Introduction aux systèmes d'objets distribués

ISIMA 3–1997/1998

TABLE DES MATIERES

1. Les objets distribués

1.1 Les exigences du monde moderne

1.1.1 Introduction

L'évolution des technologies informatiques au cours des dernières années a entraîné des modifications radicales dans la conception des applications.

L'essor d'un parc considérable de micro-ordinateurs performants du à la baisse des tarifs et à l'émergence de nouveaux processeurs très puissants.

Les nouveaux réseaux de télécommunication rapides et à large bande passante (RNIS, ATM, High Performance Ethernet) ont permis le regroupement des ordinateurs en réseaux, d'abord de taille réduite (entreprise), puis moyenne (nationale) puis finalement mondiale avec Internet .

Tout ceci a favorisé la démocratisation des communications informatiques (à commencer par les messageries électroniques et la navigation sur le Web) puis, plus récemment, l'émergence des applications distribuées.

1.1.2 Avantages des systèmes distribués

Les structures de dimension internationales telles que la défense, les banques ou les grandes entreprises voient leurs activités s'étendre planétairement et ne peuvent plus centraliser toute l'information sur un système unique. En outre, de plus en plus de transactions courantes s'effectuent via les réseaux.

Prenons l'exemple d'une société de courtage gérant des affaires dans de très nombreux pays. Plutôt que de centraliser toute l'information sur un site central, il vaudra mieux distribuer l'information locale à chaque pays dans son comptoir associé. Ce système présente les avantages suivants :

Performance : lors d'une transaction entre deux pays, les données utilisées seront principalement celles situées dans les comptoirs des pays concernés. Ainsi, on limite d'autant la congestion d'un système central. En outre, cette dispersion en unités indépendantes et autogérées permet de mettre à jour plus simplement les données ayant attrait, par exemple à la législation locale d'un pays.

Robustesse : lors d'une défaillance du système informatique d'un comptoir, seul celui-ci est touché. Le système continue de fonctionner dans sa globalité

Evolutivité : une transaction peut avoir besoin de données situées sur d'autres serveurs que les deux qui l'ont initiée et s'étendre géographiquement. Comme précisé plus haut, les mises à jour des données de base sont moins coûteuses et pénalisantes pour les performances du système.

1.1.3 Définitions

Avant d'entrer dans le vif du sujet, rappelons quelques définitions utilisées dans la suite de cet exposé.

Transaction : une transaction est une action considérée comme atomique, c'est à dire indivisible (en particulier relativement aux crashs du système), accédant à des données partagées et gérant la concurrence.

Middleware : c'est la couche logicielle qui donne l'impression à l'utilisateur de travailler sur un système monolithique alors qu'il accède à plusieurs serveurs.

1.1.4 Récapitulation des besoins :

Chaque jour, notre vie courante (et plus particulièrement, la vie économique) devient plus orientée vers les réseaux. De par la mondialisation des échanges, les réseaux classiques à l'intérieur d'une même structure ont vécu. Désormais, lors d'un échange commercial, deux programmes intelligents vont demander à se connecter pour réaliser la transaction.

La notion classique de serveur et de client s'estompe. En effet, au cours d'une transaction distribuée, le serveur d'un client A peut très bien devenir à son tour le client d'un serveur B et ainsi de suite. Les rôles de client et serveur peuvent même s'inverser au cours du temps.

Tout ceci engendre un volume de transactions et de traitement sans cesse croissant, et, manifestement, les structures informatiques classiques explosent sous la charge. Les systèmes d'objets distribués devraient permettre de palier à ces difficultés.

Afin de réaliser des applications distribuées plus efficaces, plus sûres et plus simples à mettre en œuvre, on devra disposer des technologies suivantes :

1.1.4.1 Traitement transactionnel puissant

Transactions multi-serveurs : pour certaines applications, il est nécessaire d'accéder à des données situées sur plusieurs serveurs.

Transactions imbriquées : un traitement peut très bien avoir besoin de sous-traitements eux-mêmes distribués.

Transactions longues durées : c'est un problème inhérent aux transactions multi-serveur ou imbriquées qui nécessitent un grand nombre de données situées sur des serveurs différents. Cela peut résulter en des délais de réponse non négligeables, surtout en cas d'encombrement du réseau.

Transactions sécurisées : la circulation d'informations **confidentielles** sur le réseau n'est pas sans poser des problèmes de sécurité (cryptage, etc ...).

Gestion des transactions par files d'attentes : nécessaires à la gestion d'un grand nombre de transactions et au contrôle du flux de données

Super-serveurs de transactions : pour permettre de répartir la charge de travail en cas d'arrivée massive de transactions.

1.1.4.2 Agents itinérants

Les agents itinérants sont des entités capables de voyager sur le réseau de leur propre chef. Ils se propagent en utilisant des scripts de description qui sont interprétés par des moteurs de création.

Les agents itinérants seront majoritairement chargés du recueil de connaissance sur le réseau. Que ce soit à des fins commerciales (statistiques sur les marchés ...) ou d'administration du réseau (mise à jour d'annuaires) ou simplement des outils de recherche.

Ceci suppose que les agents devront pouvoir se créer dynamiquement sur n'importe quelle nouvelle plate-forme à partir de scripts génériques. Ce qui implique l'existence de moteurs de création et d'un environnement minimal assurant la survie des agents sur chaque noeud du réseau.

1.1.4.3 Gestion de données évoluée

Gestion de documents multi-média composites modulables, visualisables, modifiables, etc ... en tout point du réseau. Ce qui suppose que les hôtes du réseau soient capables d'en assumer la complexité. Des couches logicielles telles que OLE ou OpenDoc (CORBA) leur sont dédiées.

Super-serveurs de documents permettant d'archiver et de redistribuer d'énormes quantités de documents

Compatibilité avec les bases de données existantes, notamment SQL

1.1.4.4 Nouveaux systèmes d'exploitation

Les nouveaux systèmes d'exploitation auront à subir de nouvelles contraintes du fait de la répartition. En particulier, ils devront pouvoir s'adapter à des formes de communication différentes, de se reconfigurer pour répondre à de nouveaux besoins ou de se protéger contre des intrusions. En outre, la robustesse au crash va devenir de plus en plus prépondérante. Actuellement, les fournisseurs proposent majoritairement des systèmes multi-thread orientés réseau (NT plug&play ou OS/2 Warp–Connect par exemple)

1.1.4.5 Une couche middleware intelligente

En effet, elle devra être capable de localiser des objets dispersés tout au long du réseau et d'établir des appels de procédures de manière totalement transparente à l'utilisateur et au programmeur.

1.2 L'évolution des systèmes distribués au cours du temps

1.2.1 Mainframe

1.2.1.1 Description :

La période des mainframe marque l'apogée des systèmes centralisés. Un seul et même ordinateur (le système central, ou *mainframe*) concentre toute l'intelligence du programme. Il gère également le réseau (le plus souvent en étoile) où sont connectés des terminaux passifs de requête. A noter que ces derniers n'ayant aucune intelligence, le système central se charge également de toute la partie interface utilisateur.

1.2.1.2 Problèmes :

Gérer la centralisation d'un tel système avec les problèmes généraux qui y sont liés.

Non robustesse en cas de panne

Problèmes de congestion en entrée du serveur lors de l'arrivée massive de requêtes (congestion de communication)

Congestion de traitement car le système central est chargé non seulement de l'administration des données mais aussi de tout le reste de l'application, entre autres, de la partie interface utilisateur.

1.2.2 La technologie Client-Serveur et les serveurs SQL

1.2.2.1 Description

L'application devient enfin répartie. Plutôt que de concentrer tout le processus le serveur ne se charge plus (si l'on peut dire !) que de l'administration des données. Tout le reste de l'application se déroule sur les postes clients.

Les architectures à base de serveur de requêtes SQL représentent l'archétype de cette technologie et restent de loin le modèle distribué le plus répandu.

1.2.2.2 Problèmes

Ne convient pas au traitement de données complexes, et notamment de données situées sur des serveurs différents.

La gestion des transactions est trop limitée (pas de transactions imbriquées, serveur unique, etc ...).

Middleware non standard : chaque fournisseur propose sa propre couche de communications et il n'existe pas de protocole standard d'acheminement des requêtes et des réponses.

Lenteur de fonctionnement : la gestion des requêtes et commandes SQL est particulièrement lente. Afin de palier cet inconvénient majeur, différents fournisseurs ont proposé d'utiliser des *procédures stockées*, c'est-à-dire des ensembles prédéterminés de commandes SQL compilées et stockées dans le format natif du serveur. Celles-ci sont ensuite appelées par les *clients* à travers un mécanisme de type RPC. Ceci présuppose que le client connaisse l'existence de ces procédures stockées et interdit toute utilisation en milieu hétérogène. En outre, le standard SQL est très en retard sur les solutions fournisseurs dans ce domaine.

1.2.3 Les moniteurs transactionnels

1.2.3.1 Introduction

Les moniteurs transactionnels sont apparus afin de régler les problèmes de charge de communication et de traitement sur les gros systèmes. Ils sont donc chargés d'orchestrer le fonctionnement des grands logiciels en découpant ceux-ci en sous-ensembles qui seront considérés comme atomiques : les *transactions*. La gestion des ressources ne se fera donc plus au niveau du logiciel mais au niveau des transactions.

1.2.3.2 Fonctionnement

Au moyen des transactions, un moniteur transactionnel est capable de faire cohabiter (et même coopérer) différents logiciels (ou parties de logiciels) n'ayant aucune connaissance mutuelle *a priori* et conçus pour pouvoir traiter un très grand nombre de clients.

Un moniteur transactionnel peut gérer des ressources sur différents serveurs (même hétérogènes) et peut coopérer avec d'autres moniteurs transactionnels pour, par exemple, équilibrer la charge de traitement sur plusieurs serveurs.

Vu de l'extérieur, il apparaît comme une couche supplémentaire qui accepte toutes les demandes de ressources en vrac pour les redistribuer, via une file d'attente, sur un rythme compatible avec les performances du serveur. Par exemple, il sera capable de transformer des arrivées massives par bouffées en un flot continu de demandes.

Les moniteurs transactionnels sont actuellement le modèle le plus fiable et le plus robuste pour la gestion des données réparties.

1.2.3.3 L'évolution

L'utilisation de moniteurs transactionnels peut faciliter le passage d'une architecture client-serveur classique vers un modèle d'objets distribués. En effet :

1) A l'instar des modèles d'objets distribués, les moniteurs transactionnels récents prônent le découpage d'une application en trois couches :

   ⇒ traitements

   ⇒ données

   ⇒ interface utilisateur

2) Les fournisseurs de moniteurs transactionnels ont participé de façon active à la rédaction du modèle CORBA et de ce fait, les moniteurs transactionnels les plus récents ressemblent à des ORB, les courtiers en objets distribués qui sont à la base de cette technologie.

Aussi, on peut espérer que ces deux technologies coopéreront de manière efficace, les moniteurs transactionnels gérant des composants intelligents et des transactions évoluées faisant intervenir les objets distribués.

1.2.4 Le groupware

1.2.4.1 Définition

Le *groupware* (ou *collecticiel* dans notre belle langue) propose de modéliser et traiter informatiquement les processus et activités complexes inhérents au travail en équipe. A cette fin, il repose essentiellement sur 5 technologies de base :

Les documents multimédia pour une représentation cohérente de l'information

Les flux d'activités utilisés pour acheminer automatiquement des événements et des tâches d'un programme au suivant dans les environnements clients serveurs

Le courrier électronique qui, de par sa nature profondément asynchrone (car basé sur une technologie transmettre–stocker–consulter) est bien adapté au fonctionnement des entreprises

La gestion des sessions de travail (ou gestion d'agenda) qui concerne aussi bien la gestion des rendez-vous que la préparation et la réalisation des conférences et autres réunions de travail

La planification des tâches

Les solutions logicielles proposées (telles que Lotus Notes ou Exchange) sont plus axées sur la diffusion des informations que sur la protection des données, credo de SQL. Ici, c'est l'accès des données qui constitue l'objectif prioritaire lors de la réalisation d'un projet.

1.2.5 Les objets répartis

1.2.5.1 Définition

Un objet réparti est un composant logiciel indépendant contenant sa propre intelligence de fonctionnement et capable d'interopérer avec d'autres objets distribués, indépendamment des types d'ordinateurs, des langages de programmation ayant servi à les développer, ou des systèmes d'exploitation.

A l'heure actuelle, trois systèmes sont fonctionnels.

Le système CORBA (Common Object Request Broker Architecture), dû à l'OMG (Object Management Group), un groupe de travail rassemblant de nombreux constructeurs et éditeurs de logiciel dans le but de fournir une base de travail indépendante de la plate-forme.

(D)COM / OLE / Active X (Distributed Component Object Model / Object Linking and Embedding) concurrent direct de CORBA que cherche à imposer Microsoft depuis les plate-formes de type Windows.

Java/Beans (proposé par SunSoft, filiale logiciel de Sun) système d'objets distribués axé sur l'utilisation du langage Java. Son rapprochement avec le système CORBA est indéniable.

1.2.6 Les apports des objets distribués

Les objets distribués vont radicalement modifier les phases de conception, développement et distribution des logiciels dans le sens où un objet devient à la fois une unité de distribution, de déploiement et de maintenance orientée réseau !

Les nouvelles applications seront plus intelligentes dans la mesure où elles seront à l'écoute du réseau afin de répondre à diverses sollicitations sous forme de messages (transactions, collecte des données, etc ...).

Les composant s'assembleront de manière dynamique pour créer des applications à durée de vie variable (voire à exécution unique) s'adaptant parfaitement aux besoins d'un utilisateur ponctuel.

1.2.6.1 La technologie sous-jacente : le bus à objets

Le bus à objets (egalement appelé courtier en objets par référence à la terminologie anglaise Object Request Broker), permet aux objets de communiquer intelligemment en leur permettant de s'envoyer des requêtes (et les réponses associées), de s'associer librement, etc ... Le niveau de collaboration entre objets est ainsi très élevé alors qu'un maximum d'information non indispensable est masqué à l'utilisateur.

1.2.6.2 La notion d'interface

Comme dans toute technologie orientée objet, les détails d'implémentation sont masqués à l'utilisateur et celui-ci ne connaît d'un objet que sa partie interface.

Dans les technologies à objets répartis, l'interface est un ensemble d'attributs et de méthodes que l'objet désire mettre à disposition de ses clients. Un objet peut avoir plusieurs interfaces. Par exemple, si l'on considère un objet modélisant un compte en banque, il pourrait présenter deux interfacesé :

1) La première, concernant plus précisément le détenteur du compte, lui permettrait de consulter son solde, effectuer un dépôt ou un retrait.

2) La seconde, réservée au banquier, permettrait de créer le compte, de le clôturer, d'ajouter les intérêts ... ou de prélever des agios.

Bien entendu, tous les objets d'une même classe partagent les mêmes interfaces et la même implémentation.

Pour que les systèmes répartis puissent fonctionner correctement, les courtiers tiennent à jour un annuaire des objets accessibles avec leurs interfaces. C'est pourquoi, une fois les interfaces définies, elles doivent être exportées. Par exemple, dans le système OLE, la base de registres contient les descriptions d'interface et de localisation des objets présents sur un serveur.

1.2.6.3 Héritage et polymorphisme

Tous les systèmes d'objets distribués fonctionnels supportent les mécanismes d'héritage et de polymorphisme sur l'interface. C'est-à-dire que l'acheminement d'un même message d'interface se fera correctement en fonction des références à un objet. Par exemple, les méthodes dédiées utilisateur d'un compte bancaire simple pourront sans doute s'appliquer à un CODEVI avec, éventuellement, de nouvelles adjonctions.

En revanche, l'héritage d'implémentation n'est supporté que par CORBA et JavaBeans. Les concepteurs de Microsoft ont préféré introduire de la délégation sous forme d'agrégation. Par exemple, un CODEVI présentera l'interface standard d'un compte bancaire mais comme on ne pourra hériter directement du code, une possibilité consiste à agréger un compte bancaire simple dans un objet CODEVI et à lui faire exécuter les traitements idoines.

1.2.6.4 La requête vue du côté du client

Le rôle du client se limite à demander les services d'un objet. L'administration de ce dernier (création, relecture sur disque, destruction, administration des attributs persistants et implémentation) n'étant pas à sa charge.

Dans tous les cas, un client accède à un objet au travers d'une référence et passe des requêtes au travers de son interface. La notion de référence évite au client de savoir sur quel serveur se situe l'objet destinataire.

L'envoi d'un message à un objet distant peut se concevoir en plusieurs étapes :

1) Obtenir (via le bus à objets) une référence à l'objet récepteur du message.

2) Obtenir éventuellement son interface dans le cas d'un appel dynamique (toujours via le bus à objets).

3) Construire la requête proprement dite :

   ⇒ Référence de l'objet destinataire

   ⇒ Identifiant de la requête

   ⇒ Paramètres

   ⇒ Adresse de retour (c'est-à-dire, la référence du client)

4) Attendre éventuellement la réponse en provenance de l'objet cible.

1.2.6.5 La requête vue du côté du serveur

Lors de la réception d'une requête pour un objet, le bus objet recherche son serveur et vérifie son état (actif ou inactif)

1) En cas de non activité du serveur, celui ci est réveillé. Cette phase est habituellement dépendante du système d'exploitation hôte du serveur. Si l'objet possède des données persistantes, celles-ci sont chargées.

2) Le serveur recherche la méthode spécifiée par la requête et l'exécute. Les résultats éventuels sont renvoyés au client via le bus à objets.

3) Si d'autres requêtes arrivent pour ce même objet durant le traitement, la gestion de celles-ci dépend des spécifications relatives à la concurrence sur les méthodes demandées.

Si le serveur doit s'arrêter, il doit en avertir le bus à objets afin que celui-ci lui demande de se réveiller pour le prochain traitement. En outre, il se doit de sauvegarder toutes les informations persistantes inhérentes aux objets qu'il détient.

2. CORBA

2.1 Historique

Depuis le début des années 80 et l'émergence des technologies orientées objet, les fournisseurs de matériel, de logiciels ainsi que des utilisateurs se sont réunis au sein d'un groupe de travail afin d'échanger leurs points de vue sur tous les sujets concernant la technologie objet.

Au fil du temps, ce groupe de travail est devenu un consortium regroupant la plupart des grands noms de l'informatique : l'OMG ou *Object Management Group*. A l'heure actuelle, l'OMG est le plus grand regroupement d'organismes informatiques (plus de 500 membres).

L'objectif de l'OMG est de définir des standards permettant de développer et de distribuer aisément des applications et des méthodologies pour des systèmes hétérogènes et distribués utilisant des technologies objets.

Les principaux concepts retenus pour l'élaboration de leurs documents sont les suivants :

Réutilisabilité des composants d'application

Interopérabilité en environnement hétérogène

Portabilité du code

Afin d'atteindre ces objectifs, l'emploi des technologies reposant sur les objets est indispensable. Toutefois, ceci pose des problèmes dans le sens où les trois concepts de base que sont l'encapsulation, l'héritage et le polymorphisme donnent lieu à des mises en œuvre différentes selon les langages de programmation utilisés. Aussi, l'une des premières tâches de l'OMG a été de définir une architecture orientée objet unique : l'OMA (*Object Management Architecture*) qui pose les bases d'un modèle de référence auquel les développeurs de systèmes devront se conformer. Cela contribue à décrire une architecture globale d'intégration de services orientés objet.

L'homogénéité des déclarations est assurée par l'utilisation d'un langage de description spécifique à CORBA : IDL

2.2 Spécifications de CORBA

CORBA est un bus à objets répartis offrant un environnement d'exécution pour le modèle OMAG. Il autorise la coopération d'objets développés dans des langages différents et surtout, reposant sur des systèmes d'exploitation et des architectures matérielles hétérogènes.

Le bus à objets de CORBA utilise un protocole unique de transmission des messages entre objets distribués. Afin d'assurer la coopération d'objets écrits dans des environnements hétérogènes, chaque objet est muni d'une souche (*stub*) qui assure l'encodage et le décodage des messages du format natif vers le format fixe de CORBA *(Common Data Representation).*

2.3 Le modèle Objet de CORBA

Le modèle Objet de CORBA s'appuie sur 9 définitions

L'application cliente envoie des messages à des objets via le bus CORBA.

La référence d'un Objet Corba est une métadonnée qui permet au bus CORBA de localiser un Objet où qu'il soit situé, que ce soit dans le même processus ou sur une machine distante à travers le réseau.

L'interface définit l'ensemble des attributs consultables et des méthodes d'un objet CORBA. En particulier, elle définit le prototype de chaque méthode. L'interface est définie au moyen d'un langage standard : IDL (Interface Definition Language)

Les requêtes sont le moyen standard d'invocation d'une opération sur un objet. Lorsque l'on souhaite appliquer une opération sur un objet, il faut lui envoyer une requête via sa référence et son interface. La requête contient, (outre la référence de l'objet cible), le nom de l'opération demandée ainsi que la liste des paramètres en entrée. A la suite de l'exécution de la méthode concernée, le bus CORBA renvoie le résultat : les paramètres en sortie ou alors, une erreur.

L'objet CORBA. C'est un composant logiciel complet, désigné par sa *référence* et apte à exécuter diverses opérations invoquées via des *requêtes*, lesquelles sont définies dans son *interface*. Le traitement en lui même est réalisé via *l'implémentation de l'objet* qui lui est associée par le *processus d'activation*.

L'implémentation d'un objet est une entité qui code un objet CORBA. Par exemple, une structure C, une instance d'une classe C++ ou JAVA ou alors un objet de base de données orientée objet. L'implémentation stocke les attributs de l'objet et fournit du code pour les méthodes. Cette association est, par définition, limitée dans le temps et réalisée par le processus d'activation.

Le code d'implémentation d'un objet réalise effectivement le traitement associé aux messages de l'interface d'un objet (par exemple, les méthodes d'une classe). Ce code est rassemblé dans des exécutables, des librairies ou des scripts.

Le processus d'activation associe un objet d'implémentation à un objet CORBA. Lorsqu'un objet reçoit une requête, ce processus active un objet d'implémentation et lui transmet le message. La seconde opération consiste à charger (à la demande de l'objet d'implémentation), le code d'implémentation relatif à l'opération exigée, ce qui peut exiger, par exemple, de charger en mémoire un exécutable. Il est à noter qu'un même objet CORBA peut se voir associé à divers objets d'implémentation au cours du temps.

L'application serveur offre une structure qui stocke les objets d'implémentation et le code des opérations. C'est habituellement une fonctionnalité du système d'exploitation ou d'un processus démon.

2.4 La structure de l'Object Management Architecture

L'OMG a défini une architecture logicielle d'applications réparties (OMA), qui, si elle est respectée, garantit un maximum d'interopérabilité entre composants. Cette « norme » est décrite dans un document : *Object Management Architecture Guide*.

2.4.1 Généralités et décomposition en 5 couches

L'architecture générique définit une gamme d'outils complète pour permettre d'écrire des applications réparties. Les fonctionnalités sont décrites en 5 couches isolant les fonctionnalités de communication pures, les primitives de système bas niveau, les composants logiciels de haut niveau, les objets de métiers et les objets applicatifs.

Le bus à objets répartis (*Object Request Broker*) assure le transport des requêtes en masquant les différences d'implémentation, de système d'exploitation etc ... ainsi que la localisation géographique des objets.

Les services orientés objet communs (*CORBA Common Object Services*) définissent une couche proposant une abstraction des services orientés système. Par exemple, tout ce qui concerne le cycle de vie, le nommage, les transactions ou la sécurité est englobé dans ces objets.

C'est donc principalement cette couche qui assure l'indépendance des objets CORBA vis-à-vis des systèmes d'exploitation.

Les utilitaires communs (*CORBA Facilities*) fournissent une interface de plus haut niveau aux éléments communs aux applications : l'interface utilisateur, la gestion de documents composites, l'administration du système et la gestion des tâches.

Les interfaces de domaines (*Domain Interfaces*) fournissent une gamme d'objets pour un domaine d'activité précis, on les appelle les *objets de métier* ou encore *objets métiers*. A l'heure actuelle, l'OMG s'est penché sur les domaines suivants :

⇒ La santé

⇒ Les autoroutes de l'information

⇒ Les télécommunications (en particulier, les visioconférences)

⇒ La finance

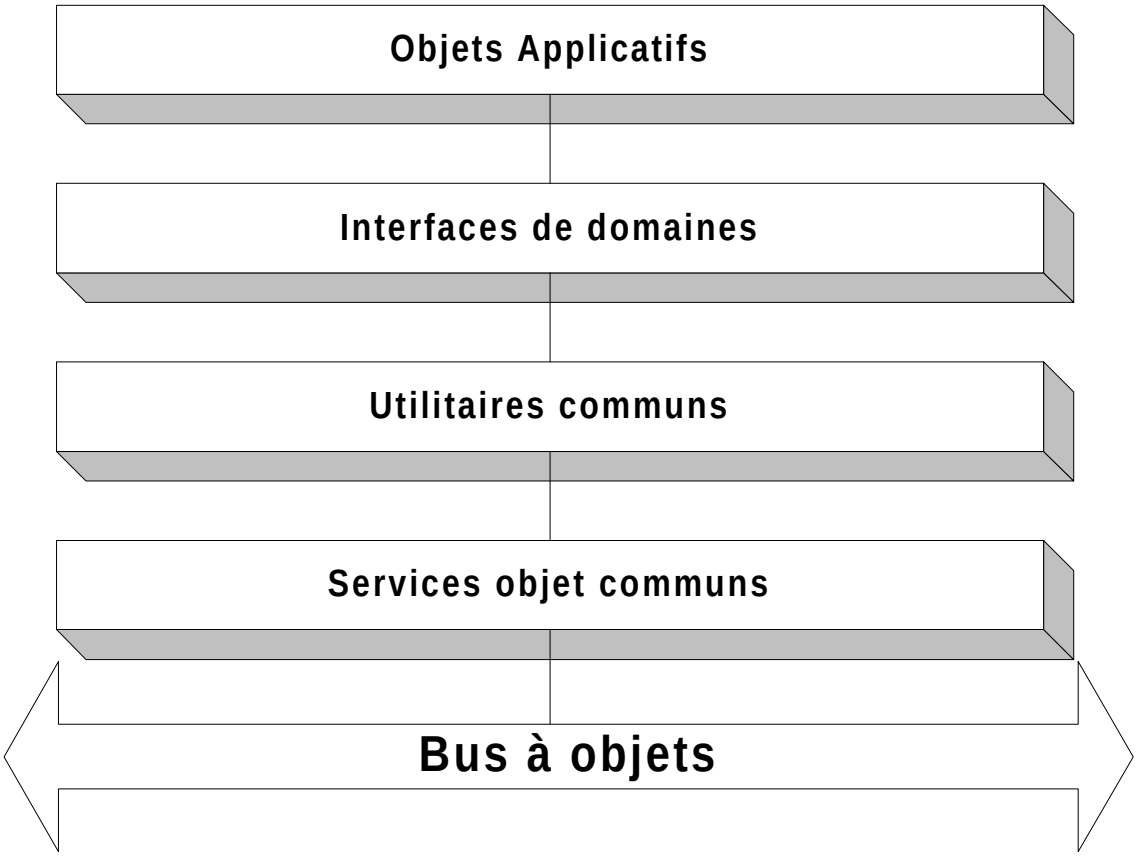
⇒ Le commerce électronique

⇒ La fabrication assistée par ordinateur

⇒ La simulation répartie

Les objets applicatifs répondent spécifiquement aux besoins d'une application. Ils ne peuvent donc pas être standardisés. Toutefois, dès qu'ils commencent à devenir génériques pour un domaine, il est recommandé de demander leur inclusion dans les interfaces standard des domaines concernés.

La figure suivante décrit l'architecture logicielle CORBA.



Les prochains paragraphes décrivent les 3 premières catégories de composants.

2.4.2 Le bus à objets CORBA

Le bus à objets CORBA fournit l'infrastructure de communication (*Middleware)* nécessaire à la création d'applications réparties. Après avoir étudié ses différentes caractéristiques, nous nous intéresserons à son architecture.

2.4.2.1 Les caractéristiques du bus CORBA

Indépendance vis à vis des langages de programmation. La grande force de CORBA réside dans la séparation entre la définition des interfaces des objets dans un langage unique (IDL) et l'implémentation de ces mêmes objets dans un langage de programmation quelconque.

Les précompilateurs IDL sont chargés de gérer des *souches de communication* en langage cible à partir des interfaces. Ces souches assurent la liaison entre un objet écrit dans un langage de programmation et le bus CORBA. Ainsi, tout langage de programmation pour lequel il existe une précompilation (ou *projection*) IDL est susceptible d'être utilisé pour implémenter des objets CORBA.

La transparence des invocations. L'utilisateur d'un objet CORBA a toujours l'impression d'invoquer un objet local alors que celui-ci peut aussi bien résider dans le même processus qu'à des milliers de kilomètres sur une autre machine.

Le bus se charge ainsi d'utiliser le meilleur moyen de transport pour chaque requête et ce de manière transparente. En outre, il masque les détails d'implémentation (techniques de codage utilisées) et d'implantation (localisation physique sur machine) de chaque objet.

Des protocoles d'invocations statique et dynamique. CORBA propose deux manières pour invoquer des opérations d'un objet. L'*invocation statique* (mécanisme le plus courant) suppose que le client connaisse l'interface de l'objet serveur au moment de la compilation du code utilisateur. C'est alors le compilateur qui se charge de vérifier l'adéquation des paramètres, etc ...

Dans le cas de *l'invocation dynamique*, la requête est construite au moment de l'exécution avec des informations en provenance du *référentiel d'interfaces* ; ce dernier contenant les descriptions des interfaces enregistrées sur le bus (cf § suivant). Ce mécanisme (au demeurant très coûteux en temps de calcul) est particulièrement puissant et souple car il permet d'invoquer des traitements à partir de n'importe quel objet.

Auto-description du système. Le bus CORBA maintient en permanence un répertoire décrivant les interfaces des objets du système : le *référentiel d'interfaces*. Ce dernier met à disposition de tous ces clients une version précompilée des interfaces IDL des objets présents sur le Bus. Ainsi, tout client peut consulter cette base de données à la recherche d'objets susceptibles de lui fournir un service particulier, ou pour savoir quels services sont mis à disposition par un objet prédéterminé.

Le référentiel des interfaces est également une aide non négligeable pour le développeur qui dispose de l'ensemble des services proposés au moment de prendre une décision.

Activation automatique des objets. Il n'est pas souhaitable que tous les objets présents sur un bus CORBA soient toujours chargés en mémoire sur leurs serveurs respectifs ; en particulier si ce sont des objets dits *patrimoine* qui sont en fait des adaptations à CORBA de gros programmes COBOL. Aussi, le bus fournit il un mécanisme totalement transparent d'activation dynamique des objets lorsqu'ils reçoivent une requête.

Au cours de l'activation d'un objet, le bus associe à une interface :

⇒ Un état de cet objet (notamment, des données persistantes)

⇒ Un contexte d'exécution (le code des méthodes de l'objet activé)

Communication entre différents bus. La norme CORBA n'imposant aucune définition de l'implémentation de chaque bus, il a fallu assurer l'interopérabilité entre des bus en provenance de fournisseurs différents. A ce sujet divers mécanismes sont proposés.

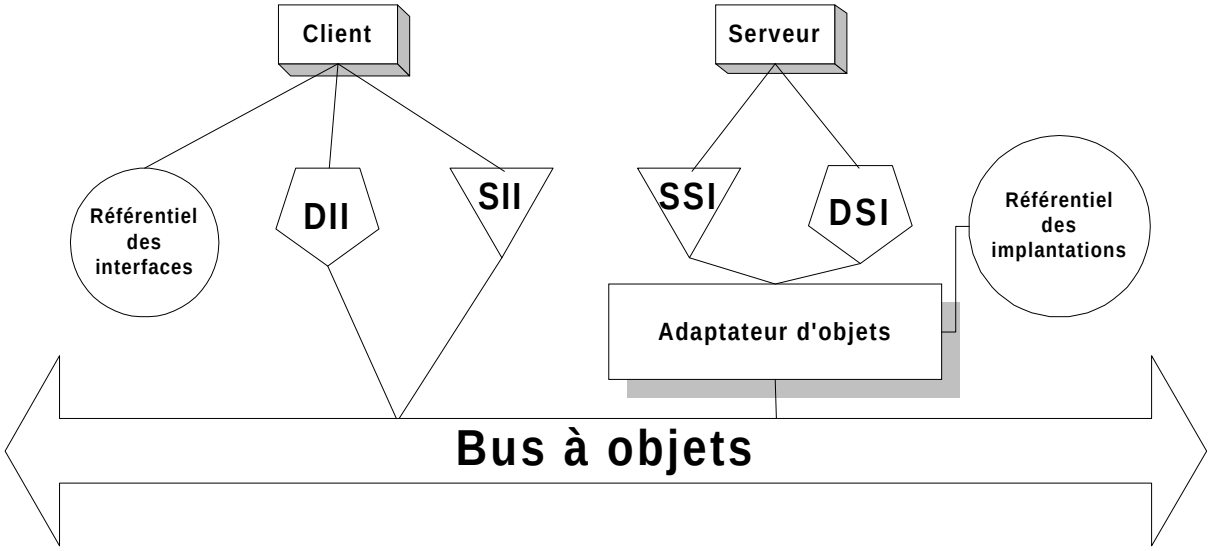
Les informations circulent en format CDR *(Common Data Representation)* qui formalise et fixe la représentation des types de données de l'IDL pour qu'ils soient transportables par les couches de communication. Théoriquement, la représentation est la même pour chaque implémentation du bus. Toutefois, dans le cas de la communication entre bus CORBA hétérogènes, le bus chargé de la réception des requêtes peut être amené à effectuer du décodage. C'est pourquoi chaque message CORBA est préfixé d'un identificateur indiquant le format utilisé par l'émetteur.

La norme prévoit l'existence de passerelles convertissant les requêtes d'un type de bus vers un autre.

Le mécanisme le plus évolué repose toutefois sur l'utilisation de protocoles communs de communication entre bus. A ce sujet, la norme a défini un protocole générique nommé *GIOP Generic Inter-ORB Protocol* dont l'application directe à Internet est *IIOP Internet Inter-ORB Protocol*.

2.4.2.2 Architecture du bus CORBA

L'architecture du bus CORBA repose sur 9 blocs dont les fonctionnalités sont normalisées mais dont l'implémentation est laissée libre. La figure suivante les récapitule.



Le noyau de communication (indisponible à l'utilisateur) est chargé du transport des requêtes et de leurs résultats. En fonction de la différence de localisation du client et du serveur, ce transport peut se résumer à un simple appel de procédures, l'utilisation de fonctionnalités inter-processus locales ou bien des techniques réseau.

L'interface d'invocation statique (*Static Invocation Interface*) est générée automatiquement lors de la projection de la définition d'interfaces IDL vers un langage cible sous la forme de prototypes de méthodes. Lors de la compilation du code client, le compilateur s'assure de l'adéquation du code d'appel et de la définition. L'interface d'invocation statique est chargée de composer la requête (en particulier, l'emballage des paramètres) et d'interpréter les résultats ; toute la partie transport étant dédiée au noyau de communication.

L'interface d'invocation dynamique (*Dynamic Invocation Interface*) est utilisée dès lors que l'on souhaite invoquer les services d'un objet dont on ignorait l'interface au moment de la compilation. Par exemple, un objet renvoyé par un autre ou dont on a acquis la référence via un service d'annuaires. La création d'une requête dynamique est plus compliquée que la création d'une requête statique. En effet, le programmeur doit fournir l'ensemble des composantes de la requête : référence de l'objet cible, nom de l'opération demandée ainsi que la liste des paramètres dont la liste des spécifications est obtenue par un appel sur une interface spéciale. C'est précisément la puissance et la souplesse d'un tel mécanisme qui donne toute sa richesse aux objets distribués.

Ajoutons que l'interface d'invocation dynamique permet de lancer une requête de manière *asynchrone* alors que l'interface d'invocation statique ne le permet pas. Dans le cadre de l'invocation statique, on doit attendre le retour de la requête avant de poursuivre les traitements engagés.

Le référentiel des interfaces (*Interfaces Repository*) contient (sous forme compilée), les déclarations IDL de tous les objets présents sur le bus. Comme cette base de données est elle même encapsulée dans un objet CORBA, son interface est également décrite en IDL. Elle offre un support incomparable pour les mécanismes d'invocation dynamiques.

Le référentiel des implémentations (*Implementations Repository*) est une base de données recelant l'ensemble des informations relatives aux implémentations des objets : noms des exécutables contenant le code, politiques d'activations, droits d'accès, etc ... ainsi que des renseignements nécessaires à l'administration des objets. Non normalisée, cette fonctionnalité est laissée libre pour chaque implémentation de CORBA.

L'interface du bus (*Bus Interface*) fournit une série de primitives concernant la connexion d'une application au bus CORBA ou l'activation de ce dernier. Par exemple, elle fournit un message permettant de récupérer la référence du référentiel d'interfaces.

L'interface de squelettes statiques (*Static Skeleton Interface*) est le pendant de l'interface d'invocation statique du côté de l'objet cible d'une requête. Lorsque la couche de transport apporte une requête, le squelette statique (pendant de la souche statique généré) est chargé de déballer le message pour le présenter à l'objet concerné. Après exécution de la méthode requise, elle emballe les résultats avant de les présenter au noyau de communication pour retour à l'initiateur de la communication. Sa génération est assurée par le projecteur IDL en langage cible.

L'interface de squelettes dynamiques (*Dynamic Skeleton Interface*) permet de traiter des requêtes pour lesquelles on ne disposait pas de squelette statique. Un exemple typique d'utilisation de cette interface est la réalisation de passerelles entre des bus provenant de fournisseurs différents.

Il est important de noter que l'interface de squelettes dynamiques n'est utilisée que dans le cas où un objet reçoit une requête qu'il ne connaît pas. En effet, si la requête reçue possède un squelette statique, celui-ci est utilisé, que la requête ait été invoquée de manière statique ou de manière dynamique.

L'adaptateur d'objets (*Object Adapter*) est le bloc fonctionnel responsable de l'exécution des requêtes. A l'aide des informations de localisation fournies par le référentiel des implémentations, l'adaptateur d'objets est capable de traiter les problèmes d'activation, de vérification de droits etc ... En théorie, n'importe quel type de substrat d'objets est susceptible d'être utilisé par un adaptateur. Toutefois, à l'heure actuelle seul l'adaptateur spécialisé dans le traitement des exécutables (BOA *Basic Object Adapter*), est partiellement normalisé. Les adaptateurs suivants sont également à l'études :LOA (*Library Object Adapter*) et DOA (*Database Object Adapter*).

2.4.3 Les services orientés objet communs

Les services orientés objet communs sont des collections d'objets CORBA assurant des fonctionnalités systèmes de bas niveau telles que le nommage, la gestion de la persistance et des transactions etc ...

Le fait d'isoler ces traitements dans une couche séparée du bus permet de laisser ce dernier se concentrer sur les communications. Il y a 16 services orientés objet identifiés par l'OMG. Comme tout le reste du système CORBA, ils sont spécifiés en IDL. Leur utilisation systématique permet de faciliter l'intégration des objets dans le système global. En outre, si les choix concernant l'implémentation de tels services sont laissés au libre arbitre des fournisseurs, toutes les interfaces sont elles fixées et exportées par l'OMG.

A l'heure actuelle, seuls quelques services sont complètement normalisés et proposés en standard par les fournisseurs. La disponibilité de tel ou tel service est un critère important lors du choix d'un fournisseur de système CORBA.

Cycle de vie : service qui décrit les interfaces et les traitements assurant la création, la destruction, la duplication d'objets ou leur migration entre différents serveurs. Il s'appuie sur le concept de fabrique d'objets (*Object Factory*)

Nommage : ce service fournit des opérations permettant de retrouver un objet à partir de son nom symbolique ou d'affecter un nom symbolique à tout objet lors de sa création et ce à partir de sa référence. Un peu à l'instar des URL, les noms sont gérés sous la forme d'une arborescence de répertoires où chaque feuille contient la référence d'un objet.

Vendeur : est un service qui permet d'identifier les objets (et les traitements associés) en fonction d'une demande de service envoyée par un client. De ce point de vue, le service Vendeur fonctionne à la manière des pages jaunes d'un annuaire. Les fournisseurs de services s'enregistrent dans une base de données. Lorsqu'un client nécessite un service, il passe par le service vendeur pour émettre une expression de besoins et reçoit une ou plusieurs référence d'objets avec les interfaces associées.

Persistance : propose des interfaces pour assurer la gestion des objets persistants, c'est-à-dire des objets dont l'état doit être sauvegardé de façon stable d'une invocation à la suivante. Ces services reposent essentiellement sur des systèmes de sauvegarde sur fichier de l'état des objets, que ce soient des systèmes de bases de données ou de simples fichiers.

Evènements : permet de gérer les objets par envoie d'événements asynchrones, comme, par exemple, des événements émanant d'un gestionnaire d'interface utilisateur. La gestion des événements se fait au travers de *canaux d'événements* reliant un ou des *producteurs* à un ou des *consommateurs*.

Concurrence d'accès : propose (toujours au travers d'interfaces IDL) tout un système de verrous permettant de contrôler les accès concurrents à un objet.

Transactions : autorise le traitement de transactions complexes qui peuvent mettre en jeu de nombreux objets (voir des objets sur différents bus), des transactions imbriquées, des transactions de longue durée, etc ... en utilisant les fonctionnalités du service **Concurrence d'accès**.

Relations : propose de définir des liens dynamiques (et donc à durée de vie limitée) entre des objets. Les relations entre objets sont typiques du modèle orienté objet : Avoir, Communiquer, Etre.

Propriétés : à rapprocher de Relations, ce service permet d'associer dynamiquement aux objets des attributs non prévus par l'interface mais directement sous la responsabilité du client. Un exemple typique d'utilisation est celui des dates de péremption. Ces attributs doivent impérativement être nommés, d'où le nom de *propriétés*.

Externalisation : A la base de la migration ou de la sauvegarde d'objets, ce service offre des mécanismes de base pour la gestion de l'état des objets sur le bus.

Licences : ce service, orienté facturation et authentification, permet de spécifier des droits d'accès vers des objets. Les licences sont accordées à vie, par requête, par session de travail ou toute autre période. Ainsi, les clients sont authentifiés pour la facturation des services demandés.

Sécurité : propose de nombreuses fonctionnalités d'authentification, identification, certification et cryptage sur les objets et les requêtes. Il s'appuie directement sur le service Licences et permet d'assurer l'intégrité des objets présents sur un bus en environnement ouvert.

Interrogations : basé sur les langages d'interrogation de bases de données de haut niveau (tels que SQL3 et OQL), ce service permet d'interroger les attributs des objets le long du bus sous forme de requêtes au sens bases de données. Les résultats sont typiquement des ensembles d'objets.

Collections : est un service fortement relié au précédent. En effet, il permet de traiter en bloc tous les objets obtenus par interrogation ou tout autre mécanisme de génération de collections d'objets. Par exemple, en leur envoyant à tous la même requête.

Temps : ensemble d'interfaces relatives à *l'horloge globale d'un bus*. Permet de gérer l'écoulement du temps, mesurer des intervalles ou synchroniser des processus.

Changements : permet de suivre l'évolution des objets (et, en particulier le suivi de leur conception et de leur mise à disposition) en affectant des numéros de version aux différents objets présents sur le bus. Des informations relatives aux différences d'interface et d'implémentation sont disponibles.

2.4.4 Les utilitaires communs

Les utilitaires communs sont des composants logiciels réutilisables de plus haut niveau que les services objet communs. Au nombre de 4, ils vont fournir des canevas de développement à 4 grands pôles d'une application : l'interface utilisateur, la gestion de l'information, l'administration du système et la gestion des tâches. Non content de faciliter le développement de ces secteurs dans une application, ils leurs garantissent également une certaine indépendance en proposant une séparation fonctionnelle *de facto* avec une liaison par envoi de messages entre les objets les constituant.

2.4.4.1 L'interface utilisateur

La normalisation de la gestion de l'interface utilisateur est basée sur 5 points clef :

La gestion du rendu qui doit être uniforme d'une application à une autre.

La gestion des documents composites avec édition *in situ* des différents composants à l'intérieur du document conteneur.

Le support de l'utilisateur qui doit reposer sur des outils standards. Par exemple la gestion de l'aide ou la vérification de l'orthographe doit reposer sur les mêmes utilitaires.

La gestion du bureau qui doit proposer une vision orientée objet de l'univers de travail de l'utilisateur.

Les scripts utilisateurs qui permettent d'automatiser certaines tâches répétitives.

A l'heure actuelle, la spécification *OpenDoc* proposée conjointement par Apple et IBM a été choisie comme base de travail et d'uniformisation de l'offre en matière de documents composites.

2.4.4.2 La gestion de l'information

L'accès à l'information est aujourd'hui primordial. Afin de le faciliter, l'OMG travaille sur la standardisation de tous les maillons de la chaîne de l'information.

Modélisation des données : ici, le but de l'OMG est de fédérer les diverses stratégies de structuration, de manipulation et d'accès aux données en intégrant les diverses technologies existantes dans des descriptions IDL communes.

Stockage structuré : basé sur le service persistance, cet utilitaire de niveau supérieur gère des métadonnées permettant de retrouver plus facilement les informations stockées. La base choisie pour travailler est *Bento,* complément d'OpenDoc.

Echange de données : ce canevas a pour but de faciliter l'échange de données entre les applications. Par exemple, il définit des protocoles pour les formats de fichier standard du marché (RTF, JPG, MPEG, HTML, etc ...)

Codage et représentation : propose des outils standard de conversion de format et de compression de données ainsi que des utilitaires favorisant la mise en forme des données pour les communications.

2.4.4.3 L'administration du système

Sans vouloir entrer dans les détails, l'utilitaire commun d'administration du système (développé en collaboration avec d'autres organismes de normalisation tels que X/Open) propose toute une série d'outils de haut niveau dédiés administration. Les composants choisis sont les suivants :

⇒ L'instrumentation

⇒ Collecte de données

⇒ Sécurité

⇒ Suivi d'instance

⇒ Ordonnancement

⇒ Qualité du service

⇒ Gestion des événements

L'utilisation de ce paquetage doit permettre, par exemple, de définir des comportements standard en cas de panne de système ou d'automatiser simplement un système de sauvegarde sélective.

2.4.4.4 La gestion des tâches

La gestion des tâches est un paquetage de haut niveau destiné à simplifier l'existence des utilisateurs. Il est basé sur l'utilisation de *scripts* interprétables par l'environnement CORBA en chaque point du réseau. La notion de *tâche* CORBA est très générale car elle englobe aussi bien une action atomique sur un objet particulier que tout le processus de gestion d'un projet. En fait, la norme CORBA prévoit tout un panel de tâches répertoriées selon leur durée, leur complexité, etc ... Les outils permettant de manipuler les tâches sont les suivants :

Flux d'activités : typiquement orientées *groupware*, les interfaces de flux d'activités comprennent, bien entendu des services (persistants) pour gérer les *worflows* mais aussi une gestion évoluée de transactions de longue durée.

Agents : les agents sont des objets de très haut niveau orientés recherche d'information sur le réseau. Les plus intéressants sont des agents mobiles qui se déplacent sur le réseau en fonction des informations qu'ils découvrent. Ceux-ci sont définis par un script interprété par un moteur de création sur la nouvelle machine qu'ils désirent explorer. Afin d'éviter toute prolifération sauvage qui entraînerait nécessairement une protection massive des données, CORBA définit tout un canevas d'accueil, d'ouverture de droits et de communications privilégiées pour les agents.

Règles d'activation : les règles d'activation permettent de définir des comportements en réponse à certains événements. Leur utilisation repose sur l'utilisation de *moteurs d'inférence* capables de reconnaître les situations satisfaisant un ensemble de prédicats avant de lancer l'action associée.

3. COM/OLE/ActiveX

3.1 Introduction

L'origine du système à objets distribués proposé par Microsoft date du début des années 90 lorsque sont apparus les premiers documents composites du monde Windows. L'enjeu était alors de pouvoir incorporer des feuilles de calcul Excel ou des textes Word dans des pages Powerpoint.

Ce dispositif ayant rapidement rencontré le succès que l'on lui connaît, le système OLE a du évoluer de manière à intégrer plus de mécanismes de communication.

Il est intéressant de noter que les développements des normes CORBA et OLE ont été totalement différents. Si les concepteurs de CORBA ont construit des standards *ex nihilo* afin d'intégrer dès le départ autant d'éléments que possible, OLE s'est bâti petit à petit pour résoudre itérativement les problèmes posés par les lacunes de ses versions antérieures ou pour répondre aux nouvelles exigences de ces concepteurs.

Initialement limité à Windows, OLE s'étend sur un nombre de plus en plus important de plate-formes à l'initiative de Microsoft et de prestataires de services chargés d'assurer le portage de ses fonctionnalités.

3.2 Les diverses fonctionnalités d'OLE

Nous donnons ici une vision synthétique des différentes composantes du système d'objets distribués de Microsoft.

Le Modèle COM (Common Object Model) définit l'architecture générale (notamment en termes de format binaire d'exécutables) sur laquelle s'appuie ces fonctionnalités.

Les services orientés documents composites représentent la motivation initiale d'OLE. A ce sujet, la réussite de Microsoft est indéniable. Les services d'ores et déjà disponibles proposent les fonctionnalités suivantes :

⇒ Stockage et archivage homogène de documents composites

⇒ Echange de données entre applications–glisser/déposer

⇒ Edition en place des composants dans le conteneur (faire un double clique sur un tableau Excel à l'intérieur d'un texte Word appelle Excel pour réaliser une édition dudit tableau)

⇒ Modularité des documents

Programmation de clients et serveurs de services. C'est la partie la plus proche de l'ORB de CORBA. Elle repose sur un concept nommé *OLE Automation* qui permet à certains objets d'exporter les définitions des méthodes que leurs clients pourront appeler. On dispose ainsi *d'objets automates* (les serveurs) et de *contrôleurs d'automates* (les clients).

Les contrôles OLE ou ActiveX sont destinés à promouvoir l'architecture OLE dans le contexte d'Internet et du World Wide Web. A ce propos, ActiveX propose une infrastructure unique pour tous les services OLE :

⇒ Document composites

⇒ Utilisation d'objets OLE dans les pages WEB

⇒ Composants visuels déportés (les fameux *contrôles ActiveX*)

⇒ Communication client-serveur

Ce sont les contrôles ActiveX qui permettent, par exemple, de lancer automatiquement Excel pour visualiser un document feuille de calcul intégré dans une page Web.

3.3 Le modèle COM

A l'heure actuelle, le modèle COM (Common Object Model) est l'élément unificateur de l'architecture OLE. Apparu après OLE, son développement s'est avéré nécessaire dès lors que les différentes fonctionnalités d'OLE ont eu besoin de reposer sur une base commune.

COM repose sur deux principes fondamentaux :

1) L'interaction entre fichiers binaires exécutables écrits dans des langages de programmation différents. A ce sujet, il définit un format commun des fichiers binaires exécutables destiné à promouvoir leur uniformisation.

2) Les interfaces des objets. A l'instar de CORBA, les parties publiques des objets sont décrites dans leur *interface*. Celle-ci est toutefois différente du modèle IDL car de plus bas niveau d'abstraction. Nous reviendrons plus en détail sur ce sujet essentiel mais pour l'instant, il suffit de savoir qu'un même objet peut présenter plusieurs interfaces et que l'on peut uniquement accéder aux informations d'un objet au travers de celles-ci.

3.3.1 Le format commun de fichiers binaires

Considérons une application complexe séparée en diverses unités cohérentes sous la forme de fichiers exécutables plus petits. Sous Windows, ce seront typiquement des exécutables .EXE ou des DLL (Dynamic Link Library).

Lorsque le programme « principal » nécessite une fonctionnalité délocalisée, il procède ainsi :

1) Chargement de l'unité considérée

2) Utilisation de la fonctionnalité (le plus souvent, un appel de procédure)

3) Libération de la mémoire

Ce processus, aussi simple soit-il n'est pas sans poser de nombreux problèmes dont nous retiendrons les 2 plus graves :

1) Les architectures des différents EXE ou DLL sont dépendantes, non seulement des langages de programmation utilisés mais également des environnements ayant présidé à leur création.

2) Changer une procédure impose de redistribuer complètement l'unité (EXE ou DLL) qui la contient.

Tout ceci est, rappelons-le, contraire aux principes du développement orienté objet qui prône l'indépendance de l'architecture vis-à-vis des langages de programmation et qui spécifie qu'une modification localisée d'implémentation ne doit pas affecter l'intégrité de l'ensemble.

C'est dans ce contexte que COM propose un format unique pour les EXE et les DLL, indépendant des langages de programmation et autres environnements de développement. Notons toutefois que Microsoft privilégie fortement l'utilisation du C++.

3.3.2 COM et le modèle orienté objet

La question fondamentale est la suivante :

COM respecte-t-il les grands principes du modèle orienté objet ?

Nous répondrons à cette question en considérant les trois points essentiels suivants : l'encapsulation, le polymorphisme et l'héritage.

3.3.2.1 COM et l'encapsulation

La séparation de l'interface et de l'implémentation est présente dans le modèle COM même si elle est moins nette que dans le modèle CORBA. De même, on ne peut accéder uniquement à un objet à travers les méthodes décrites dans son interface ce qui rejoint bien le concept d'abstraction de données.

3.3.2.2 COM et le polymorphisme

COM supporte bien le polymorphisme que ce soit au niveau des interfaces ou des méthodes. En effet, si plusieurs objets présentent les mêmes interfaces, on accède de la même manière aux informations qu'elles contiennent. De la même manière, deux méthodes de même spécification seront invoquées de façon similaire.

3.3.2.3 COM et l'héritage

C'est ici que le bât blesse. En effet, OLE propose un mécanisme (assez limité) d'héritage des interfaces *mais aucun mécanisme d'héritage des implémentations* c'est-à-dire du code des méthodes. Nous verrons dans les chapitre 3.5 comment la réutilisabilité du code peut néanmoins être assurée par les mécanismes de *délégation* et *agrégation*.

Finalement, quelle réponse fournir à notre grande question ? Si l'on considère uniquement les points de vue de l'encapsulation et du polymorphisme, OLE respecte les principes des systèmes Objet. En outre, des mécanismes de réutilisation sont proposés.

Toutefois, il aurait été souhaitable de supporter le mécanisme de l'héritage tant cette notion est fondamentale et naturelle. Son absence impose aux concepteurs d'architecture des mécanismes de modélisation moins naturels et conduit à des schémas objet peu orthodoxes. Cette lacune est partiellement comblée par les boites à outils proposées par d'autres éditeurs de logiciels.

3.4 Les interfaces d'objets OLE

A l'instar de CORBA, les objets OLE sont définis par leur interface et ne peuvent être accédés qu'au travers de celle-ci. En OLE, seules les méthodes sont proposées dans l'interface. Il n'y a pas de mécanisme semblable aux attributs interrogeables. La terminologie Microsoft parle de *contrat* établi entre le client et le serveur et définissant quelles opérations du serveur le client va pouvoir invoquer. Un même objet peut proposer plusieurs interfaces.

En particulier, le serveur et le client doivent se mettre d'accord sur les points suivants (termes du contrat) :

⇒ Identifier les interfaces

⇒ La description des méthodes d'une interface

⇒ L'implémentation de l'interface

Contrairement aux interfaces IDL de CORBA, il n'existe pas de véritable mécanisme d'héritage sur les interfaces des objets OLE. En revanche les mécanismes d'*Inclusion* et d'*Agrégation* seront fréquemment utilisés.

3.4.1 L'identification des interfaces

Une interface est définie par un *nom d'usage* (une chaîne de caractères commençant traditionnellement par la lettre I, comme dans *IUnknown*) et un identificateur numérique unique appelé *IID (Interface Identifier)*. L'IID est une forme particulière de GUID (Globally Unique Identifier). Ce dernier est garanti unique car il contient, en plus d'une information relative à l'interface, les précisions suivantes :

Une marque temporelle *(time stamp)* indiquant la date de création de l'interface. Ceci garantit que tous les GUID des interfaces créées sur une machine quelconque sont différents

Une marque de lieu *(localization stamp)* relative à la machine de création : le code MAC d'identification de la carte Ethernet de la machine hôte. Dans le cas où la machine ne possède pas de carte réseau, un identificateur est généré aléatoirement. De toute façon, cela ne pose pas de problème car cette machine est isolée du monde extérieur et les GUID auxquelles elle accède ne peuvent être que les siens propres, donc différenciés par la marque temporelle.

L'IID est une manière certes peu agréable d'emploi mais efficace de définir de manière unique une interface. Les IID sont ensuite enregistrés dans le *registre* qui est une forme très rudimentaire du Référentiel d'Interfaces de CORBA. En particulier, si le Référentiel d'Interfaces est local à un Bus, chaque machine possède son registre. Aussi, l'un des problèmes à résoudre dans un environnement réseau, est celui de l'harmonisation des registres entre les différentes machines concernées.

Si l'on prend l'exemple de Windows, le registre est une partie de la fameuse base de registres.

L'unicité et l'enregistrement des IID a une autre conséquence : une fois une interface définie et enregistrée, *elle ne peut en aucun cas être modifiée*. Il faut donc être particulièrement minutieux lors de la création d'une interface. Par exemple, l'interface standard de création d'instances a récemment due être modifiée pour incorporer des mécanismes de gestion de licences. Résultat de l'inaltérabilité des interfaces, il a fallu créer une nouvelle interface avec un nouveau nom et surtout un nouvel IID !

3.4.2 Description des interfaces

La description d'une interface OLE peut théoriquement se faire dans n'importe quel langage du moment que sa forme compilée remplit certaines conditions. En outre, elle doit contenir, au minimum les informations suivantes :

Le nom de l'interface sous forme d'une chaîne de caractères commençant traditionnellement par la lettre I (i majuscule !)

L'IID sous forme d'un GUID renommé ici *uuid*, terme emprunté à *DCE* protocole unifié par X/Open pour normaliser les RPC, mécanisme à la base de tout appel distant en OLE.

La définition des méthodes, avec leur nom et la liste des paramètres

L'inclusion d'autres interfaces que l'objet souhaite présenter, et en particulier, une interface de base nommée IUnknown

Des informations d'implémentation (en particulier, le nom de la DLL ou de l'exécutable qui contiendra les objets)

On voit que, contrairement à CORBA qui ne présentait dans l'interface que des informations de haut niveau, ici l'on doit spécifier l'IID (ce qui est très contraignant) et des informations d'implémentation. On perd ici la séparation interface/implémentation chère à CORBA et aux schémas orientés objet en général.

Alors que les interfaces peuvent être décrites, à priori, avec n'importe quel langage, Microsoft a développé son propre IDL (quelque fois nommé ODL pour Object Definition Language) permettant de réaliser cette opération de façon un peu plus sure. Toutefois, les opérations de projection d'ODL vers des langages de programmation cibles pour l'implémentation sont pour l'instant assez rares. En fait, seule sa projection vers le C++ fonctionne vraiment.

3.4.3 L'implémentation d'une interface

En C++, langage privilégié par Microsoft, l'implémentation d'une interface passe par la déclaration d'une classe dont les méthodes doivent coïncider avec celles décrites en interface.

Une fois compilée, la partie déclaration d'une interface ressemble à s'y méprendre aux tables de méthodes virtuelles du C++. Cette similitude n'est pas qu'apparente : les concepteurs de COM cherchaient un moyen d'implémenter un mécanisme d'invocation dynamique supportant le polymorphisme ; le C++ s'imposant comme le langage de programmation dominant, il était naturel d'adopter une architecture qui soit directement compatible avec les *virtual method tables* du C++.

En fait, la partie compilée d'une interface contient un pointeur sur un tableau (nommé *vtable*) recensant les pointeurs vers les méthodes invoquables. Il est clair que dans le cas du C++, le pointeur sur la *vtable* pointe en fait sur la *virtual method table* de la classe implémentant l'interface. De ce fait, les méthodes décrites dans une interface doivent obligatoirement être virtuelles.

La relation entre la classe d'un objet et les interfaces qu'il supporte est toutefois assez subtile. En effet, tous les objets d'une même classe présentent habituellement le même jeu d'interfaces alors que des objets présentant les mêmes interfaces ne sont pas nécessairement de la même classe.

En outre, COM n'interdit pas de rajouter des interfaces à un objet sans changer sa classe. Il suffit pour cela que les méthodes restent compatibles avec l'interface. Du coup, l'identification d'un objet passe l'identification de sa classe et celles des interfaces au travers desquelles on va pouvoir l'utiliser.

Alors que la résolution du problème de l'identification des classes aurait pu être résolu par un mécanisme semblable au RTTI (Run Time Type Information) préconisé par le C++, COM préfère réutiliser la technique déjà en vigueur pour l'identification des interfaces : l'utilisation d'un GUID qui porte ici le nom de CLSID (*Class Identifier*). Un CLSID identifie de manière unique le code d'implémentation associé à une classe. Par exemple, lors de la création d'un objet, on devra fournir le CLSID de la classe dont on crée une instance et l'IID de l'interface portée par cet objet et sur laquelle on recevra un pointeur.

L'invocation de méthodes

OLE ne supporte pas de mécanismes d'invocation statique. Tout repose sur des mécanismes d'invocation et de squelette dynamique pour reprendre la terminologie CORBA.

3.4.4 L'interface IUnknown

Tout objet COM doit présenter l'interface *IUnknown*. En effet, c'est à travers celle-ci que le client va pouvoir récupérer les pointeurs vers les autres interfaces et gérer le cycle de vie du client.

Cette interface décrit trois méthodes fondamentales :

AddRef et Release permettent respectivement d'incrémenter et de décrémenter le compteur de références d'un objet. Nous reviendrons sur ce mécanisme dans un prochain chapitre.

QueryInterface permet de récupérer un pointeur sur une interface quelconque d'un objet via le GUID de celle-ci.

Comme toute interface dérive de IUnknown, les méthodes précédemment décrites peuvent être invoquées à partir de n'importe quel pointeur d'interface. En fait, le scénario d'utilisation d'un objet OLE est le suivant :

Lorsque l'on demande à accéder à un objet, on reçoit un pointeur sur son interface IUnknown laquelle permet alors de récupérer les pointeurs vers les autres interfaces via la méthode QueryInterface.

3.5 Cycle de vie des objets OLE

3.5.1 Le comptage des références

Lorsqu'un objet n'est plus utilisé, il est important de libérer la mémoire qu'il occupe pour la rendre au système. Aussi, il est nécessaire de savoir à tout instant si un objet est utilisé ou non par des clients.

Le mécanisme choisi par OLE est celui du comptage des références à un objet *(Référence Counting)*.

Lorsqu'un client reçoit un pointeur vers un objet, c'est au travers d'une interface. Or toutes, les interfaces dérivent de IUnknown, laquelle présente la méthode *AddRef* dont le rôle est précisément d'incrémenter le compteur de références d'un objet. Aussi, la première chose à faire lorsque l'on reçoit un pointeur sur un objet est d'appeler explicitement *AddRef*.

De la même manière, lorsqu'un client n'a plus besoin d'un objet, il doit appeler la méthode *Release* qui elle, décrémente le compteur de références

Lorsque le compteur de références d'un objet chute à zéro (par suite d'appels à *Release*), celui-ci est détruit. S'il présente des interfaces de gestion de la persistance (voir plus bas), ces données persistantes sont préalablement sauvegardées.

3.5.2 Création d'une instance unique d'un objet

COM fournit une primitive de création d'un objet non initialisé nommée *CoCreateInstance*. Celle-ci prend plusieurs paramètres, dont :

⇒ Le CLSID de la classe de l'objet à construire

⇒ L'IID d'une interface supportée par l'objet à construire

⇒ Un pointeur sur l'interface *IUnknown* de l'objet englobant si l'objet en cours de création est agrégé.

⇒ Un paramètre indiquant si l'objet nouvellement créé l'est dans un contexte distribué sur plusieurs machines ou non.

Au cours de l'invocation de *CoCreateInstance* COM exécute les opérations suivantes :

Rechercher le serveur responsable du code exécutable de la classe demandée. Ceci requière que le système sache associer les CLSID avec des fichiers exécutables. Dans les systèmes Windows c'est encore la base de registres qui détient cette information. Lors de l'installation d'une application, celle-ci ajoute dans la base les CLSID de toutes les classes qu'elle administre.

Lancer le serveur. A ce moment là, le serveur crée l'instance demandée et renvoie un pointeur sur l'interface requise, à condition que celle-ci existe. Dans le cas contraire, une erreur est générée. L'objet créé est habituellement non initialisé. Aussi, l'interface demandée lors de l'appel de *CoCreateInstance* propose traditionnellement des méthodes permettant d'initialiser correctement l'objet demandé, par exemple, à travers un mécanisme de gestion de la persistance.

Renvoyer un pointeur sur l'interface demandée

Il est nécessaire d'insister sur le fait qu'il est important d'appeler la méthode *AddRef*, soit directement, soit au travers d'une méthode d'initialisation afin de ne pas risquer de voir un objet disparaître inopinément.

Il est intéressant de noter que l'utilisation d'un serveur local ou distant est transparent à l'utilisateur. D'un côté COM lancera l'exécutable et utilisera des facilités de communication inter-processus pour lancer les méthodes. Dans le cas distant, des appels RPC seront nécessaires.

3.5.3 Création de nombreuses instances

Le mécanisme précédent est bien adapté à la création d'instances uniques d'une classe. Dans le cas où l'on souhaite créer de nombreuses instances d'une même classe, il est préférable de sous-traiter l'opération à un objet spécial : une *fabrique de classe (Class Factory).* Les fabriques de classe sont spécifiques à une classe d'objet, aussi, l'utilisation d'une fabrique de classe dispense de connaître le CLSID de la classe de l'objet souhaité. En revanche, il sera toujours nécessaire de fournir l'IID de l'interface désirée.

En fait, tout objet est toujours créé par une fabrique de classe. La primitive CoCreateInstance ne fait qu'accéder à une fabrique de classe de façon transparente à l'utilisateur.

Les fabriques de classe sont des objets COM qui présentent l'interface *IClassFactory* laquelle contient les deux méthodes suivantes :

CreateInstance crée une nouvelle instance de la classe d'objets associée à la fabrique de classe. Dans un deuxième temps, elle fournit un pointeur sur l'interface désirée. Notons que cette méthode prend un paramètre qui doit être le pointeur sur l'interface *IUnknown* de l'objet englobant si l'objet en cours de création est agrégé.

LockServer permet de gérer la persistance en mémoire du serveur des objets. Semblable à *AddRef* et *Release* cette méthode permet, au moyen d'un paramètre, d'incrémenter ou décrémenter un sémaphore contrôlant la présence en mémoire du serveur.

La nouvelle interface *IClassFactory2* dérivant de *IClassFactory* permet, en plus des opérations précédentes, de gérer des licences d'utilisation, spécialement dans le cas de l'utilisation de DCOM.

L'accès à la fabrique de classe d'une classe particulière se fait au travers de la primitive *CoGetClassObject* qui permet, au travers du CLSID de la classe dont on désire fabriquer des instances et de l'identificateur de l'interface *IClassFactory* de récupérer un pointeur sur l'interface *IClassFactory* de la fabrique de classe nécessaire.

3.5.4 Initialisation des objets créés

Les objets créés par la fabrique de classe sont habituellement non initialisés. C'est pourquoi, l'interface demandée lors de la création propose habituellement des méthodes permettant d'initialiser les données membres de ces objets. Deux cas se présentent alors :

L'initialisation ne nécessite pas de données persistantes : elle est alors prise en compte par une méthode simple.

L'initialisation nécessite des données persistantes. La gestion de la persistance est assez complexe en OLE et peut reposer sur deux mécanismes différents :

⇒ Dans le premier cas, on fournit des méthodes aux objets qui décident d'eux mêmes à quel moment ils vont stocker sur support permanent leurs données membres.

⇒ Dans le second cas, les objets persistants présentent des interfaces spécifiques qui indiquent comment (et sur quel support) la persistance sera gérée. A partir de ce moment, les mécanismes de sauvegarde effectifs sont automatisés.

3.6 COM et la réutilisation du code

Plutôt que de s'appuyer sur des graphes d'héritage comme le fait CORBA, OLE préfère employer les mécanismes d'inclusion et d'agrégation.

3.6.1 COM bannit l'héritage des implémentations

Les concepteurs de COM encouragent la réutilisabilité maximale du code. Ce qui pourrait sembler paradoxal avec les limitations drastiques imposées au processus d'héritage. En effet, s'il existe une possibilité de dérivation des interfaces, l'utilisation de cette fonctionnalité n'est nullement encouragée. Le seul héritage d'interface utilisé dans les faits est celui sur *IUnknown*.

En outre, l'héritage d'implémentation, c'est à dire celui qui permet d'hériter directement du code d'implémentation des méthodes, est interdit. Ce qui veut dire qu'un objet COM ne pourra pas dériver d'un autre objet COM au niveau implémentation. Supposons que des objets COM soient implémentés sous la forme de classes C++. La classe C++ d'implémentation d'un objet COM pourra dériver de toutes les classes qu'elle souhaite, sauf celles servant déjà à implémenter un autre objet COM.

Microsoft prétend que cela aide à garantir la robustesse des classes existantes. A leur avis, pour bien travailler avec l'héritage, il est nécessaire de connaître, certes les spécifications des classes que l'on souhaite dériver mais également leur implémentation, notamment pour savoir quand déléguer certaines opérations à la classe mère. Hors, cet état de fait est difficile à obtenir lorsque différentes entreprises fournissent le code de la classe mère et des classes filles.

En outre, la modification de l'objet de base peut entraîner la modification de l'objet dérivé, ce qui peut amener à des incohérences à l'intérieur d'un même système.

Donc, COM ne permet pas l'héritage des implémentations s'appuyant sur le fait que quiconque doit pouvoir utiliser les interfaces des classes, que cela soit à l'intérieur d'une même organisation ou non. En revanche, ils prônent l'utilisation systématique des mécanismes de l'*Inclusion* (ou *Délégation*) et de l'*Agrégation* pour la réutilisation du code existant.

Ces deux mécanismes sont basés sur l'encapsulation d'un ou plusieurs objets *internes* par un objet *externe*, lequel est le seul visible par l'utilisateur. On dit que l'objet externe utilise les services de son ou ses objets internes.

A ce sujet il est important de noter la différence de vocabulaire entre COM et la conception Objet, portant en particulier sur l'utilisation des mots agrégation et délégation.

En conception orientée objet, agréger un objet signifie qu'une instance d'une classe apparaît dans la déclaration d'une autre. Donc, si l'on applique cette définition, les deux mécanismes cités ci-dessus sont des cas particuliers de COM. En outre, typiquement, le mécanisme classique de délégation utilise deux objets distincts non reliés par une relation d'appartenance.

3.6.2 Le mécanisme d'inclusion

Dans le cadre de l'inclusion (ou *délégation*) l'objet externe délègue l'exécution de certaines de ces méthodes à ces objets internes. Pour cela, il invoque certaines des méthodes de l'objet interne en réponse à l'appel des méthodes de son interface. L'objet externe peut donc être considéré comme un client de l'objet interne. De ce fait, l'implémentation de la délégation est extrêmement simple car elle ne diffère pas de l'utilisation quelconque d'un objet COM par un autre.

Il est important de noter qu'un objet ne peut différencier l'utilisation de ses interfaces dans le cadre de la délégation d'un appel par un autre objet.

3.6.3 Le mécanisme de l'agrégation

Si le mécanisme de délégation est très simple, il peut devenir très lourd, comme, par exemple, dans le cas de la délégation en cascade de plusieurs fonctionnalités.

L'agrégation permet à un objet de présenter les interfaces d'un ou de plusieurs de ces objets internes comme si c'était les siennes. Lors de l'appel à QueryInterface, l'objet externe va renvoyer un pointeur sur l'interface d'un objet interne comme si l'interface lui appartenait et le client n'en saura rien.

Ce mécanisme est certainement très puissant car il évite les appels d'interface en cascade. Il est toutefois contraignant car un objet utilisé en interne doit être spécialement conçu pour cet usage. En effet, certains problèmes apparaissent dès lors que l'on souhaite utiliser les méthodes définies dans *IUnknown*.

En effet supposons que le client désire accéder à une méthode de l'interface A qui est en fait, une interface de l'objet interne. A partir de cette interface, le client voudra sans doute utiliser *AddRef* puis obtenir une autre interface à l'aide de *QueryInterface*. Il est alors clair que ces méthodes doivent être obtenues à partir de *IUnknown* de l'objet externe ne serait ce que pour assurer que le comptage des références se fasse bien sur l'objet externe. En outre, le client ne connaissant pas l'existence de l'objet interne, il pourrait vouloir récupérer une interface fournie par l'objet externe à partir d'une interface de l'objet interne.

En fait, les objets internes délèguent leur interface IUnknown à l'objet externe. C'est un mécanisme d'inclusion inversée nécessitant pour l'objet interne d'obtenir un pointeur sur l'interface IUnknown de l'objet externe. Ceci est réalisé au moment de la création de l'instance de l'objet interne en spécifiant le pointeur sur IUnknown de l'objet externe à la primitive CoCreateInstance ou à la méthode CreateInstance de la ClassFactory.

3.7 OLE Automation

3.7.1 Généralités

OLE automation est le mécanisme qui permet à un client d'invoquer les méthodes d'un objet serveur. Ceci passe par un mécanisme d'invocation dynamique qui n'est pas sans rappeler les *Dynamic Invocation Interface* et *Dynamic Skeleton Interface* de CORBA.

En effet, OLE permet au client de découvrir, au cours de l'exécution du programme quels sont les services proposés par un objet et de construire dynamiquement une requête d'invocation. Réciproquement, OLE permet de construire une application serveur à même de déterminer dynamiquement quel objet activer et quelle méthode exécuter en réponse à un message émanant d'un client quelconque.

On retrouve ici le concept de Métadonnée (une donnée en décrivant une autre) cher aux concepteurs d'architectures orientées objet. En effet, les informations présentes dans les librairies de type ne sont autres que des formes particulières de méta classes.

Ces systèmes présentent des avantages certains liés à la robustesse et à la flexibilité qu'ils sous-entendent. Ainsi, il est possible de changer l'implémentation d'un serveur ou d'un client sans que l'autre partie n'ait à subir de modifications.

L'architecture d'OLE Automation repose sur les concepts d'*Objets Automates* et de *Contrôleurs d'Automates*. Les premiers enregistrent les services qu'ils fournissent et auxquels les seconds accèdent par voie de requête.

3.7.2 L'interface *IDispatch*

L'utilisation de l'interface standard *IDispatch* est le moyen par lequel les automates publient leurs fonctionnalités et par lequel leurs clients peuvent y accéder. Il peut d'ailleurs paraître étonnant d'utiliser la même interface des deux côtés de l'invocation.

L'utilisation de l'interface *IDispatch* permet de connaître le nom des méthodes que le client souhaite invoquer, ainsi que la liste des paramètres. Pour ce faire, les primitives de cette interface consultent une *Librairie de types (Type Libraries).*

Les méthodes de l'interface *IDispatch* permettent d'explorer la librairie de types (à travers une interface spécialisée nommée *ITypeInfo*) pour construire la liste des paramètres d'une méthode qui sera ensuite appelée par *Invoke* au travers d'un nouvel identificateur, le *DISPID (Dispatch Identifier*), GUID spécialisé dans les interfaces exportées.

3.8 Les composants ActiveX

3.8.1 Généralités

Les composants ActiveX représentent la forme la plus aboutie d'OLE. Ils prennent leur source dans la réalisation de documents composites, revenant ainsi aux sources d'OLE.

Le mécanisme d'OLE Automation repose sur un schéma de communication, certes bidirectionnel mais néanmoins très orienté dans le sens où les notions d'Automate et de Contrôleur sont figées.

Dans le cadre des composants ActiveX, ces différences s'estompent. En effet, considérons l'exemple d'un tableau Excel dans un document Word. Si vous faites un double clique sur le tableau afin de l'éditer, les menus d'Excel remplacent provisoirement ceux de Word. Ceci est du à un double mécanisme :

1) Des fonctionnalités Automate enregistrées dans le document et qui permettent à Word (le contrôleur) d'invoquer des commandes Excel (l'Automate). Les interfaces utilisées ici sont de type OLE Automation, c'est-à-dire, principalement, *IDispatch*.

2) Une communication inversée lorsque Excel à travers le composant demande à Word (à travers le conteneur) d'adopter sa barre de menus.

   Si *IDispatch* est une nouvelle fois mise à disposition (par exemple, pour l'envoi de messages permettant de modifier les menus), ce sont de nouvelles interfaces qui permettent, par exemple, de gérer les événements clavier et souris du composant.

Ici, ce ne sont plus directement les applications qui contrôlent l'interface utilisateur mais plutôt le contenu sémantique des documents pour le plus grand contentement de l'utilisateur. Ceci impose une révolution architecturale complète des logiciels.

Afin de fonctionner correctement, les composants ActiveX doivent être enregistrés dans le registre OLE. Cette opération est habituellement effectuée au moment du chargement du document composite que ce soit par fichier ou à travers un réseau.

Ceci pose le problème de la saturation du registre car, lorsqu'il libère la mémoire, le composant ActiveX ne se désenregistre pas !

3.8.2 Les conteneurs ActiveX

ActiveX fait une différentiation de traitement importante entre le *conteneur* (ou *contenant*) et le *composant*. Dans les applications les plus courantes, le conteneur est plutôt un fichier de type texte alors que les composants sont des tableaux ou des graphes etc ... bien que rien n'empêche d'inverser cet état de fait.

A la longue, les documents composites feront intervenir indifféremment des textes, des tableaux, des graphiques, des éléments multimédia ou des pages WEB.

L'uniformité de l'interface utilisateur (épaisseur des fenêtres, couleur du fonds, aspect des menus) est à la charge du document conteneur (et de son application associée). A cet effet, il définit des *propriétés ambiantes* et des *événements*.

Les propriétés ambiantes sont partagées par tous les composants ActiveX d'un même document. Afin de les rendre disponibles à tous ses composants, le conteneur utilise interface *IDispatch* transmise par l'interface spécialisée *IOLEClientSite*.

Les événements (par exemple, l'activation de raccourcis clavier) sont eux aussi générés par une interface *Idispath* et transmis via une interface spécialisée de la plus haute importance : *IOLEInPlaceActiveObject*.

3.8.3 Les composants ActiveX

Les composants ActiveX sont des composants visuels réalisant des tâches spécifiques pouvant être communes à de nombreuses applications. Par exemple, quelle que soit l'application office que vous utilisez, le contrôle visuel représentant un calendrier est le même, c'est un contrôle ActiveX présent dans une DLL. Avec l'avènement du Web, certaines pages interactives reposent essentiellement sur l'utilisation de contrôles ActiveX permettant, par exemple d'écouter du son. Chaque contrôle ActiveX est caractérisé par :

1) Les propriétés ambiantes

2) Les propriétés standard

3) Les propriétés étendues

4) Les propriétés spécifiques

5) Les événements reçus du contrôleur

6) Les événements émis vers le contrôleur

On retrouve ici le vocabulaire typique à l'automation OLE. En effet, les contrôles ActiveX dialoguent essentiellement via des mécanismes d'Automation avec l'application qui les a lancés.

Comme nous l'avons vu précédemment, les *propriétés ambiantes* sont partagées par l'ensemble des composants présents dans un document et servent avant tout à uniformiser l'aspect global du document. Lorsque l'une d'elles change, le composant en est averti par le conteneur via une méthode de l'interface spécialisée *IOLEControlSite*. Un bon exemple de propriété ambiante est la police de caractères utilisée dans les menus et qui doit être la même pour chacune des applications responsables d'un composant.

Les *propriétés standard* contrôlent l'apparence du composant en dehors du contexte global. Par exemple, on retrouve la police de caractères en cours, le style de dessins etc ...

Les listes de propriétés ambiantes et de propriétés standard sont directement établies par Microsoft.

Les *propriétés étendues* sont gérées directement par le contrôleur. Le meilleur exemple concerne le bouton associé à la touche return du clavier. Il doit être géré au niveau contrôleur car il peut mettre en jeu différents composants. Toutefois, chaque composant propose habituellement de lui même une valeur par défaut de chacune de ses propriétés spécifiques.

Les *propriétés spécifiques* ne concernent qu'un composant particulier et n'ont pas à être connues du conteneur ou d'un autre composant. Leur gestion est totalement à la charge du développeur du composant.

Les *événements* permettent au composant d'invoquer ou de faire remonter des informations vers son contrôleur. Leur gestion est assurée par deux interfaces de type *IDispatch* (donc associées, une fois de plus, à des mécanismes d'automation) au niveau du contrôleur et au niveau du composant. Du fait de Microsoft, et pour garantir que, virtuellement, tout événement peut être traité, ces derniers sont subdivisés en quatre grandes catégories pour lesquelles chaque contrôleur et chaque composant doit prévoir des fonctionnalités. A l'intérieur de chaque catégorie, une méthode du type I*QueryInterface* permet de découvrir exactement quels événements un composant est susceptible de proposer.

Les quatre grandes catégories sont les suivantes :

Les requêtes permettent à un composant de s'assurer de la validité d'une action (par exemple, fermer le composant) vis à vis du contrôleur.

Les événements préopératoires (*before events*) sont envoyés au contrôleur lorsque le composant désire commencer un traitement. Ils n'ont qu'un but de notification dans le sens où, contrairement à l'émission d'une requête, le contrôleur ne peut s'opposer au déroulement de l'action à commencer.

Les événements postopératoires (*after events*) notifient au contrôleur l'achèvement d'une opération particulière.

Les événements opératoires (*do events*) sont au cœur des mécanismes d'échange de données et de partage de tâches entre le composant et le contrôleur. Ils peuvent être implémentés soit au niveau du composant, soit au niveau du contrôleur ou bien par les deux. En effet, lorsqu'un composant envoie un événement au conteneur, celui-ci peut décider s'il y répond ou non.

Ces événements sont, en particulier, utilisés pour tout ce qui concerne la gestion de l'interface utilisateur et sont normalisés par Microsoft.

Soit, par exemple, l'exemple du contrôle ActiveX gérant un calendrier. Il est possible d'associer des actions à la sélection d'une date.

3.8.4 ActiveX et Internet

Le développement des composants ActiveX a été catalysé par Internet, friand de documents composites multimédia. En fait, le but ultime de Microsoft est de faire fusionner le futur standard de documents remplaçant HTLM avec son modèle de documents composites basé sur OLE/COM.

3.9 Le modèle distribué DCOM

Nous allons conclure cette rapide présentation d'OLE par une brève description du modèle COM distribué ou DCOM.

Tout d'abord, il faut remarquer que la distribution des composants n'est apparue que sur le tard dans OLE qui considérait initialement que toutes les opérations s'effectuaient sur une même machine.

La diffusion des messages en environnement distribué s'effectue par le biais de fonctionnalités RPC (*Remote Procedure Call*), aussi n'est-il-pas surprenant de voir que DCOM est basé sur DCE (*Distributed Computing Environment*), standard proposé par le groupe de travail X/Open et définissant quels types de données sont censés circuler sur un réseau et selon quels protocoles. DCE définit également des mécanismes de licences, de nommage et d'annuaires permettant d'accéder plus facilement et en toute sécurité à des services distants.

Il est à noter que dans le cas monomachine, COM utilise un protocole plus léger et dérivé de RPC nommé LRPC (*Lightweight Remote Procedure Call*).

En fait, DCOM propose une extension de DCE connue sous le nom d'ORPC (*Object Remote Procedure Call*) qui spécifie notamment les protocoles de communication et *le comptage des références distribuées*. OPRC s'appuie sur les mécanismes suivants :

Un format standard de représentation des données circulant sur le réseau : NDR (*Network Data Representation*) validé par DCE

L'utilisation des services de sécurité de DCE

Un mécanisme de gestion des versions nécessaire à la gestion de données circulant sur des machines équipées d'applications qui ont tout lieu d'être de versions différentes. Ce mécanisme est basé sur l'utilisation de GUID.

Sans vouloir entrer dans les mécanismes internes assez compliqués de DCOM, retenez qu'un message DCOM (acheminé par DCE) contient un identificateur de type GUID regroupant des informations aussi diverses que la machine destinataire, l'objet requis et l'interface demandée. Un processus démon semblable au BOA et connu sous le nom d'*Object Exporter* permet d'assurer l'envoi des messages ainsi que leur traitement sur la machine cible

4. Java Beans et RMI

Initialement, les communications en Java étaient limitées à l'utilisation de sockets. Avec l'arrivée du JDK 1.1, Java s'est vu muni d'un système de communication à base d'objets répartis : RMI (Remote Method Invocation). Celui-ci est encore limité dans le sens où le client et le serveur doivent tous deux avoir été écrits en Java et tourner sur une machine virtuelle.

Ce système n'est pas le seul à pouvoir être utilisé avec Java, car il existe une interface IDL pour Java qui permet d'utiliser des spécifications CORBA. Si CORBA est plus adapté aux environnements hétérogènes, RMI s'intègre directement dans Java et permet de bénéficier des mécanismes réseaux intrinsèques à ce langage.

Les buts de RMI sont les suivants :

⇒ Permettre d'invoquer plus facilement des méthodes Java entre applets ou applications sur des machines virtuelles différentes

⇒ S'intégrer le plus facilement et le plus naturellement dans Java de manière à promouvoir l'écriture d'applications réparties dans ce langage

⇒ Permettre à une applet d'appeler directement une application Java sur son serveur.

RMI est à la base de Java Beans, librairie de composants destinés à faciliter la construction et le déploiement d'applications distribuées écrites à l'aide d'objets distribués Java. En particulier, la publication des interfaces des composants disponibles se fait automatiquement.

Le développement d'objets compatibles RMI/Java Beans nécessite relativement peu d'efforts par rapport au développement d'objets non distribués. D'une part, les éditeurs de logiciels mettent à la disposition du programmeur tout un ensemble d'outils qui simplifient l'existence. D'autre part, peu de modifications dans la manière de programmer sont nécessaires. Notons toutefois, que le changement radical de gestion de l'interface utilisateur entre les JDK 1.0 et 1.1 était essentiellement motivée par le passage a RMI.

Le but ultime est de pouvoir insérer un composant Java Beans dans tout environnement d'objets distribués, c'est-à-dire essentiellement OLE/ActiveX et CORBA/OpenDoc. Dans ce sens, des efforts ont été faits afin de supporter les déclarations en IDL directement (attention, ce n'est pas la même chose que de profiter de la projection IDL en Java), de s'interfacer avec les composants ActiveX et les documents OpenDoc.