de Windows[1] contient pour chaque classe connue du système son nom, son CLSID, le chemin d'accès du serveur correspondant, le type de ce serveur (DLL ou application), ainsi que diverses informations nécessaires à l'administration du système.

- Initialiser la bibliothèque COM. L'appel à l'API de COM *CoInitialize* permet cette initialisation. Cette bibliothèque, une fois chargée, permet d'accéder à des fonctions et des services de COM.

- Posséder un mécanisme de déchargement. S'il n'y a plus de clients pour ses objets, le serveur doit pouvoir s'arrêter tout seul. COM se chargeant de le relancer par la suite si le besoin s'en fait ressentir.

- Désinitialiser la bibliothèque COM. Le serveur appelle l'API *CoUninitialize* avant de s'arrêter.

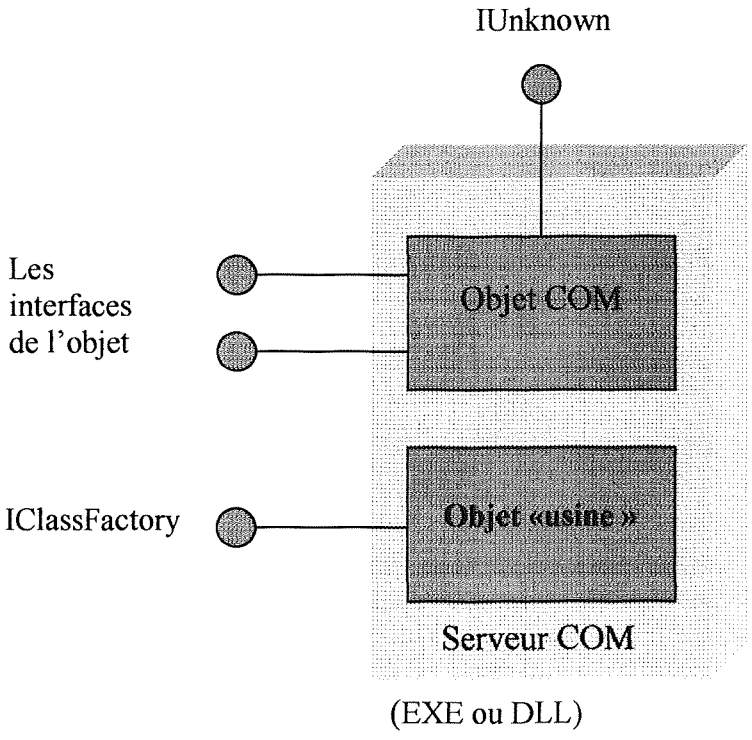


Figure 3.5. Un serveur COM.

[1] Concrètement, ce registre est un fichier système de Windows.

1.2. Gestion du cycle de vie d'un objet COM

Un objet COM est créé lorsqu'un client ayant le droit de créer cet objet le demande. Mais, la destruction d'objets COM est une autre histoire : Comment peut-on savoir qu'un objet n'est plus utilisé par un seul client ?

Pour résoudre ce problème, COM utilise un compteur de références.
Ce compteur représente en quelque sorte le nombre de clients utilisant un objet. Chaque objet COM possède son propre compteur. Celui-ci est géré par deux méthodes de l'interface *Iunknown*, à savoir *AddRef* et *Release*.
Chaque fois qu'un client obtient un pointeur vers une interface d'un objet particulier, il doit appeler la méthode *AddRef* de cet objet. Celle-ci incrémente le compteur des références de l'objet en question.

Lorsque le client en a terminé avec l'objet, il doit appeler la méthode *Release* qui décrémente le compteur de références. Lorsque ce compteur atteint la valeur nulle, l'objet est détruit par le système et disparaît de la mémoire.

1.3. Création d'objets COM

Comment créer une instance d'une classe, comment le client obtient-il un pointeur vers l'interface désirée d'un objet serveur?
Comme nous l'avons signalé précédemment, un serveur COM possède une classe « usine » pour chacune des classes qu'il contient
Elle est chargée (via une de ses instances) d'instancier la classe à laquelle elle est attachée et de renvoyer au client un pointeur vers l'interface souhaitée.
Une « *Class Factory* » doit implémenter l'interface *IClassFactory*. Cette interface définit deux méthodes :

- La méthode *CreateInstance* : elle exige comme paramètre l'IID d'une interface. Elle crée une instance de la classe implémentant cette interface et renvoie un pointeur vers l'interface correspondant à l'IID (si celle-ci existe).

- La méthode *LockServer* : elle permet de « verrouiller » un serveur d'objets. Elle offre la possibilité de conserver un objet en mémoire même si celui-ci n'est plus utilisé par un seul client.

Voici un des scénario possibles de création d'un objet COM :
Nous allons supposer qu'il existe une classe appelée **Test** enregistrée dans le registre de Windows. Cette classe implémente une interface appelée *Itest* et est définie dans un serveur local appelé Test.exe. Dans le registre Windows, au CLSID de Test est associé le chemin d'accès : c:\exemple\test.exe.
Dans ce scénario, le serveur est chargé en mémoire grâce à une fonction particulière de

COM. Cette fonction a comme paramètre le CLSID de **Test** et suite à son appel, le système COM en consultant le registre identifie et active le serveur test.exe.
Le but du client est d'obtenir un pointeur vers l'interface *Itest.*

1. Dès que le serveur test.exe est chargé en mémoire, il doit instancier toutes les « *class Factory* » qu'il contient. Il doit également enregistrer ces instances auprès du système COM. Dans cet exemple, le serveur contient uniquement la classe **Test**. Par conséquent, il ne possède qu'une classe « usine » qui est associée au CLSID de **Test**. Le serveur instancie cette classe « usine » et enregistre l'instance ainsi créée auprès de COM.

2. Le client utilise la fonction *CogetClassObject* pour obtenir un pointeur vers l'interface *IclassFactory* (de l'instance de la classe « usine » de Test) à partir du CLSID de **Test**.

3. Le client utilise la méthode *CreateInstance* de l'interface *IclassFactory* en fournissant comme paramètre l'IID de *Itest.*

4. La « class Factory » crée un instance de Test et renvoie au client un pointeur vers l'interface *Itest.*

1.4. La délégation et l'agrégation d'interfaces

COM n'autorise pas le mécanisme d'héritage d'interfaces contrairement à Corba. Microsoft justifie cette absence par la notion de fragilité des classes. En effet, si un programmeur veut sous-classer une classe donnée, il a besoin de connaître non seulement la définition de la classe mais, il peut aussi avoir besoin de connaître son implémentation, pour savoir quand déléguer ou non une opération à la classe parente. En plus, si on modifie la définition d'une classe parente, les implémentations de ses sous-classes ne sont plus correctes.

Pour pouvoir définir une interface en fonction d'autres interfaces, Microsoft introduit deux mécanismes : la délégation et l'agrégation d'interfaces.

1.5.1. La délégation

Elle permet simplement à un objet de réutiliser les interfaces (ou certaines interfaces) d'un autre objet en les appelant comme un client normal. Ceci afin de réaliser ses propres interfaces.
L'objet qui délègue est appelé objet externe, l'objet utilisé est lui appelé objet interne. Donc, l'objet externe utilise les opérations d'un objet interne pour implémenter les siennes.

Dans l'exemple ci dessous, la classe Avion possède deux interfaces : *Iavion* et *Idéplacer*. Elle délègue son interface *Idéplacer* à la classe véhicule. Après avoir géré un code spécifique pour gérer l'altitude, on peut appliquer les opérations de véhicule

directement sur l'avion.

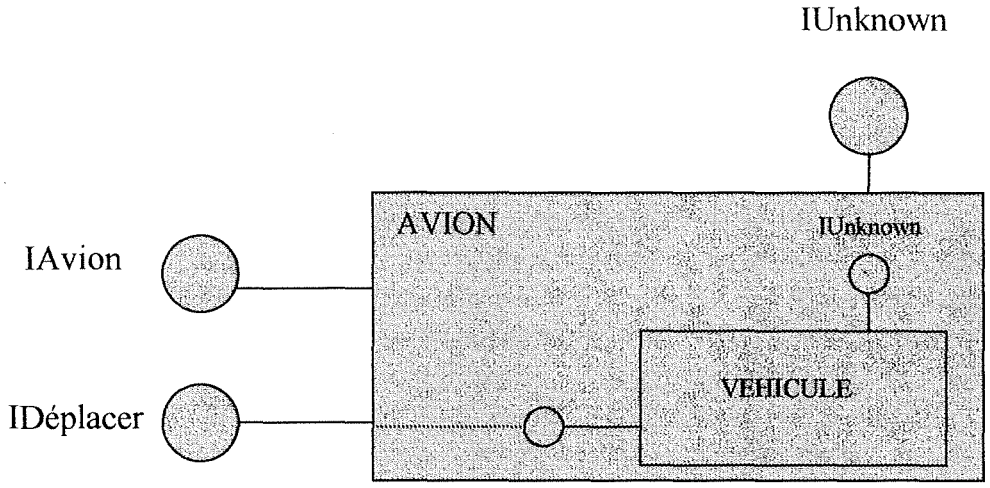


Figure 3.6. Exemple de délégation d'interfaces.

1.5.2. L'agrégation

Elle constitue un cas particulier de la délégation. En effet, un objet (dit externe) expose directement les interfaces d'un autre objet (dit interne). Tout appel à une fonction décrite par une interface de l'objet externe, résultera directement en un appel à une interface de l'objet interne. C'est une délégation sans ajout de code.

1.5. COM et la distribution d'objets monomachines

COM permet de gérer la communication entre objets, ceux-ci peuvent être contenus dans des DLL ou des applications différentes. Mais, toutes ces « serveurs d'objets » doivent se trouver sur la même machine.

Cette communication entre objets locaux est totalement transparente pour le client. En effet, un objet COM est accessible via un pointeur d'interface. Le client ne se préoccupe pas de la localisation de l'objet.

Pour localiser un serveur d'objets, le système OLE consulte le registre de Windows. Dans COM, ce serveur se situe toujours sur la même machine que le client. Une fois qu'il est localisé, il existe deux cas de figure: le serveur est une DLL ou une application.

1.5.1. Le serveur est une DLL

L'activation du serveur consiste à charger la DLL dans l'espace mémoire de l'application cliente pour permettre des appels directs entre l'objet client et les méthodes de l'objet serveur.
Ces serveurs sont appelés « in-process» car ils s'exécutent dans le même contexte que l'application cliente.

1.5.2. Le serveur est une application

Le système OLE l'active et crée (conceptuellement) du côté serveur un "*stub*" (équivalent au Skeleton dans CORBA) et un "*proxy*" (équivalent au Stub dans CORBA) du coté de l'application cliente.

Ici, le proxy ou objet par procuration est un représentant d'une des interfaces de l'objet serveur dans le contexte de l'application cliente, chargé de recevoir les appels du client, de les transmettre au serveur via le stub, de récupérer les réponses et de les retourner au client. Le pointeur d'interface retourné au client par le système OLE est en fait un pointeur vers l'interface du *proxy* correspondant à l'interface souhaitée du côté du serveur. Le proxy reçoit les appels du client, assemble les paramètres dans un message et envoie ce message via un Lightweight Remote Procedure Call (LPRC).

Le *stub* quant à lui, est chargé de recevoir des opérations sur le serveur et de communiquer les réponses au *proxy*. En fait, il transmet les appels au serveur et les réponses au proxy via LPRC.

1.5.3. Comment sont générés les proxies et les stubs ?

COM fournit les proxies et les stubs pour toutes les interfaces standards à OLE et COM. Pour développer leurs propres interfaces, les programmeurs doivent créer un fichier décrivant les méthodes d'une interface en utilisant le langage IDL de COM. Ils doivent ensuite compiler ce fichier avec un compilateur MIDL de Microsoft. Celui-ci crée le stub et le proxy de l'interface. La localisation de ceux-ci est sauvegardée dans le registre Windows.

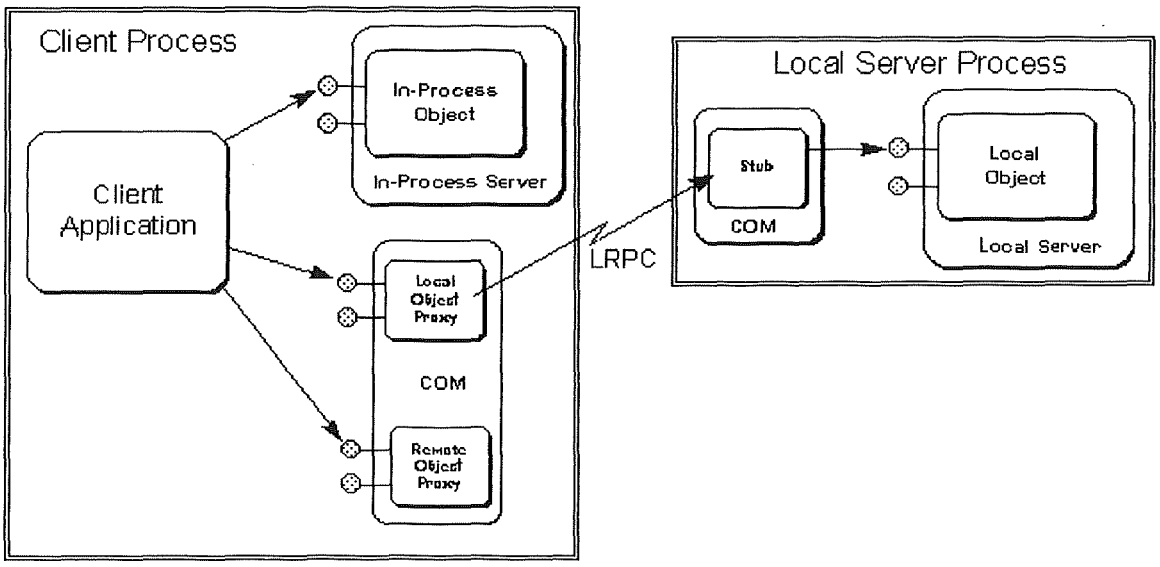


Figure 3.7. COM et la distribution monomachine.

2. The Distributed Component Object Model (DCOM)

La distribution d'objets sur plusieurs machines ne fait pas partie des spécifications de COM. Pour faire d'OLE un véritable système d'objets distribués, Microsoft a établi la spécification de DCOM, une extension au modèle COM.
Cette extension se base sur le mécanisme d'appel de procédures distantes (RPC[1]) pour la communication entre client et serveur au travers d'un réseau.
En fait, il y a toujours création par le système OLE du proxy chez le client et du stub du côté du serveur, mais celui-ci se trouve sur une machine différente.
Cette communication entre objets distants est également transparente pour le client.

Depuis quelques années, les mécanismes de RPC ont été standardisés par l'OSF, puis par l'Open Group.
Ce standard s'appelle DCE (*Distributed Computing Environment*). Sans entrer dans les détails, sachez juste qu'il définit le type des données pouvant circuler entre un appelant et un appelé, le format des paquets transitant sur le réseau, ainsi que le mécanisme d'appel de procédure. DCE définit également des services de sécurité des communications, des services de nommage, d'annuaire,...

DCOM se présente, en fait, comme une extension du standard DCE pour effectuer des appels de procédures distantes entre deux objets (ORPC[2]). Il spécifie comment ces appels sont construits et comment sont représentées et administrées les références d'objets. L'implémentation de DCOM repose donc essentiellement sur celle de DCE.

2.1. Contenu d'un message DCOM

L'en-tête d'un message DCOM est constituée d'un identifiant unique au monde appelé IPID[3]. Il représente une interface particulière sur un objet particulier dans un serveur particulier.
Après cette en-tête, il y a un autre identifiant appelé IID (*Interface Identifier*), qui indique quelle méthode de l'interface doit être appelée.

Ensuite, il y a des informations propres à DCOM. Nous n'allons pas détaillé leurs contenus mais sachez qu'elles sont appelées OPRCTHIS ou ORPCTHAT suivant que le message est un appel ou un résultat de procédure
Enfin, il y a les arguments de l'appel de procédure traduits dans un format général de DCE .

[1] *Remote Procedure Call.*
[2] *Object Remote Procedure Call.*
[3] *Interface Pointer Identifiant.*

2.2. OXID et Object Exporter

Chaque machine utilisant le protocole DCOM, possède un démon qui reçoit et émet les messages DCOM. Ce processus s'appelle l'*Object Exporter*. Il est unique sur une machine et responsable de l'administration des objets COM hébergés par cette dernière. Pour un *Object Exporter*, chaque objet du serveur est identifié par un OXID.

A chaque OXID est associé un objet COM qui porte plus communément le nom d'objet OXID. Tous ces objets OXID possèdent au moins une interface : *IremUnknown* dont nous parlerons plus tard.

Quant l'Object Explorer reçoit l'IPID d'un message, il détermine s'il existe un objet OXID qui implémente l'interface voulue. Si tel est le cas, la méthode spécifiée de l'interface est exécutée. Chaque *Object Exporter* maintient une table de correspondance entre les IPID des interfaces et les OXID des objets présents sur la machine.

Remarque : Une des possibilités offertes par le serveur en ce qui concerne l'instanciation d'une classe DCOM est celle, pour le client, de pouvoir instancier une classe, d'obtenir son OXID et d'acquérir une ou plusieurs IPID des interfaces qui intéressent le client, le tout en un seul aller et retour au niveau du réseau.

2.3. L'interface IremUnknown

Un objet COM identifié par son OXID porte une interface similaire à l'interface *Iunknown* du modèle COM. Dans DCOM, elle porte le nom de *IremUnknown* et possède les mêmes méthodes que *Iunknown* mais avec un nombre de paramètres différents.

Le système OLE définit une fonction appelée *ResolveOxid*. Cette fonction renvoie pour un identifiant donné un pointeur sur cette interface *IremUnknown* de l'objet demandé.

Dans COM, *AddRef* et *Release* sont deux méthodes appelées par le programme client pour gérer le compteur des références aux objets. Dans DCOM, ces appels répétés entre client et serveur auraient vite fait de surcharger le réseau. Pour empêcher cela,
DCOM définit deux méthodes dans l'interface *IremUnknown* : *RemAddRef* et *RemRelease*. Elles peuvent incrémenter et décrémenter le compteur des références de plusieurs objets sur le même serveur et ce, en un seul appel de procédure distante.

2.4. Le compte distribué des références aux objets

Ce mécanisme est une extension directe de celui du modèle COM.
La distribution multimachine entraîne un problème sérieux : si le client ou le serveur se déconnecte du réseau, fortuitement ou en conséquence du crash d'une machine, le client et le serveur, lorsqu'ils rétablissent leur connexion, peuvent avoir un compte des références différent pour les objets qu'ils utilisent ou qu'ils implémentent respectivement.

Pour détecter ce type d'événement, DCOM utilise un mécanisme de communication : le ping.
Les objets clients envoient à intervalle régulière, un ping aux objets serveurs dont ils ont une référence. Si un objet serveur ne reçoit plus de signe de vie du client au bout d'un certain laps de temps prédéfini, le compteur des références est diminué et la connexion avec ce client considérée comme terminée anormalement.

Il existe un problème avec cette solution : s'il y a un grand nombre d'objets distribués, le réseau risque d'être surchargé par tous ces pings. La solution prévue par DCOM est la possibilité d'envoyer un même message pouvant contenir plusieurs pings destinés à plusieurs objets sur le même serveur.

3. OLE Automation

OLE Automation permet à une application cliente d'invoquer dynamiquement des méthodes d'un objet appelé objet automate. Comme nous l'avons signalé au cours du premier chapitre, l'avantage obtenu par l'invocation dynamique est une plus grande flexibilité dans le système d'objets. Un programme peut utiliser un objet qui n'existe pas encore au moment de la compilation. Il sera découvert au moment de l'exécution.

Bâti au-dessus de COM, OLE Automation permet à une application d'en contrôler une autre en manipulant les objets automates que celle-ci rend accessible.
L'équivalent à OLE Automation dans les spécifications Corba serait à chercher dans les mécanismes d'invocation dynamique.

Il y a 3 éléments qui constituent OLE automation :

- Les objets automates.
- Les contrôleurs d'automates.
- La bibliothèque de types *(Type library).*

Nous allons maintenant analyser ces différents éléments constitutifs d'OLE automation.

3.1. Les Objets automates

Un objet automate est un objet OLE implémentant obligatoirement, en plus de l'interface *IUnknown*, une interface supplémentaire : *Idispatch.*

Cette interface fournit le mécanisme d'invocation dynamique à travers lequel l'objet peut exposer ses fonctions et ses attributs.

Elle est définie de manière suivante :

```
Interface IDispatch
{
        virtual HRESULT GetTypeInfo ();
        virtual HRESULT GetTypeInfoCount ();
        virtual HRESULT GetDsOfNames ();
        virtual HRESULT Invoke ();

}
```

- La méthode *GetTypeInfo* renvoie des informations (les noms, les paramètres, ...) concernant les fonctions et attributs que l'objet serveur décide d'exporter. Ces informations sont contenues dans ce qu'on appelle une bibliothèque de types.
  Une bibliothèque de types est attachée à une classe d'objets automates.

- La méthode *GetTypeInfoCount* renvoie 1 ou 0 suivant qu'il existe ou non une bibliothèque de types associée à un objet automate particulier.

- *GetIDofNames* fournit un identifiant unique d'une méthode ou d'un attribut d'une classe d'objets automates à partir du nom de cette méthode ou de cet attribut fourni comme paramètre.

- La méthode *Invoke* a comme paramètre l'identifiant (appelé *dispID*) renvoyé par *GetIDofNames*. Elle se charge de la correspondance entre cet identifiant et l'appel de la méthode ou l'accès à l'attribut souhaité.

Remarque : Pour qu'un objet expose des méthodes ou attributs pouvant être invoqués dynamiquement, le programmeur doit décrire les interfaces implémentées par la classe de l'objet en ODL[1] et il doit aussi y dire grâce au mot-clé *dispInterface* quelles sont les méthodes et attributs qu'il rend accessible dynamiquement. Il doit également donner pour chacune de ces méthodes un *dispID*.
Cette description ODL permet, grâce à un compilateur spécial, de générer la bibliothèque de types.

Comment un client peut-il invoquer de manière dynamique une méthode qu'un objet automate expose? Voici la description de ce qu'un client d'OLE automation peut faire, en sachant que le système OLE est initialisé:

1. Créer une instance de la classe d'objets automates. Le client crée cet objet automate grâce à l'appel *CoCreateInstance* et il fournit le CLSID de la classe

[1] *Object Description Language*.

comme paramètre. L'appel renvoie un pointeur vers l'interface *Iunknown* de l'objet automate.

2. Obtenir un pointeur vers l'interface *Idispatch* de l'objet automate. Le client fait appel à la méthode QueryInterface à partir du pointeur de l'interface *Iunknown*. Si *Idispatch* est implémentée, la méthode renvoie son pointeur d'interface

3. Obtenir des informations sur les méthodes ou les attributs souhaités. Pour cela, il fait appel à la méthode *GetTypeInfo* de l'interface *Idispatch*. Cette méthode renvoie au client les informations sur les attributs et les méthodes disponibles.

4. Obtenir un *dispID* à partir d'un nom de méthode ou d'attribut. Le client exécute pour cela la méthode *GetIDsOfNames*.

5. Invoquer. Le client utilise la méthode Invoke pour exécuter une méthode ou pour lire ou modifier la valeur d'un attribut à partir du *dispID* obtenu précédemment.

3.2. Les contrôleurs d'automates

Ce sont généralement des langages de scripts tels que Visual Basic. Ils permettent de piloter les objets automates.

3.3. La bibliothèque de types

Une bibliothèque de types contient la description d'une classe d'objets automates. Par description, on peut entendre le nom des méthodes, les attributs, les paramètres des méthodes, ...Comme nous l'avons signalé précédemment, la bibliothèque de types est générée à partir d'une description ODL. Ces bibliothèques sont généralement sauvegardées dans des fichiers ayant l'extension .TBL

Elles sont inscrites dans le registre d'OLE. Le client peut dès lors obtenir le chemin d'accès du fichier .TBL contenant la bibliothèque souhaitée.
Une fois obtenu, il utilise une API particulière (*LoadRegTypeLib*) pour charger en mémoire la bibliothèque. Cette fonction renvoie un pointeur vers une interface appelée *ItypeLib*. A partir de ce pointeur, on peut naviguer dans toute la bibliothèque. Comme vous l'avez sans doute remarqué, la « *type library* » est accessible en tant qu'objet par les applications.

4. Les composants OLE : ActiveX

Diviser une application en composants est une idée qui s'est exprimée au fur et à mesure de la maturation des produits de Microsoft. Il y eut plusieurs tentatives pour la généraliser dans Windows.

La première tentative marquante est celle du Visual Basic et de son interface

d'extension (VBX). Mais, le problème était que les composants VBX étaient très dépendants des versions de Windows et l'introduction de Windows NT et 95 mettait en danger cette architecture. COM et OLE automation furent dès lors utilisés pour construire une architecture de composants (*OLE Controls*).

La première version des composants avait pour nom OCX, mais ils ne pouvaient être distribués sur plusieurs machines. La seconde version appelée ActiveX permet à ces composants d'être distribués sur plusieurs machines. Cette version des composants repose sur DCOM.
L'usage essentiel de VBX,OCX et ActiveX est la création de documents composés. Un document composé est produit en commun par plusieurs applications, chacune est chargée d'un constituant du document.

Un composant ActiveX peut être considéré comme un objet automate mais il est plus que cela. En effet, il doit pouvoir recevoir des messages venant du document le contenant, mais il doit aussi pouvoir répondre à ces messages.
Il implémente deux séries d'interfaces distinctes : les interfaces OLE automation, pour pouvoir être piloté par le document qui le contient, et des interfaces « spécifiques » lui permettant d'envoyer des messages à ce document.

4.1.Le contenant ActiveX

Un document peut contenir un ou plusieurs composants ActiveX. Le document joue le rôle de contenant tandis que le composant joue celui du contenu.
Le contenant (qui est finalement un objet) implémente une interface très importante : *Idispatch*.

Cette interface permet aux composants ActiveX d'avoir accès aux propriétés ambiantes. Ces propriétés sont celles partagées par tous les composants ActiveX du document. Elle permet également au contenant de gérer les événements reçus du composant.

4.2. Le composant ActiveX

Les composants ActiveX ont comme caractéristiques leurs propriétés et les événements qu'ils reçoivent ou qu'ils émettent à destination de leur contenant (généralement un document).

4.2.1. Les propriétés

Il existe quatre catégories de propriétés pour ces composants :

- **Les propriétés ambiantes** : elles sont issues du document et communes à tous les composants qui le constituent. Un composant ActiveX doit obligatoirement utiliser toutes les propriétés ambiantes définies par son contenant.
  Si, par exemple, l'utilisateur change la police de caractère d'un document,

les composants doivent tenir compte de cette modification. Pour cela, le document utilise une des méthodes de l'interface spécifique *IOLEControlSite* implémentée par tous les composants ActiveX.

- **Les propriétés standards:** elles définissent le composant ActiveX et son apparence à l'intérieur d'un document. Exemple : couleur, libellé, ...

- **Les propriétés spécifiques** : elles concernent les fonctions et le comportement du composant. Ces propriétés sont définies par le développeur et sont indépendantes du document contenant le composant.

- **Les propriétés étendues** : Elles sont très particulières car leurs valeurs sont contrôlées par le contenant du composant.

4.2.2. Les événements

Les événements constituent en quelque sorte le moyen de communication du composant vers le document auquel il appartient. L'interface *Idispatch* est implémentée par le contenant et permet la gestion des événements. Ceux-ci sont divisés en un ensemble standard de catégories défini par Microsoft.

Cet ensemble standard est implémenté par tous les composants et les contenants ActiveX. A cela, s'ajoute un mécanisme d'extension, permettant à un contenant de découvrir les événements spécifiques générés par un composant donné.

Microsoft définit 4 grandes catégories d'événements caractéristiques des composants ActiveX :

- **Les requêtes** : ce sont des événements envoyés par un composant à son contenant pour l'autoriser à poursuivre et à commencer une opération telle que sa fermeture.

- **Les événements préopératoires** : ils sont envoyés par le composant à son contenant avant d'entamer une opération. Contrairement aux requêtes, le contenant n'a aucun contrôle sur l'exécution de l'opération.

- **Les événements postopératoires** : ils sont générés par le composant vers son contenant pour l'avertir de la fin de l'exécution d'une opération.

- **Les événements opératoires** : ces événements constituent le noyau du partage des tâches entre contenant et composant. Ils peuvent être gérés par l'un, par l'autre ou même par les deux. Ce sont des événements tels que les clics de souris, les pressions de touches du clavier,...

4.3. ActiveX et Internet

Tous les éléments que nous venons de vous présenter étaient déjà présents dans la première version des composants OLE (OCX).
ActiveX apporte une grande nouveauté : la distribution via Internet. En effet, un composant ActiveX est transférable d'un serveur Web vers un poste client.
C'est un élément très important dans la stratégie de Microsoft.

Comment sont-ils transférés d'un serveur Web à un poste client ?
En fait, les composants ActiveX sont signalés par un libellé particulier dans une page HTML. Il s'agit du libellé <OBJECT>. Internet Explorer interprète le libellé <OBJECT> comme une instruction d'insertion du composant OLE défini notamment par un CLSID.

Les composants ActiveX sont comparables aux applets Java. Une fois téléchargé sur une machine via le protocole HTTP, le composant ActiveX peut communiquer avec des serveurs OLE distants (via DCOM) ou présents sur la machine[1].
Il faut remarquer que ces composants sont dépendants de Windows, il faut avoir Windows comme système d'exploitation pour pouvoir les utiliser.

Finalement, les composants ActiveX se positionnent comme un fondement possible des futures interfaces graphiques et interactions homme-machine. Les applications risquent de disparaître au profit de documents constitués de composants qui capturent non seulement l'apparence visuelle mais également un comportement fonctionnel.

5. Les autres services d'OLE

Ces services s'élèvent sur une même fondation, celle du modèle COM. Ils sont à la fois nécessaires pour l'implémentation des composants et des contenants d'Activex, mais en même temps destinés à voir leur importance décroître au fur et à mesure qu'ActiveX occupe le devant de la scène, éclipsant ces services de plus bas niveau.

Ces services couvrent des domaines assez similaires à ceux qui constituent *CorbaServices*, sans en avoir la généralité. Ces différents services utilisent des interfaces que nous ne mentionnerons pas ici.

5.1. Le stockage d'objets composés

Les *fichiers composés* sont le pendant des documents composés dans le modèle de stockage. Un document composé constitué, par exemple, de trois objets différents, sera stocké dans un fichier composé (*storage*), constitué de trois flots de données (*streams*), un pour chaque objet. Tout se passe comme si le fichier composé était à lui seul un répertoire dans lequel chaque objet constituant était sauvé indépendamment dans un fichier particulier. Chaque objet contrôle le flot qui lui est attribué dans un fichier

[1] Ce que ne permet pas une applet Java.

composé, et peut y lire et écrire toutes les données nécessaires, indépendamment des autres objets qui partagent ce fichier composé.

5.2. Lien et Encapsulation d'objets

L'un des aspects les plus frappants des documents OLE est l'édition *sur place*. Dans un document composé OLE qui contient un certain nombre d'objets, un double-clic sur l'un d'entre eux a pour effet de remplacer la barre de menus du document par celle de l'application qui est responsable de l'objet sélectionné et de permettre de le modifier sur place, dans le document qui le contient. Ces objets que l'on peut modifier sur place sont des objets *encapsulés* (*embedded*, le "E" de OLE). Toutes les données nécessaires à un objet encapsulé sont stockées avec le document qui le contient.

Par contre, un objet est *lié* à un document lorsque le document ne stocke qu'une référence, ou un lien (le "L" d'OLE), à l'objet plutôt que l'objet lui-même. Dans ce cas de figure, un double-clic sur un objet lié lance le serveur de l'objet correspondant où s'effectuent les modifications avant de revenir au document initial. Ce lien qui est stocké avec le document s'appelle un *moniker*.

5.3. Transport uniforme de données

Les anciennes incarnations de l'interface de programmation Windows permettaient d'échanger des données entre applications par l'intermédiaire du presse-papiers, qui mémorisait les données pendant les opérations de copier-coller. Avec l'introduction d'OLE 2, ces mécanismes sont implémentés par des interfaces particulières qui unifient le transport des données entre applications, que ce soit par *copier-coller* ou par *glisser-déposer*. Alors qu'auparavant ces opérations se traduisaient pour le développeur par deux parties distinctes dans le programme, ces nouvelles interfaces simplifient le développement en centralisant tout ce qui a trait au format de données et à leur transport dans diverses interfaces.

6.Conclusion

La technologie OLE est clairement au cœur de l'évolution de Windows. En fait, Microsoft a dévoilé en 1996, une stratégie visant à intégrer complètement le modèle d'objets d'OLE et le modèle de distribution de l'Internet dans son système d'exploitation Windows 98. Il est également important de rappeler l'importance d'OLE dans les produits serveurs de Microsoft.

Comme vous pouvez le constater Microsoft accorde une grande importance à son système d'objets distribués. Que ce soit sur Windows ou sur Internet avec les composants ActiveX, cette société met tout son poids technologique et ce, pour rattraper le retard qu'elle avait sur l'OMG. En effet, on peut estimer que Microsoft avait accumulé un retard de plus ou moins 2 ans par rapport à l'OMG.

Microsoft est maintenant un acteur devenu essentiel dans le monde des objets distribués. OLE est le seul système à pouvoir réellement se mesurer à Corba.

Chapitre 4

RMI : Remote Method Invocation

0. Introduction

Nous avons déjà évoqué au cours de cet ouvrage le cas du célèbre langage Java[1]. Rarement dans l'histoire de l'informatique, un langage de programmation a connu pareil succès. Il y a de bonnes raisons à cela. La portabilité des applications Java est un atout important dans un monde multi-plateforme ; la possibilité de télécharger ses applets via le Web en fait un médium idéal pour le développement d'applications sur Internet ; sa facilité d'utilisation est remarquable comparée à d'autres langages orientés objet tels que le C++.

Face au développement des objets distribués, Sun ne pouvait pas ne pas profiter du succès de son langage pour développer son propre système d'objets distribués s'appuyant directement sur les objets Java.
C'est ce qu'ils firent en développant un mécanisme appelé *Remote Method Invocation*[2].

Les principaux objectifs de ce mécanisme Java sont les suivants :

- Permettre l'invocation de méthodes d'objets Java entre différentes machines virtuelles Java.
- Intégrer un modèle d'objets distribués aussi conforme que possible au modèle objet Java existant.
- Minimiser la complexité du développement d'applications distribuées en Java.
- Préserver les caractéristiques de sécurité propres à ce langage.

On peut considérer RMI comme un ORB dans la mesure où tout deux ont pour objectif principal le support d'invocation de méthodes sur des objets distants. Mais, RMI est natif à Java. Il constitue une extension de ce langage et repose par conséquent sur les caractéristiques de celui-ci.

En plus de permettre l'invocation d'objets Java distants, RMI offre, de manière transparente pour les clients, les services suivants :

- Gestion de la mémoire grâce au *Garbage Collector*.
- Réplication d'objets distants.
- Activation d'objets persistants.

[1] Voir chapitre 1, section 4.
[2] Pour plus de facilité, nous utiliserons toujours l'abréviation RMI.

Au cours de ce chapitre, nous examinerons les principaux concepts du modèle d'objets distribués Java servant de support à RMI. Nous nous attarderons ensuite sur son architecture, ainsi que sur ses caractéristiques les plus importantes.

1. Le modèle d'objets distribués Java RMI

1.1. Objets, invocation et interfaces distantes

Dans la terminologie Java RMI, un objet distant est un objet dont les méthodes peuvent être invoquées depuis une autre machine virtuelle Java, localisée soit dans un processus différent sur la même machine ou sur une machine distante. Un objet de ce type est décrit par une ou plusieurs interfaces distantes qui déclarent ses attributs et ses méthodes.

Une invocation distante est l'action d'invoquer une méthode d'une interface distante d'un objet distant. Elle a la même syntaxe qu'une invocation sur objet local.

Pour pouvoir exécuter une méthode d'un objet distant, un client doit disposer d'une interface distante implémentée par la classe de cet objet et décrivant la signature de la méthode en question. Les clients d'objets distants n'interagissent qu'avec des interfaces distantes, jamais directement avec leurs implémentations.

1.2. Le passage des paramètres

Les arguments ou la valeur de retour d'une méthode distante peuvent être des types simples de Java, des objets locaux ou des objets distants.

Un objet local, qui est passé comme paramètre ou comme résultat d'une invocation distante, est toujours passé par *copie*. Pour pouvoir transmettre la copie d'un objet, RMI utilise un mécanisme appelé sérialisation d'objet permettant à des objets Java d'être transmis entre espaces d'adressage.

Si un objet distant est passé comme paramètre, c'est le stub qui est passé. Tout comme dans Corba, un stub est un objet local représentant l'objet distant[1].

1.3. La recherche d'objets distants

Pour localiser un objet distant, RMI offre aux clients un service de dénomination des objets très similaire à celui de Corba. Ce service repose sur un serveur de noms qui maintient simplement une liste de correspondance entre références d'objets et noms d'objets. La nommage de ces objets repose sur le système d'URL[2] rendu populaire par Internet (Exemple : http:\\java.sun.com*Monobjet*).

[1] Voir page suivante.
[2] *Uniform Resource Locator.*

Pour obtenir la référence d'un objet distant, le client interroge simplement ce service en fournissant le nom de l'objet. RMI offre bien évidemment la possibilité d'obtenir une liste de noms d'objets Java distants présents sur une machine particulière.

2. Architecture

L'architecture des RMI est structurée autour de trois couches de services formalisés, définies par leurs interfaces : la couche *stub/skeleton*, la couche des *références distantes* et la couche *transport* .

Chaque niveau est indépendant des autres et peut, par conséquent, être remplacé par une implémentation différente sans affecter les autres couches. Par exemple, nous pourrions remplacer la couche transport layer actuelle basée sur TCP par UDP sans modifier les autres couches. La Figure 5.1 présente l'architecture de RMI :

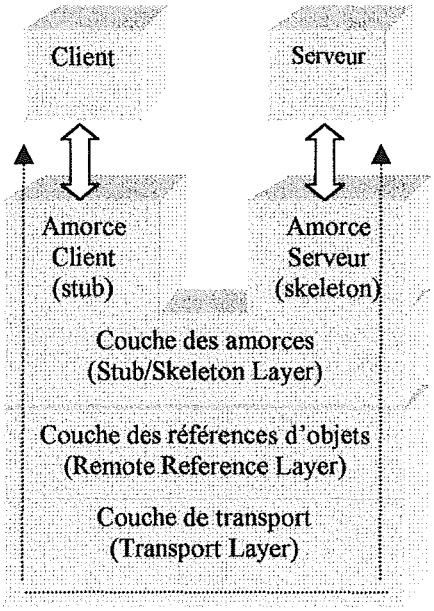


Figure 5.1. Architecture de RMI

L'appel du client à une méthode du serveur passe à travers chaque couche avant d'arriver au serveur.

2.1. Couche d'amorces (Stub/Skeleton Layer)

Cette couche est l'interface entre l'application et le système RMI.
Comme nous l'avons signalé précédemment un *stub* est un objet local représentant un objet serveur distant. Il implémente toutes les méthodes supportées par l'implémentation de l'objet distant qu'il représente.
Un client invoquant une méthode d'un objet distant utilise un *stub* comme un conduit vers l'objet distant.

A Chaque référence d'objet distant que le client manipule, correspond une référence de stub. Le client peut dès lors effectuer une invocation distante, celle-ci étant transmise par le stub à la couche inférieure de gestion des références d'objets.
Voici les fonctions assurées par l'amorce cliente (le stub):

- Initialiser un appel vers l'objet distant en appelant la couche des références distantes.
- Pliage des arguments de la méthode invoquée et transmission à la couche suivante. Pliage signifie : "*Fait de transformer les paramètres d'appel d'une méthode en un flux binaire transmissible via le réseau*". Ce mécanisme[1] est utilisé afin de permettre aux données d'être transmissible via le réseau.
- Signaler à la couche inférieure qu'un appel va avoir lieu.
- Dépliage de la valeur de retour s'il n'y a pas eu d'exceptions.
- Signaler à la couche inférieure que l'appel est terminé.

Un skeleton est une entité du côté serveur qui contient une méthode pour envoyer les appels vers l'implémentation réele d'un objet serveur
La couche amorce du côté serveur assure les services suivants :

- Dépliage des arguments reçus .
- Renvoie l'appel reçu vers l'implémentation de l'objet serveur.
- Pliage de la valeur de retour.

Les stub et les skeleton sont générés par un compilateur appelé *rmic* à partir du code des classes des objets distants. Il ne faut pas, ici, passer par un langage du type IDL.

Nous examinerons par la suite la possibilité de chargement dynamique de ces amorces.

2.2. La couche des références distantes (Remote Reference Layer)

Elle gère l'aspect fonctionnel de l'invocation.. C'est elle qui, par exemple, démarre un nouveau processus si l'objet serveur le nécessite, ou qui fait appel à un processus existant si l'objet est déjà actif dans ce processus. Elle reprend en quelque sorte le rôle de l'*Object Adapter* de Corba.

La couche des références d'objets communique avec la couche inférieure, à savoir la couche transport, grâce aux abstractions permises par cette couche. L'abstraction du canal de communication entre deux machines virtuelles Java est utilisée afin de permettre l'échange de données entre le serveur et le client. La couche des références distantes n'a simplement qu'à communiquer avec la couche transport via les différentes abstractions.

Cette couche offre les services suivants :

[1] Celui-ci se base sur le mécanisme de *sérialisation* que nous avons évoqué précédemment.

- Invocation point à point simple.
- Invocation vers des groupes d'objets répliqués.
- Support d'un mécanisme spécifique de réplication des objets.
- Support de l'activation d'objets persistants.
- Mécanisme de reconnexion automatique (si l'objet distant est inaccessible ou qu'il y a eu un problème sur la ligne) .

Cette couche, comme la précédente, dispose d'une partie cliente et d'une partie serveur distincte, qui assure notamment la gestion des objets répliqués.

2.3. La couche transport (Transport Layer)

En général, la couche transport de RMI est principalement responsable de :

- La connexion entre les deux espaces d'adressage (les deux machines virtuelles).
- La gestion des connexions en cours.
- L'écoute et la réponse aux appels d'invocations.
- La constitution d'une table de tous les objets distants disponibles dans un espace d'adressage.

Cette couche présente certains niveaux d'abstraction. Ces niveaux d'abstraction sont utilisés pour permettre à la couche supérieure, la couche des références distantes, d'être indépendante des protocoles de communication supportés par la couche transport. Ces niveaux d'abstraction permettent aux couches supérieures de ne pas tenir compte de certaines données physiques.

Ces abstractions sont les suivantes :

- Le flux est une abstraction qui permet d'utiliser différentes possibilités de transmission pour les appels.
- Les JVM (Java Virtual Machine) qui sont en communication.
- Le canal de communication établit entre deux JVM.
- Les connections comme abstraction du transfert de données.

3. Le ramasse-miettes d'objets distribués

Un des caractéristiques importantes du langage Java est sa capacité de gestion de la mémoire. En effet, un programmeur ne doit pas se préoccuper de la destruction des objets inutilisés en mémoire. Pour cela, Java dispose de ce qu'on appelle le *garbage collector*.

Dans sa version non-distribuée, le garbage collector fonctionne d'après le mécanisme suivant :

supposons la création d'un objet *Test*. Java associe à cet objet, un compteur. Ce compteur est incrémenté lors de la référenciation d'un autre objet à l'objet *Test*. Lorsqu'un objet qui référençait l'objet *Test* disparaît, le compteur est décrémenté. Lorsqu'il atteint la valeur nulle, l'objet *Test* est candidat à la disparition.

Le *garbage collector* est un processus qui se lance à certains moments de la vie d'un programme Java. Ces moments sont par exemple lorsqu'il y a pénurie de ressource mémoire ou à cause d'un appel particulier de la part d'un programme.

Le processus *garbage collector* passe en revue tous les objets du système et lorsqu'il trouve un objet dont la valeur du compteur est nulle, il le détruit. Le programmeur ne doit, dès lors, plus se soucier de la gestion de la mémoire.
Il existe également un système de *garbage collector* dans le mécanisme des RMI, son principe est similaire à celui exposé plus haut.

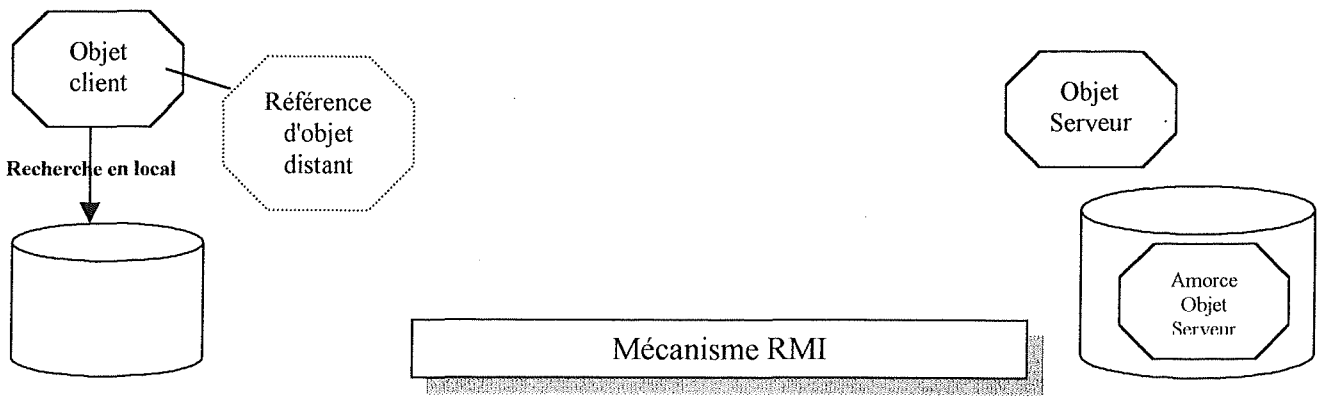
Ce *garbage collector* distribué implémente aussi en dehors de la gestion des objets référencés, un système de *time-out* qui permet de libérer des références du côté serveur détenues par des clients dont on n'a plus de nouvelles depuis un certain temps.

4. Le chargement dynamique d'amorces.

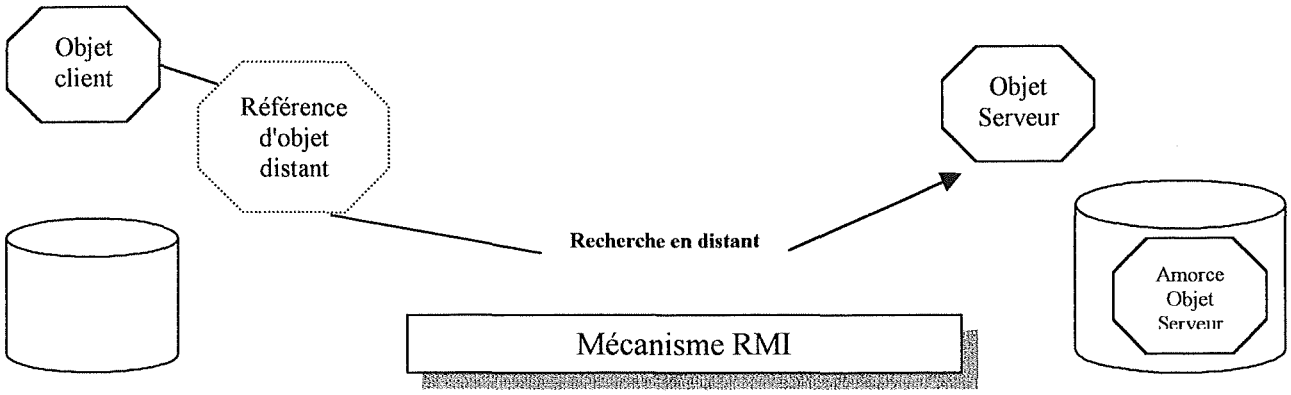
Une des caractéristiques intéressantes de RMI, est sa capacité à permettre l'utilisation d'objets distants dont les amorces ne sont pas disponibles au moment de la compilation.

Imaginons une applet s'adressant à des objets dont on ne dispose que de l'interface. Il est possible de développer uniquement sur base de cette interface des appels à des objets distants. Lors de l'exécution, quand un client obtient la référence d'un objet distant, RMI charge le stub qui transforme les appels sur cette référence en appels distant vers l'objet serveur. Ce chargement de stub est effectué par un mécanisme appelé *RMIClassLoader*. Voici le principe de fonctionnement de ce chargement dynamique:

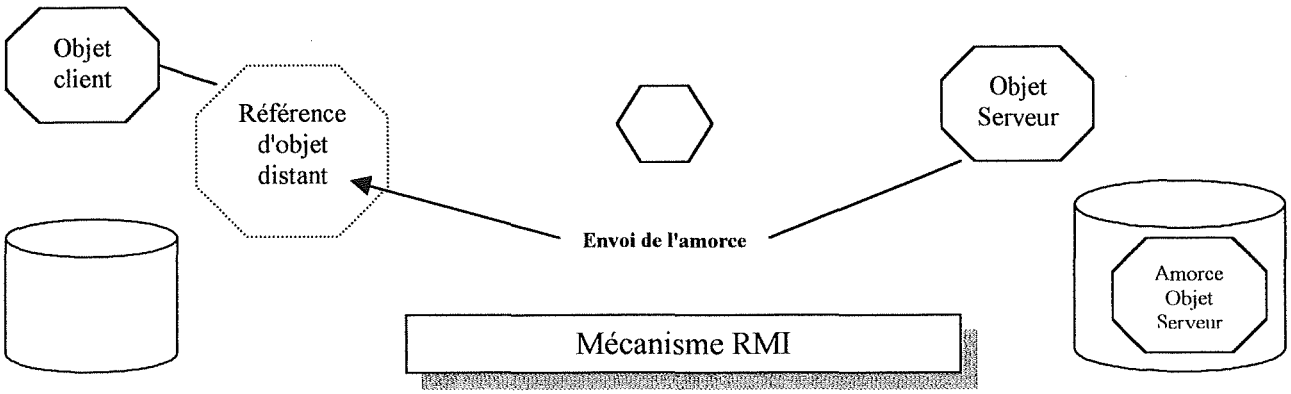
[1]



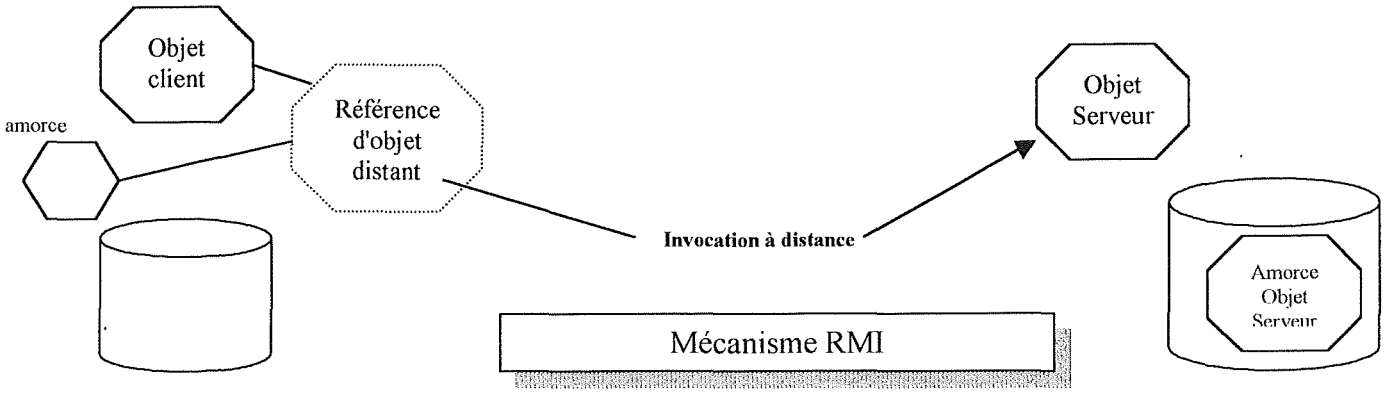
[2]


Objet client
Référence d'objet distant
Objet Serveur
Recherche en distant
Amorce Objet Serveur
Mécanisme RMI

[3]


Objet client
Référence d'objet distant
Objet Serveur
Envoi de l'amorce
Amorce Objet Serveur
Mécanisme RMI

[4]


Objet client
amorce
Référence d'objet distant
Objet Serveur
Invocation à distance
Amorce Objet Serveur
Mécanisme RMI

1. Le *RMIClassLoader* vérifie si le stub est en local.
2. Si il n'y est pas, c'est qu'il est situé sur le serveur distant.
3. L'amorce serveur est chargée sur le client.
4. Le client peut envoyer des requêtes à l'objet serveur grâce à l'amorce cliente.

On peut remarquer que ce chargement dynamique est assez proche du mécanisme des applets.

5. La sécurité dans les RMI.

Il nous a semblé intéressant d'aborder quelque peu la sécurité dans RMI et dans Java. Comme RMI est bâti entièrement sur Java, il respecte son modèle de sécurité.
Ce modèle de sécurité permet de ce prémunir contre quelques risques :

- Vérification du code et de son intégrité.
- Pas d'accès aux ressources locales pour les applets.
- Pas de connexion autres qu'au serveur hôte pour les applets.

En plus du *RMIClassLoader*, Il existe aussi une classe java appelée `java.rmi.RMISecurityManager`. Cette classe peut-être utilisée quand une application doit personnaliser les accès permis à des objets distants.

Cependant, certains aspects de sécurité n'ont pas encore été traités , en voici un exemple :

- Les classes chargées peuvent abuser du temps CPU en local. On pourrait supposer qu'une applet soit chargée et lance un mécanisme sans fin, ralentissant ainsi la machine. Ce problème n'est pas directement lié au mécanisme RMI en particulier, mais le fait de distribuer les objets, et donc de transporter des exécutables, augmente ce genre de risques

6. Conclusion

Java RMI est une spécification très générale et complètement adaptée au monde Java, dans lequel clients et serveurs sont écrits en Java. Elle définit en quelque sorte l'ensemble minimal de services nécessaires à la construction d'applications orientées objet distribuées entièrement en Java.

Les RMI font figure de poids plume par rapport à Corba et OLE. Mais, bien que moins complet que ceux-ci, il n'en est pas moins plus adapté au langage Java.

Ce mécanisme est également beaucoup plus simple d'utilisation et la position de Sun au sein d'Internet fait de RMI un outsider non négligeable.

Chapitre 5

ISM et les SML Object Wrappers

0. Introduction

ISM (Integrated System Management) est un outil d'administration de réseaux développé par la société Française Bull. Cet outil permet de gérer les différentes ressources d'un système distribué (machines, OS, périphériques, ...) et les réseaux les reliant.

Comme vous le savez peut-être, Bull est un membre actif de l'OMG et par conséquent, s'intéresse de très près à l'utilisation des technologies objet, ainsi qu'au développement d'Internet.
Dès lors, il n'est pas surprenant de voir l'ouverture d'ISM vers Internet et vers le langage Java comme un des objectifs de cette société.
La possibilité de pouvoir gérer son réseau à partir d'un browser Web permet à Bull d'embellir encore ISM, qui est déjà considéré comme un des meilleurs, si ce n'est le meilleur outil d'administration à l'heure actuelle.

Comment permettre cet accès via le Web? La solution toute simple saute aux yeux : Java et ses applets[1]. Les applets Java permettent de charger et d'exécuter des objets Java dans le navigateur Web d'une machine cliente.
Cependant, pour permettre une véritable interaction entre ces objets Java et les applications de gestion de la plate-forme ISM, Bull a développé un *middleware* objet :les SML Wrappers, du nom du langage servant de support à ISM, le SML[2].

Au cours de ce chapitre, nous vous présenterons brièvement ISM. Ensuite, nous analyserons en détail les SML Wrappers (buts, concepts, modèle de sécurité, ...).
Enfin, nous parlerons un peu de l'avenir d'ISM.

1. Présentation d'ISM

Notre but n'est pas ici d'examiner complètement ISM, nous pourrions d'ailleurs consacrer tout un ouvrage sur ce sujet. Notre objectif est simplement de présenter le cadre dans lequel les SML Wrappers furent développés.

[1] Voir chapitre 1, section 4.
[2] *System Management Language*.

1.1. Introduction

Avant toute chose, il faut savoir qu'ISM s'intègre dans un modèle appelé *Distributed Computing Model*[1]. Ce modèle d'architecture distribuée permet l'implémentation d'applications au sein d'un environnement distribué. Voici les différents composants de cette architecture :

- Les applications offertes à l'utilisateur.
- Les services d'applications (affichage, traitement de texte, ...).
- Les services de distribution (RPC, ...).
- Le développement d'applications (Atelier de génie logiciel,...).
- La gestion intégrée et la sécurité.
- Les services de communication.

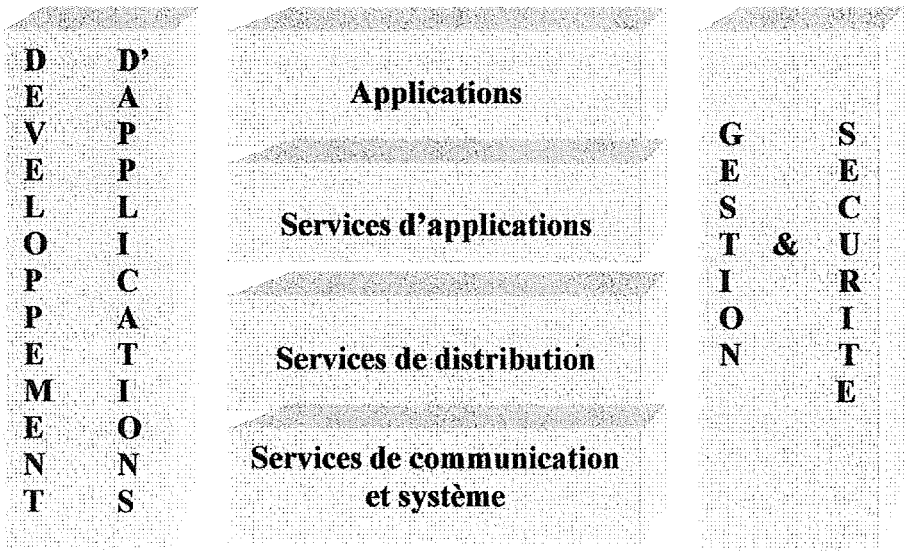


Figure 5.1. Distributed Computing Model.

ISM n'est finalement que la partie gestion du modèle DCM. Comme nous l'avons déjà signalé, le but d'ISM est de pouvoir gérer les différentes ressources d'un système distribué, ainsi que bien évidemment les réseaux reliants ces différentes ressources.

1.2. Architecture

L'architecture d'ISM est constituée de deux parties : le manager (*ISM Manager*) et les agents (*ISM Agent*). Les interactions entre ces deux parties se font via des protocoles de gestion tels que SNMP[1] ou CMIP[2]. Le manager tourne sur une machine appelée *machine administrateur*, tandis que les agents tournent sur les machines administrées.

[1] Nous utiliserons toujours l'abréviation DCM pour plus de facilité.
[1] *Simple Network Management Protocol.*
[2] *Common Management Information Protocol.*

La figure 5.2 nous présente cette architecture qui a la particularité d'être basée sur un modèle orienté objet.

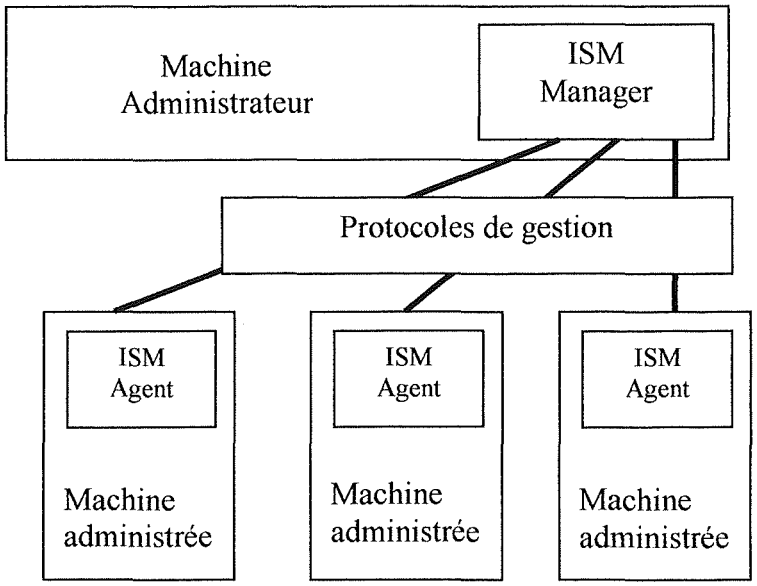


Figure 5.2. Architecture d'ISM.

Les agents ISM ont pour rôle de présenter au manager une vision du monde à travers des objets de gestion. Ces objets représentent les ressources du système distribué, son état, ses utilisateurs, ses alarmes, etc...
Ils sont organisés dans une MIB (*Management Information Base*). C'est au travers de cette MIB que l'*ISM Manager* « voit » le système distribué ; il ne s'adresse pas directement à une machine mais aux objets qui la représente .

1.3. ISM Manager

Le manager ISM possède différents composants : des services, des intégrateurs d'agents, des applications et une infrastructure de communication. La figure 5.3 nous offre une vision de ces différents composants.

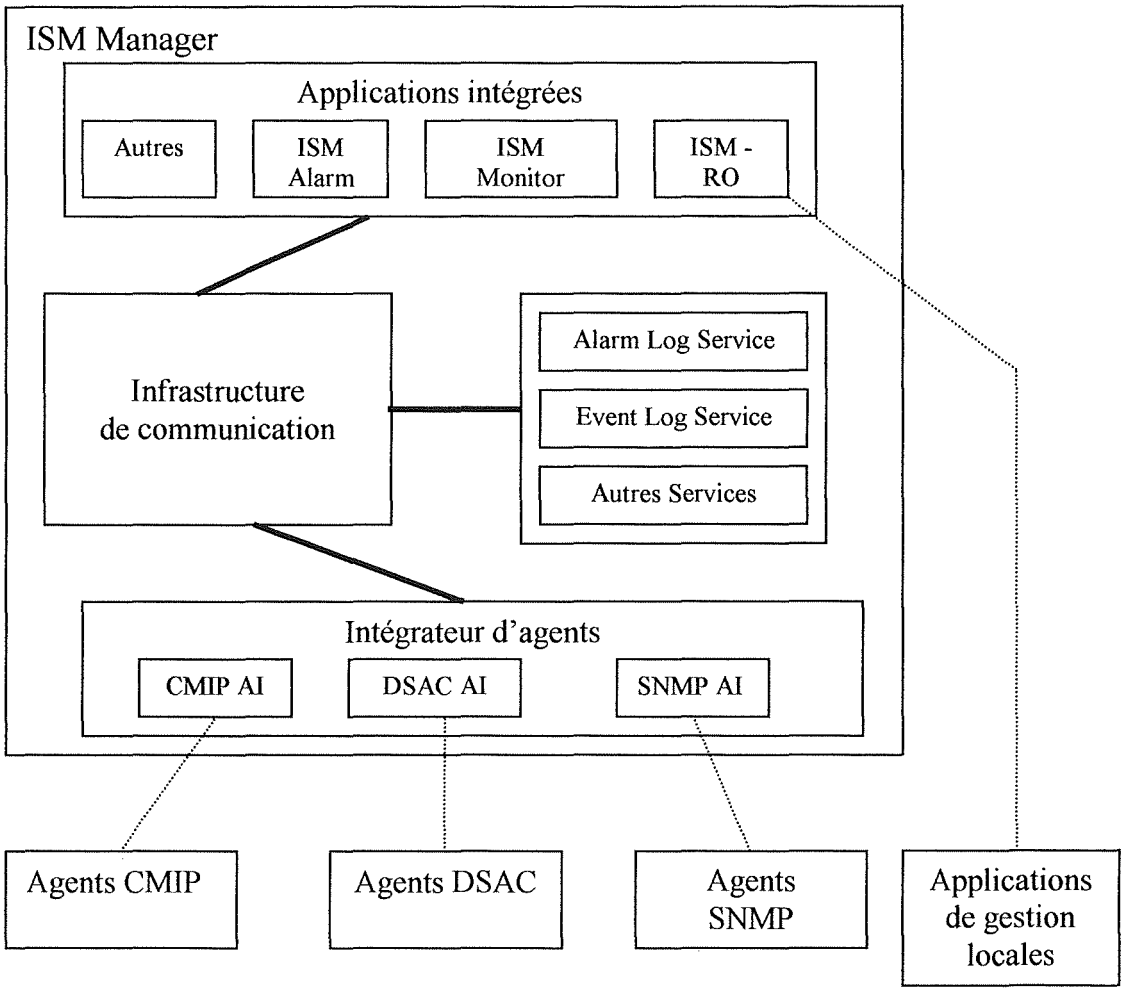


Figure 5.3. Architecture du manager ISM.

1.3.1. Les applications

Les utilisateurs interagissent directement avec ISM qu'à travers les applications de gestion. Celles-ci offrent une multitude de fonctionnalités allant de la gestion des erreurs à la gestion des performances ou à l'analyse de réseau.
Nous allons passer en revue quelques-unes d'entre elles :

- **ISM Monitor :** c'est l'application centrale servant de base à ISM. Elle permet d'accéder aux autres applications et offre notamment la possibilité de représenter un objet tel qu'un réseau par un symbole graphique. L'utilisateur peut, dès lors, sélectionner cet objet et lancer une application se rapportant à celui-ci.

- **ISM Alarm :** il s'agit d'une application de gestion des fautes qui permet la détection et l'affichage d'alarmes.

- **ISM Discovery :** cette application est utilisée pour découvrir les machines présentes sur un réseau particulier.

- **ISM Remote Operation:** elle permet à l'administrateur de se connecter à distance sur une machine depuis ISM Monitor.

- **Etc, ...**

1.3.2. L'Infrastructure de communication

Tous les éléments du manager ISM communiquent à travers cette infrastructure. Elle fournit les fonctions de transfert de messages du manager ISM et supporte la répartition des différents composants à travers un système distribué.
Cette infrastructure offre de nombreux services dont notamment :

- **L'Internal Protocol Stack** : il s'agit d'un protocole interne permettant aux différents composants du manager ISM de communiquer ; ce protocole se base sur CMIP.

- **Une interface de programmation** : elle est accessible via le langage de développement SML. Elle fournit, notamment les services CMIS[1] aux composants du manager ISM.

- **Un routeur d'événement :** ce service est utilisé pour router les événements émanant des composants de l'*ISM Manager*. De plus, il permet de filtrer ces événements, sur demande d'une application ou d'un service.

1.3.3. Les services

ISM fournit un ensemble de services communs aux applications. La plupart d'entre eux offrent généralement l'accès à des bases de données contenant, par exemple, les objets gérés ou leurs classes. Il existe également un service permettant la communication entre ISM Manager et d'autres systèmes d'administration.

1.3.4. Les intégrateurs d'agents[2]

Ces éléments permettent la communication entre l'ISM Manager et les agents des machines administrées. Ils constituent une des particularités d'ISM.

Comme nous l'avons signalé précédemment, la plate-forme ISM supporte plusieurs protocoles de gestion, à savoir :CMIP, SNMP et DSAC. Chaque protocole a ses particularités.

---

[1] ISM utilise en interne le protocole CMIP, CMIS représente l'ensemble des services de gestion mis en œuvre par ce protocole.
[2] Nous utiliserons l'abréviation AI pour plus de facilité.

Il est bien évident qu'une requête envoyée par un agent CMIP n'a pas la même structure qu'une requête envoyée par un agent SNMP.
Le protocole interne d'ISM étant basé sur CMIP, il faut ajouter des composants capables de comprendre les requêtes ISM et de les reformuler en fonction des agents qui leur sont assignés. C'est le rôle des intégrateurs d'agents.

Si un utilisateur via une application souhaite interroger un agent particulier, il enverra une requête respectant le protocole interne d'ISM. Suivant que la requête s'adresse à un agent CMIP, SNMP ou DSAC, elle sera routée vers l'AI correspondant, qui sera en mesure de la transcrire en une requête CMIP, SNMP ou DSAC.

L'AI est chargé de recevoir les réponses d'un agent et de les retranscrire en requêtes ISM vers le client. Il existe donc un agent intégrateur pour chaque protocole de gestion admis par ISM, à savoir SNMP, CMIP et DSAC.

1.4. Les agents

Sur chaque machine du système distribué tourne au moins un agent responsable du contrôle de cette machine. Tout agent doit effectuer les opérations de gestion demandées par l'ISM Manager (qu'il s'agisse de consultation ou d'actions). Chaque agent travaille selon un protocole de gestion propre et dialogue avec l'ISM Manager au travers d'un seul AI.

1.5. Le langage SML

Ce langage est un langage interprété, fonctionnel assez proche du Lisp. Il fournit les moyens aisés de développer des applications d'administration dans l'environnement d'ISM et permet d'accéder, de manipuler et de présenter l'information de gestion.
SML intègre également un modèle objet.

2. Principes et concepts des SML Wrappers

2.1. Vue d'ensemble

Comme nous l'avions brièvement annoncé dans l'introduction, les SML Wrappers ont été développé pour permettre l'accès à une plate-forme ISM à partir d'un browser Web. Autrement dit, le but principal des Wrappers est de permettre le développement d'applications de gestion sur le Web en utilisant les applications et les services déjà existants sur la plate-forme ISM.

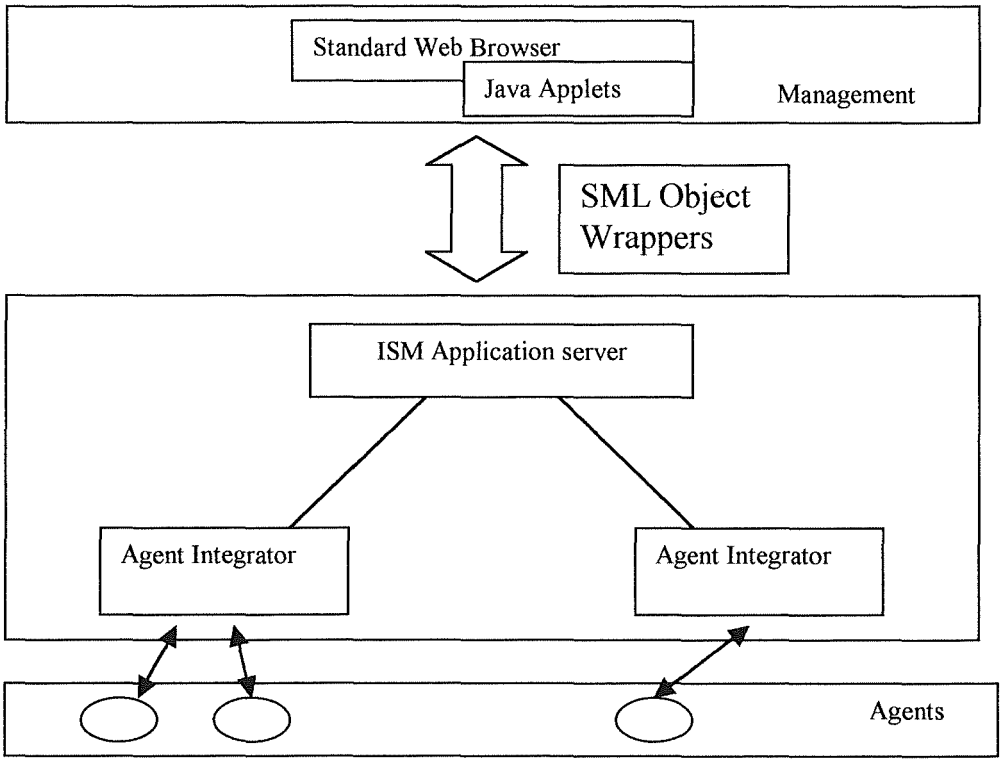


Figure 5.4. Objectif des SML Wrappers

Dans le cadre de mon stage chez Bull, j'ai participé au développement de l'applet chargée d'utiliser les services d'une application appelée *ISM Query Builder*. A titre indicatif, cette application permet de construire des requêtes afin d'interroger une base de données contenant des informations de gestion.

Java n'offrant que des moyens de communication très limités à partir des applets, il était indispensable de développer un *middleware* pour permettre aux objets java de l'applet d'interagir avec des objets SML ou des fonctions SML présentes du côté du serveur d'applications d'ISM.
Mais, pourquoi Bull a-t-il développé son propre middleware alors qu'il existe des standards d'objets distribués tels que Corba, OLE ou encore RMI ?

Disons que ces systèmes ne correspondaient pas exactement aux besoins de Bull. Ce qu'il fallait, c'était un *middleware* léger, simple, sûr, et utilisant pleinement les caractéristiques d'ISM et de son langage SML.
Bull aurait pu utiliser les RMI, mais ceux-ci ne sont pas supportés par *Internet Explorer*.
Il y avait également la solution de DCOM ou Corba. Mais, ces *middlewares* sont trop lourds à mettre en place alors qu'on sait très bien que le langage du côté serveur sera toujours SML.

Ainsi, Bull développa son propre *middleware* : les SML Wrappers. Bien qu'ils aient été développés dans l'optique de l'extension d'ISM sur le Web, ils permettent également la construction d'applications clientes en C et en SML.

2.2. Concepts de base

Grâce aux SML Wrappers, un client peut soit invoquer une méthode d'un objet SML ou soit une fonction SML présente sur un serveur d'applications ISM distant.

Nous allons maintenant envisager ces deux cas de figure en examinant les mécanismes mis en œuvre par les Wrappers.

2.2.1. L'invocation de fonctions

2.2.1.1. Du côté client

Pour qu'un client puisse effectuer l'invocation d'une fonction offerte par un serveur ISM, il doit disposer de ce qu'on appelle un *proxy* client.
Cet objet est chargé de recevoir la requête du client et de la transmettre au serveur auquel le client souhaite se connecter.

C'est au programmeur de l'objet client de créer ce *proxy*. Pour cela, les SML Wrappers mettent à sa disposition la classe *proxyClient* et ce, dans le langage qu'il utilise (Il s'agit généralement de Java).
Le programmeur n'a, dès lors, plus qu'à créer une instance de la classe *proxyClient* en donnant des paramètres tels que le nom de la machine hébergeant le serveur ISM souhaité.

Une fois, ce *proxy* client créé, le programmeur n'a plus qu'à l'utiliser pour envoyer au serveur une requête pour invoquer une fonction. Il doit bien évidemment connaître le nom de la fonction et les paramètres lors de la phase de développement de son application.

2.2.1.2. Du côté serveur

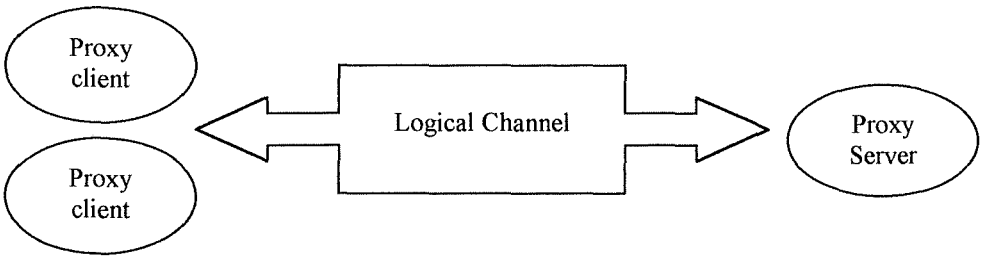
Pour pouvoir recevoir les requêtes du côté serveur, il faut que celui-ci dispose d'un *proxy* serveur. Cet objet est chargé de:

- Gérer les communications avec les proxy clients.
- Gérer les implémentations d'objets.

Un *proxy* serveur ne rend accessible qu'un certain nombre de fonctions que le programmeur du serveur aura spécifié. Si un client veut exécuter une fonction que le *proxy* serveur n'exporte pas, une exception sera générée.

2.2.1.3 Le canal logique

Les *proxy* clients et les p*roxy* serveurs communiquent au travers d'un canal logique identifié par un nom unique pour une machine hôte donnée.
A un *proxy* serveur d'une machine particulière correspond un canal logique par lequel un ou plusieurs *proxy* clients communiquent.



5.5. Architecture client-serveur.

Un *proxy* serveur « écoute » un canal logique, prêt à recevoir les requêtes envoyées par un *proxy* client.
Lors de la création du *proxy* serveur, le nom du canal logique par lequel le *proxy* serveur est accessible doit être spécifié.
Le programmeur du client doit quant à lui, lors de la création du *proxy* client, spécifier le nom du canal logique que le *proxy* client utilisera.

L'implémentation de ce canal logique est totalement transparente pour le client. Sachez juste qu'elle repose actuellement sur un protocole propriétaire de Bull et qu'une implémentation utilisant le protocole IIOP est en cours d'achèvement.

2.2.2. L'invocation de méthodes

Avant de pouvoir invoquer les méthodes offertes par un objet serveur, un client doit d'abord créer cet objet.
Pour créer un objet SML distant, le programmeur client doit tout comme pour l'invocation des fonctions disposer d'un *proxy* client. Ensuite, il doit utiliser la méthode *makeInstance* offerte par ce *proxy* client en fournissant comme paramètre le nom de la classe qu'il souhaite instancier. Cette méthode renvoie un objet de la classe *SOWShadow*.

Le client, pour invoquer une méthode de l'objet distant, utilise l'objet Shadow correspondant en spécifiant le nom de la méthode qu'il souhaite exécuter et en fournissant les paramètres.

Notez bien que l'objet Shadow ne représente pas l'interface de l'objet distant. En fait, il possède des méthodes (ex :la méthode *delegateSync()*) que le client invoque en donnant comme paramètres le nom de la méthode de l'objet SML distant qu'il souhaite exécuter et, la liste des paramètres exigés par cette méthode.

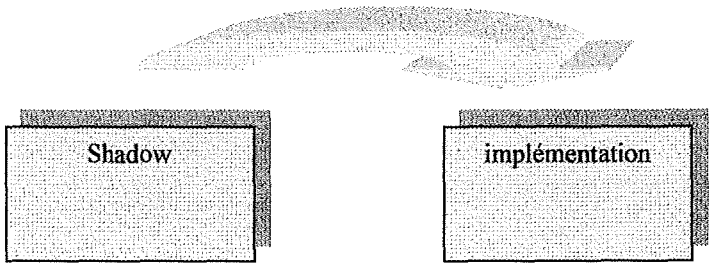


Figure 5.6 . Shadow et Implémentation

Concrètement, lorsqu'un client utilise la méthode *makeInstance()* d'un *proxy* client, un message est envoyé au p*roxy* serveur.
Ceci implique que tout comme pour l'invocation de fonctions, le *proxy* serveur doit exister du côté serveur et il doit exporter les classes disponibles pour les clients.

Suite à la réception de la requête de création par le *proxy* serveur, l'instance de la classe donnée est créée et sa référence est renvoyée au *proxy* client. Celui-ci crée un objet *Shadow* qui va stocker la référence de l'objet distant.

Ainsi, lorsque le *Shadow* reçoit une requête du client, il la transmet au *proxy* serveur qui, à partir de la référence et du nom de la méthode, s'arrange pour exécuter la bonne méthode du bon objet.

2.2.3. Requête synchrone versus asynchrone

Les SML Wrappers permettent l'envoi aussi bien de requêtes synchrones que de requêtes asynchrones.

Dans le cas, le client envoie la requête et attend jusqu'à ce que le résultat soit retourné.

Dans le mode asynchrone, le client envoie la requête et continue son exécution sans attendre la réponse. Une fois que la requête du côté serveur est exécutée et le résultat renvoyé, une méthode particulière d'un objet de type c*allback* est invoquée du côté client.

Ici, une *callback* est une interface qui décrit un certain nombre de méthodes spécifiées par les Wrappers. Ces méthodes correspondent aux différents cas de figure qui pourraient avoir lieu lors de l'exécution de la requête asynchrone.
Une fois que la réponse est reçue du côté client, la bonne méthode d'un objet implémentant l'interface *callback* est appelée.

Il est bien évident que dans ce cas-ci, le programmeur client doit non seulement fournir le nom d'une méthode ou d'une fonction à exécuter avec les paramètres, mais également le nom de l'objet qui sera exécuté lors de la réception de la réponse.

2.3. Autres Concepts

2.3.1. Les événements

Les Wrappers permettent à un serveur d'envoyer des événements aux clients prêts à recevoir ces événements. Un événement est une instance de la classe SOW :Event :

- Un événement est envoyé à un *proxy* client connecté à un *proxy* serveur, en utilisant l'appel SOW:throw sur ce *proxy*.
- Un événement est envoyé au *proxy* client où l'instance *Shadow* est situé, en utilisant la méthode SML SOW :throw sur une instance implémentée.

2.3.2. Le Garbage Collector

Quand une instance n'est plus référencée, cette instance est automatiquement détruite et la mémoire est libérée.

2.4. Le modèle de sécurité

Nous n'avions pas encore abordé le problème de la sécurité mais, il ne fait pas l'ombre d'un doute qu'il s'agit du problème le plus important à prendre en considération dans le cas des SML Wrappers. Effectivement, si aucune mesure de sécurité n'est prise, on pourrait voir des *personna non-grata* accéder aux services d'ISM. Elles auraient ainsi la possibilité de connaître la configuration des réseaux, lancer des requêtes pouvant ralentir le système, ...

Les services de sécurité fournit par les SML Wrappers sont les suivants :

1. Authentification mutuelle pour les clients et serveurs des SML Wrappers. Après cette phase d'authentification mutuelle, l'utilisateur final a des garanties sur l'identité du serveur, et l'application serveur a des garanties sur l'identité de l'utilisateur.

2. L'intégrité de toutes les données échangées entre clients et serveurs. Les messages sont protégés contre d'éventuelles modifications par l'utilisation de sceaux cryptographiques.

3. Confidentialité des données sensibles . Certaines parties de messages sensibles peuvent être cryptées.

Une fois qu'une session est établie, les messages sont délivrés à travers un canal de communication sécurisé. Un contexte de sécurité est associé à chaque requête et l'application serveur peut vérifier si le canal est vraiment sécurisé ou si le client est bien celui qu'il prétend être.

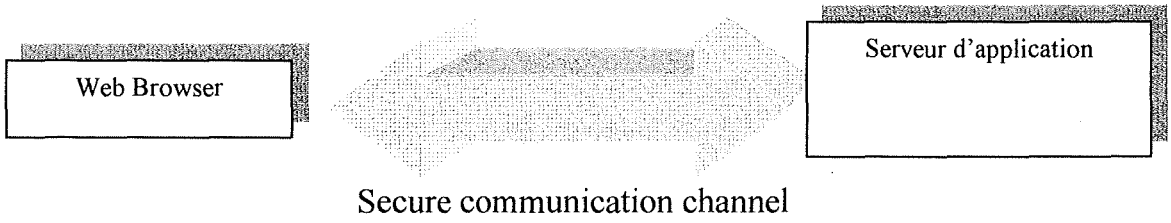


Figure 5.7 : canal de sécurité.

3. Et l'avenir ?

Cette année est sans aucun doute pour ISM l'année Java. Au moment de la rédaction de ce mémoire, la plupart des applications et des services d'ISM était presque disponible en Java.
Est-ce que Bull en restera là ? Certains indices nous font penser le contraire. L'implémentation des canaux logiques est déjà sur le point de passer sur IIOP, ce qui constitue une passerelle vers Corba. Mais, de là à ce qu'on aie un système d'administration basé sur Corba, il n'y a qu'un pas que nous ne franchirons certainement pas ici.

4. Conclusion

Nous venons de voir dans ce chapitre un exemple d'application pratique des objets distribués. Beaucoup d'entreprises, dans de nombreux domaines commencent à les utiliser et il y a fort à parier que presque personne ne pourra y échapper.

Le cas des SML Wrappers fait de nouveau ressortir l'importance d'Internet dans le développement des objets distribués. Dans le premier chapitre, nous avions dit que l'on trouverait de tels objets partout où il y aurait un accès à Internet. Ce cas-ci nous le prouve bien.

CONCLUSION

Nous terminons ici ce mémoire sur cette technologie fascinante et en pleine expansion que constituent les systèmes d'objets distribués. Nous espérons avoir répondu à la plupart des questions que vous vous posiez à ce sujet.

Nous avons dans le premier chapitre, introduit les concepts nécessaires à la compréhension des systèmes d'objets distribués. Nous nous sommes également intéressés aux raisons du développement phénoménal de ces systèmes, en attirant tout particulièrement l'attention sur le rôle d'Internet.

Nous avons ensuite analysé brièvement au cours du deuxième et du troisième chapitre, OLE et Corba. Même si nous n'avons pas effectué une réelle comparaison entre ces deux technologies, vous avez pu constater par vous-même les différences techniques importantes entre les deux, tout en notant bien la similitude des services offerts.

Dans le quatrième chapitre, nous avons introduit l'outsider de luxe d'OLE et de Corba, à savoir RMI. Ce chapitre, bien qu'assez court, nous montre bien les différences de RMI par rapport aux autres systèmes.

Nous avons clôturé ce mémoire par la présentation d'ISM et des SML Wrappers. Ce cas nous a permis de saisir la réelle importance des objets distribués, mais également celle d'Internet qui constitue véritablement le tremplin de cette technologie objet.

BIBLIOGRAPHIE

[BULL,97] BULL, « *SML Object Wrappers* », Bull, 1997.

[BULL,97] BULL, « *SML Object Wrappers, the Tutorial* », Bull, 1997.

[CHAPPELL,96] CHAPPELL Davis, « *Au cœur de ActiveX et OLE* », Microsoft Press France, 1996.

[CHAUVET, 97] CHAUVET Jean-Marie, « *corba, ActiveX et Java Beans* », Editions Eyrolles, 1997.

[DCOM, 97] MICROSOFT « *DCOM Technical Overview* », Microsoft, 1996.

[FLANAGAN, 96] FLANAGAN David, «*Java in a nutshell* », Edition O'Reilly International Thomson, 1996.

[GARDARIN, 96] GARDARIN Georges, GARDARIN Olivier, «*Le client-serveur* », Edition Eyrolles, 1996.

[JERPH, 98] BULL, «*OpenMaster Java-based interface*», Bull, 1998.

[MARTIN, 97] MARTIN Michel, «*Java 1.1, le platinium* », Sybex, 1997.

[MOWBRAY,95] MOWBRAY Thomas, ZAHAVI Ron, « *The Essential CORBA* », John Wiley & Sons, 1995.

[NICOLAS, 98] NICOLAS Cédric, AVARE Christophe, NAJMAN Frédéric, «*Java Client-Serveur*», Editions Eyrolles, 1998.

[OMG, 91] Object Management Group, «*The Common Object Request Broker Architecture and Specification*», OMG, 1991.

[OMG, 92] Object Management Group, «*Object Management Architecture Guide*», OMG, 1993.

[OMG, 97] CURTIS David, *«Java, RMI and Corba»*,OMG, 1997.

[ORFALI,96] ORFALI Robert, HARKEY Dan, EDWARDS Jeri *«The essential distributed objects survival guide»*, Wiley computer publishing, 1996.

[ORFALI,96] ORFALI Robert, HARKEY Dan, EDWARDS Jeri, *« Client/Serveur, guide de survie »*, International Thomson Publishing France, 1997.

[RMI, 96] SUN, *«Remote Method Invocation Specification»*,SUN, 1997.

[SIEGEL,96] SIEGEL John, *« CORBA Fundamentals and programming. »*, John Wiley & Sons, 1996.

[TAYLOR,90] TAYLOR David, *« Object-Oriented Technology : A Manger's Guide »*, John Wiley & Sons, 1990.

[VOGEL, 97] VOGEL Andreas, KEITH Duddy, *«Java programing with CORBA»*, Wiley computer publishing, 1997.

[VOSS, 91] VOSS Greg, *« Object-Oriented programming : An introduction »*, Osborne McGraw-Hill, 1991.

ADRESSES INTERNET

OMG – Object Management Group (Corba)
http://www.omg.org

Les spécifications de Corba 2.0 peuvent être consultées à l'adresse suivante :
http://www.omg.org/corba2/corb2prf.htm

W3C – World Wide Web Consortium
http://www.w3.org

OPEN GROUP (DCE)
http://www.opengroup.org/tech/dce/

MICROSOFT (ActiveX, COM/Dcom...)
http://www.microsoft.com

Les informations relatives aux technologies ActiveX sont regroupées sur le site suivant :
http://www.microsoft.com/ActiveX

SUN (RMI)
http://java.sun.com

Vous trouverez les spécifications des RMI à l'adresse suivante :
http://java.sun.com/products/jdk/1.1/docs/guide/rmi/spec/rmiTOC.doc.html

JAVASOFT

http://www.javasoft.com