Conservatoire National des Arts et Métiers

Centre de Paris

Mémoire pour l'examen probatoire

Spécialité : Informatique Systèmes d'Information

par

Robin Maltête

ADA 95 et la répartition

Recommandation : traiter ADA 95 et les normes CORBA

Soutenu le mardi 22 mai 2007 à 16 heures

Jury

Président : Bernard Lecussan

Membres : Catherine Coquery, M. Gouet

Table des matières

Introduction

L'évolution des machines et surtout la généralisation des réseaux de communication a bouleversé l'informatique.

En remplaçant les applications monolithiques des grands serveurs par des applications réparties entre des clients et des serveurs, l'industrie informatique a créé un profond changement de paradigme.

Dans un premier temps, l'apparition des micro-ordinateurs et des réseaux locaux à faible coût (Ethernet) a modifié notre façon d'interagir avec nos applications.

La seconde révolution fut celle des réseaux mondiaux (WAN), qui ont repoussé les limites des réseaux locaux (notamment grâce à la bande passante disponible qui a évolué de façon vertigineuse).

La répartition est ainsi apparue : elle permet d'augmenter la puissance de calcul en répartissant la charge sur plusieurs machines, de mettre en commun quantité de ressources, de favoriser les échanges mondiaux (Internet), mais la complexité associée a elle aussi fortement évolué.

Les systèmes répartis permettent l'interaction de matériels, de systèmes d'exploitations et de logiciels hétérogènes. La qualité de la communication est importante, mais il est surtout nécessaire que les différents acteurs puissent communiquer en s'accordant sur un protocole commun.

Le but de cette étude est de comparer deux approches pour mettre en place un système réparti :

- **l'utilisation d'un langage unique** orienté répartition : ADA 95
- **l'utilisation d'un intergiciel** (« *middleware* » en anglais) permettant la communication entre des systèmes hétérogènes : CORBA

Nous décrirons dans un premier temps le fonctionnement des systèmes répartis et les différents modèles disponibles.

Par la suite, nous étudierons en détail un exemple de langage réparti (ADA 95), puis un intergiciel (CORBA).

Nous pourrons alors comparer les deux approches, ainsi que les passerelles disponibles entre celles-ci et en tirer les conclusions qui s'imposent.

A Les systèmes répartis

La répartition[1] est une mise à disposition, un partage de ressources et de services. Elle utilise notamment l'activation de traitements à distance. Les différents acteurs, des machines séparées physiquement, communiquent entre eux via un réseau de communication. Les systèmes peuvent être hétérogènes ou non, situés sur un réseau local ou un réseau mondial. L'ensemble peut être vu comme un seul système cohérent qui fournit divers services à ses membres.

A.1 Définition

Un système réparti ou distribué (« *distributed system* » en anglais) est [TAR 2004] :

- composé de plusieurs systèmes calculatoires autonomes
- sans mémoire physique commune (sinon c'est un système parallèle, cas dégénéré)
- qui communiquent par l'intermédiaire d'un réseau

Quelques exemples de systèmes répartis utilisés couramment : le web, les distributeurs automatiques des banques, les téléphones portables.

A.2 Comment communiquer ?

Quand on veut faire communiquer deux systèmes (ou ordinateurs), on utilise un (ou plusieurs) protocole de communication, c'est-à-dire un langage commun compris par tous. Il existe différents protocoles, selon la couche de communication qu'on utilise.

Certains protocoles sont dits « fiables » : TCP, ATM. La réémission et le ré-ordonnancement sont automatiques, mais la charge de travail des systèmes est plus grande.

D'autres sont « non fiables » : IP, UDP. La réémission et le ré-ordonnancement sont à la charge de l'utilisateur, ils sont plus efficaces sur un réseau fiable et disponible.

Le modèle OSI propose une représentation en sept couches [ORF 1999] :

Application	MOM, ORB, RPC	fournit à l'application les services réseau
Présentation	XDR	représentation des données
Session	Sockets, NetBios, TLI	étiquette sur le réseau : qui commence, qui se reconnecte en cas d'échec, etc.
Transport	TCP-IP, NetBEUI, SPX/IPX	fiabilité de la transmission
Réseau		routage des paquets
Liaison	Token-ring, Ethernet, RNIS, SDLC	transmission des paquets
Physique	Fibre optique, Coaxial, Paire torsadée	matériel

N.B. Nous nous intéresserons ici plus particulièrement aux couches « présentation » et surtout « application ».

[1] Au cours de cette étude, nous utiliserons indifféremment les termes « distribution » ou « répartition » pour désigner les systèmes répartis.

A.3 Avantages des systèmes répartis

- **Ubiquité** : Utilisables par plusieurs acteurs depuis des endroits différents.
- **Disponibilité** : La redondance disponible dans les systèmes partagés permet de pallier un défaut d'un des acteurs
- **Efficacité** : Il est possible de choisir parmi les différents systèmes le service disposant du temps de réponse le plus court.
- **Performance** : les calculs parallèles permettent d'améliorer sensiblement les temps de traitements.
- **Confidentialité** : Les données étant segmentées, chaque unité n'en possède qu'une partie, on peut ainsi assurer une plus grande sécurité.

A.4 Inconvénients

- **Complexité** : La nécessaire harmonisation de systèmes hétérogènes entraîne un empilage de couches de communication qui complique fortement la programmation (il est nécessaire de s'approprier différentes technologies).
- **Dépendance forte au réseau** : l'utilisation d'un réseau de communication entraîne la nécessité de traitements supplémentaires (correction d'erreurs, validation des messages, etc.). De plus, le réseau doit bien sûr être fiable.

La complexité d'un système réparti est beaucoup plus importante que celle d'un système monolithique. En effet, beaucoup de problèmes théoriques se posent : gestion de l'ordonnancement, votes ou consensus (élection d'un leader), détection de fautes, partitionnement des données, gestion de la réconciliation.

Auparavant, dans les applications distribuées, les développeurs géraient eux-mêmes les communications entre systèmes, décidaient de leurs propres protocoles, des types de données utilisées, des représentations de celles-ci. Cette façon de travailler avait l'avantage de la simplicité, mais la maintenance de ces applications était souvent acrobatique, et il était difficile d'utiliser des systèmes hétérogènes (personne n'est capable de connaître tous les systèmes et tous les langages de programmation).

Pour résoudre ces problèmes, des **modèles de répartition** furent progressivement mis en place.

A.5 Trois classes de systèmes répartis

Au cours du temps, et en association d'idées avec les méthodes de programmation courantes, trois modèles sont apparus :

A.5.1 Les envois de message

Ce type de communication, assez primitif, est toutefois encore utilisé (notamment dans des systèmes homogènes, où l'on peut se passer des conventions de format de données).

Des messages indépendants sont envoyés sur un réseau (fiable ou non fiable). Il s'agit en général de flux d'octets non structurés. L'échange est de type client-serveur : l'appelant envoie une requête et ses arguments, il attend les résultats. L'appelé attend les requêtes et leurs arguments, calcule le résultat et le renvoie. Pour distinguer les différents types d'applications sur une machine, on utilise des ports spécialisés (par exemple 21 pour le FTP).

Exemple : IRC, le protocole de « clavardage » ou « *chat* » en anglais.

Pour pouvoir utiliser ce type de communication il est nécessaire au préalable de disposer des services suivants :

- **Localisation des services**. Il est nécessaire de se mettre d'accord sur les ports utilisés. Un certain nombre de ports sont normalisés (« *well-known ports* »). Sur les machines Unix, le fichier **/etc/services** répond à ce besoin. Cela peut poser des problèmes dans les entreprises qui utilisent des pare-feux à blocage de ports fixes : si le port de communication n'est pas paramétrable dans l'application distribuée, la communication est impossible.
- **Conventions de format de données.** Dans un système hétérogène, il faut définir un format d'échange commun pour les types de base (little-endian, big-endian; 32, 64 ou 128 bits, représentation des nombres en « virgule flottante »). Pour les types complexes, il faut également utiliser des dictionnaires pour déterminer l'ordre des composants dans une structure. Il existe des formats d'échange normalisés : XDR (Sun), CDR (CORBA).

Ces conventions doivent être connues **a priori** par les différents systèmes. Certaines données ont une sémantique locale (pointeurs, tâches, fichiers actifs), elles sont difficilement transportables. On peut contourner ces limitations en « aplatissant » les listes ou les tableaux (sérialisation), et en encapsulant les données d'une tâche.

A.5.2 Les appels de procédures à distance

Également appelés RPC (*Remote Procedure Call*), les appels de procédure distants sont encore fréquemment utilisés.

Il s'agit ici de remplacer les appels réseau par des appels de sous-programmes. Le dialogue, de type question-réponse, est géré par le système, les paramètres sont empilés sur le canal de communication plutôt qu'en mémoire. **La communication est synchrone** : la procédure appelante attend les résultats demandés avant de poursuivre son exécution.

En environnement concurrent, un client peut faire plusieurs requêtes à un serveur. Pour cela, on dispose de plusieurs méthodes :

- Utiliser un canal de communication différent pour chaque tâche cliente
- Différencier les requêtes avec des numéros de séquence

Du côté serveur, on dispose aussi de plusieurs stratégies pour traiter les requêtes entrantes :

- Lancer un nouveau processus enfant (*fork*) pour chaque nouvelle connexion (exemple : le serveur web apache)
- Lancer une nouvelle tâche pour chaque requête reçue (exemple : NFS)
- Utiliser la sérialisation (exécuter les requêtes les unes après les autres).

La tâche du programmeur est facilitée :

- Il n'est plus nécessaire de coder les boucles d'attente
- La génération automatique de code est possible
- Il n'est plus nécessaire de s'occuper du format de données.

Stubs et skeletons

Le système RPC s'efforce de maintenir le plus possible la sémantique habituelle des appels de fonction : tout doit être le plus transparent possible pour le programmeur.

Pour que cela ressemble à un appel de fonction local, il existe dans le programme client une fonction locale qui a le même nom que la fonction distante et qui, en réalité, appelle d'autres fonctions de la bibliothèque RPC qui prennent en charge les connexions réseaux, le passage des paramètres et le retour des résultats.

De même, côté serveur il suffira (à quelques exceptions près) d'écrire une fonction comme on en écrit tous les jours, un processus se chargeant d'attendre les connexions clientes et d'appeler votre fonction avec les bons paramètres. Il se chargera ensuite de renvoyer les résultats.

Les fonctions qui prennent en charge les connexions réseaux sont des « stub » et des « skeleton ». **Il faut** donc **écrire un stub client et un skeleton serveur en plus du programme client et de la fonction distante**.

Le stub (*souche* en français) intercepte les appels lancés par le client à l'interface de référence et redirige ceux-ci vers le service distant, c'est un **relais côté client**. Il emballe les arguments (« marshaling »), les envoie dans un flux de données vers le skeleton distant, et déballe (« unmarshalling ») le résultat reçu avant de le retourner à la méthode appelante.

Le skeleton déballe les paramètres reçus, les transmet à la méthode locale, puis remballe le résultat pour les communiquer au client. C'est un **relais côté serveur**

Cette structure évite au programmeur d'avoir à communiquer explicitement avec le service distant.

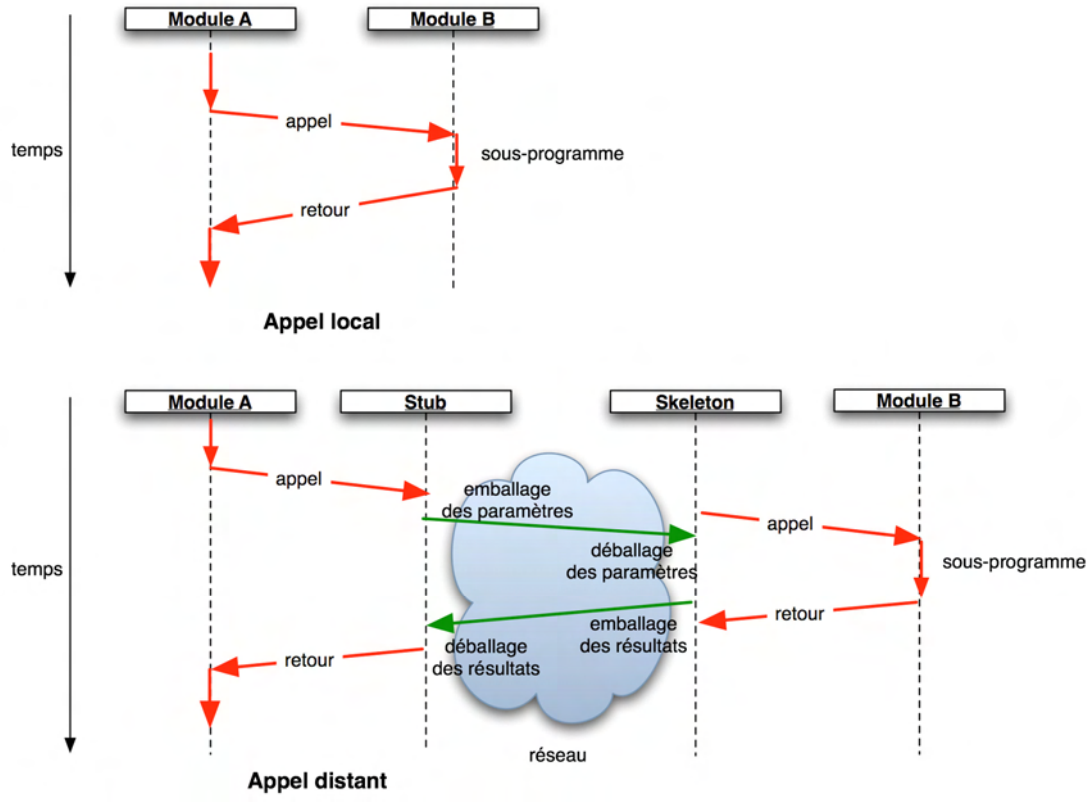

Figure 1 : Mécanismes RPC

RPC évolués

Dans certains RPC évolués, il est possible de **transmettre les exceptions** de l'appelé vers l'appelant. Certains RPC évoluent vers des **procédures asynchrones** : il n'est plus nécessaire d'attendre le résultat d'un sous-programme. Le **contrôle de versions** est aussi parfois disponible : on peut vérifier la cohérence entre le stub et le skeleton.

Services associés

- Service de nommage qui enregistre les coordonnées des serveurs distants
- Service de sécurité : il permet de s'assurer de l'identité des différents acteurs et fournit des jetons d'authentification et de chiffrement.

Exemples de RPC

- Sun propose un RPC qui utilise **XDR** comme format de données, fournit le programme **rpcgen** pour construire les stubs et les skeletons à partir d'une description d'interface. Pour cela il est nécessaire d'utiliser le service NFS, qui permet d'utiliser des répertoires distants en mode sans état et fonctionne avec UDP et TCP. Le service de nommage est plus souple que `/etc/services` : tout nouveau service s'enregistre auprès du portmapper, toute requête à un service interroge d'abord le portmapper afin de connaître le numéro de port utilisé. Contrainte importante : le portmapper utilise un port fixe.
- **XML-RPC/ SOAP / REST** : Ces trois systèmes permettent l'échange de messages unidirectionnels sans états, ils utilisent des technologies normalisées : XML pour les données, HTTP pour le transport. SOAP a été proposé par Microsoft. REST tente une approche plus facile d'accès. On les appelle aussi « web services », et ils ont beaucoup de succès principalement à cause de leur facilité de mise en oeuvre.

Les RPC sont **plus fiables que les envois de messages,** ils sont adaptés aux langages procéduraux, mais, comme les communications par messages, ils **ne peuvent s'affranchir d'un système de localisation de services**. Néanmoins, dans un environnement homogène, l'envoi de messages reste parfois adapté car il nécessite moins de ressources et reste plus simple à mettre en place.

A.5.3 Les objets répartis

La programmation objet est plus riche que la programmation structurée.

L'encapsulation, notamment, permet une plus grande souplesse : **les données mais aussi les méthodes sont encapsulées à l'intérieur d'une classe**. Ceci permet de s'affranchir des problèmes de nommage, et améliore l'évolutivité (on peut modifier une méthode ou enrichir une classe sans remettre en cause la communication extérieure avec celle-ci). Le polymorphisme et l'héritage facilitent également la maintenance et la réutilisation du code.

Les objets répartis enrichissent les RPC : la conception est orientée acteurs plutôt que fonctionnalités, et il est possible d'étendre un service progressivement.

Il n'existe aucune différence de concept entre les modes monolithique (non réparti) et réparti.

Les objets répartis communiquent par l'intermédiaire de leurs méthodes publiques, ils ne se déplacent pas (seules les références sont changées), et ils ne sont utilisés qu'à travers leurs pointeurs.

Il est à noter que **les objets peuvent agir alternativement comme client ou comme serveur** en fonction des besoins : les rôles client et serveur ne sont utilisés que pour coordonner les interfaces.

On peut enrichir encore les objets répartis en :

- **Partageant la mémoire** (DSM : *Distributed Shared Memory*) :
  Les zones de mémoire sont dupliquées sur chaque noeud du système, chaque noeud possède une copie des objets manipulés.
- **Utilisant des MOM** (*Message Oriented Middleware*) :
  On utilise ici des files d'attente de messages. Cela permet au système de communiquer de manière asynchrone : l'appelant et l'appelé peuvent travailler à des rythmes différents. Un consortium existe depuis 1993 qui gère la normalisation des middleware orientés message.

La communication entre objets répartis est souvent prise en charge par un logiciel spécialisé nommé intergiciel (middleware en anglais). Le intergiciels proposent aussi certaines fonctionnalités complémentaires :

- **Transactions**
  On pourra grouper au sein d'une transaction un ensemble d'échanges (messages, écritures sur un espace partagé, appels de méthodes, etc.) pour augmenter la cohérence du système. Le concept de transaction permet de contrôler l'application de cet ensemble et de tous ses composants de façon globale : si une seule unité échoue, la transaction n'est pas validée et elle est annulée. Les transactions possèdent généralement les propriétés ACID[2]
- **Avortements d'appel**
  Dans le mode d'interaction de type requête-réponse (et pas dans le mode « passage de messages »), on peut souhaiter annuler un appel de méthode distante avant la fin de son exécution normale. Ceci est particulièrement utile dans les systèmes « temps réel ».
- **Gestion des exceptions**
  Si une exception provient pendant l'appel d'une méthode distante, l'exception est signalée à l'appelant. Si le langage de l'appelant gère les exceptions, on pourra lui transmettre celle-ci de façon native.

[2] ACID est un acronyme (ISO/IEC 10026-1:1992 Section 4) référant aux propriétés suivantes :
- **Atomicité** : une transaction doit soit être complètement validée ou complètement annulée.
- **Cohérence** : aucune transaction ne peut sortir du système dans un état incohérent.
- **Isolation** : une transaction ne peut voir aucune autre transaction en cours d'exécution.
- **Durabilité** : aucune transaction validée ne sera perdue : le système est capable de résister aux pannes.

A.6 Analogies de concepts

Les différents modèles de répartitions ont évolué historiquement parallèlement aux concepts de programmation :

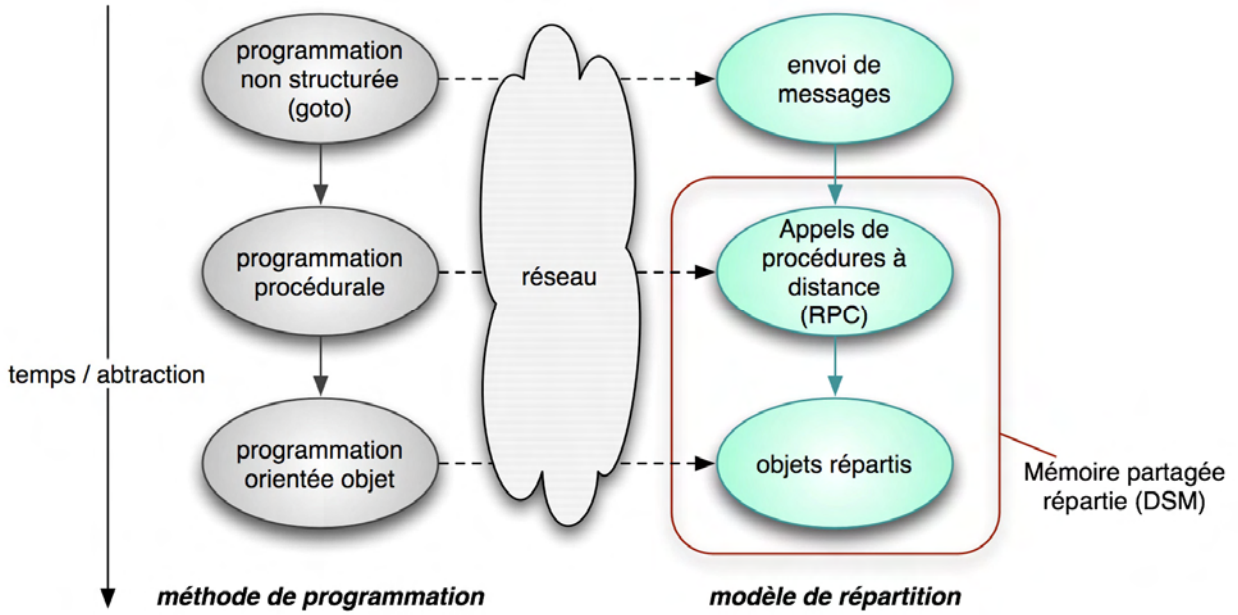

Figure 2 : Analogies entre méthodes de programmation et modèles de représentation

Exemples de systèmes objets répartis

- **ADA 95** et son annexe (lire la suite de ce document)
- **CORBA** (lire la suite de ce document)
- **DCOM** : Langage de définition d'interfaces proposé par Microsoft, concurrent direct de CORBA (objets répartis).
- **RMI** : Modèle de répartition Java, basé sur les objets, plus simple à mettre en place que CORBA.

Selon nous, les concepts essentiels qui permettent de caractériser un environnement de gestion d'objets répartis sont :

- La **localisation des services**
- L'**interopérabilité**
- Le **niveau de sécurité**

Note : **Nous ne traiterons dans cette étude qu'ADA 95 et CORBA.**

B La répartition en Ada 95

B.1 Caractéristiques du langage ADA

Le langage Ada a vu le jour au sein du ministère de la défense des États-Unis : en 1975, ses services utilisaient plus de 450 langages de programmation différents ! Un appel d'offres a donc été lancé, dans le but de concevoir un langage unique et généraliste, adapté aussi bien aux besoins de gestion qu'aux applications scientifiques, ou encore aux programmes « temps réel » et embarqués. La norme définitive a été adoptée en 1983[3]. Elle est devenue une norme internationale grâce à son adoption par l'ISO1 en 1987[4].

La conception d'Ada est guidée par un souci d'**intégrer dans le langage des pratiques de programmation respectueuses des principes de génie logiciel**, et de **faciliter le développement de larges projets**, qui font intervenir de nombreuses équipes indépendantes, et qui doivent être maintenus longtemps après leur livraison.

En 1995, une nouvelle norme ISO correspondant au langage Ada a été publiée[5], elle introduit un grand nombre de nouveaux concepts, comme les **fonctionnalités orientées objets**, un **modèle pour le développement d'applications réparties** et de nouveaux **mécanismes pour le « temps réel »**, tout en préservant une compatibilité ascendante avec la première version.

L'une des annexes optionnelles fournie avec ADA 95 concerne la répartition : il s'agit de l'**annexe E** appelée aussi **DSA** (*Distributed System Annex*) qui nous intéresse ici tout particulièrement.

Les caractéristiques principales du langage Ada 95 sont :

- **Typage fort** : Il permet de détecter un maximum d'erreurs lors de la compilation. Par exemple, plusieurs types de même représentation interne peuvent être incompatibles (par exemple un type entier et un autre allant de 1 à 5).
- **Abstraction et encapsulation** : un programme Ada est organisé en paquetages et en sous-programmes. Chaque paquetage contient une partie visible représentant l'interface à utiliser pour accéder à ses services, une partie privée contenant les détails des types utilisés dans la partie publique et un corps abritant la mise en œuvre des services du paquetage.
- **Modularité** : La séparation du programme en différentes entités (ou modules) permet la compilation séparée, la réutilisation, facilite la mise au point et la portabilité.
- **Synchronisation** : des interactions entre les tâches peuvent être créées par le biais de rendez-vous. Ces rendez-vous permettent l'échange d'information entre tâches de manière synchronisée.
- **Facilité de lecture** : contrairement à d'autres langages de programmation, l'utilisation d'Ada facilite la relecture d'un code existant (ou sa reprise ultérieure) plus que l'écriture du code initial. Ada étant précis et « verbeux », le code source est très « lisible ».

---

3 ANSI/MIL-STD A8A5 A, 983
4 ISO 8652 : 1987
5 ISO/IEC/ANSI 8652 : 1995

- **Bibliothèques prédéfinies** : des bibliothèques normalisées dans le manuel de référence permettent de disposer de certaines fonctionnalités courantes.
- **Contrôle de l'élaboration** : les différents paquetages formant une application Ada s'élaborent selon des règles précises ; leurs déclarations sont évaluées avant de lancer le programme principal afin notamment de positionner les variables à leur valeur initiale.
- **Validation** : tout programme prétendant être un compilateur Ada doit, au préalable, être validé par un institut indépendant. Cette validation est organisée autour d'une vaste suite de tests vérifiant le respect de la norme.
- **Programmation orientée objets** : Ada supporte l'héritage simple, le polymorphisme et l'aiguillage dynamique (ou liaison dynamique) d'appels de méthodes. Ada 95 est le premier langage objet normalisé par l'ISO.
- **Unités de bibliothèque hiérarchiques** : Une unité de bibliothèque est un paquetage ou un sous-programme qui n'est pas imbriqué dans une partie déclarative. Il s'agit d'une unité d'abstraction du niveau le plus haut. Il est possible d'étendre un paquetage par la création d'un paquetage enfant, qui obtient la visibilité sur la partie privée de son père.
- **Annexes optionnelles** : ces annexes, qui ne sont pas obligatoirement mises en œuvre dans tout compilateur Ada validé, doivent, si elles sont fournies, respecter la norme. Des annexes existent entre autres pour le temps-réel et la répartition.

Ces caractéristiques font d'Ada un **langage puissant adapté à un grand nombre de situations**. Il satisfait aux exigences du cahier des charges initial, qui imposent la possibilité d'utiliser le langage dans tous les domaines de l'industrie. En plus de ses usages les plus courants (applications critiques, temps-réel, aéronautique, transports – la ligne 14 du métro parisien a été conçue avec ADA-), le langage ADA est utilisé dans l'enseignement de la programmation, dans le but de transmettre une certaine rigueur et de bonne pratiques.

B.2 L'annexe DSA : la répartition en ADA 95

La programmation distribuée revient à découper un programme en plusieurs exécutables (appelés partitions en ADA) devant s'exécuter sur des machines (voire des systèmes d'exploitations) différentes.

L'annexe optionnelle des systèmes répartis définit un moyen de **développer des applications réparties sans sortir du cadre du langage** : **toutes les règles concernant le typage fort et les exceptions restent applicables dans l'intégralité du programme réparti**, sans considération de localisation des différents nœuds logiques.

B.2.1 Philosophie de l'annexe DSA

L'**annexe des systèmes répartis d'Ada a pour but de supprimer la barrière existant entre les programmes centralisés et les programmes répartis.** Il est donc très facile de passer de la version monolithique d'une application à sa version répartie : **Le code source d'un programme ADA peut être utilisé indifféremment** (et sans modifications) pour la **construction d'une application monolithique ou répartie**.

En effet, la seule différence se situera au moment de la compilation : Grâce aux **directives de compilation** (ou « *pragma* » en anglais) qui sont des indications données au compilateur, on pourra attribuer des propriétés supplémentaires à certains modules à des fins de répartition.

Le compilateur vérifiera (entre autres) que les types utilisés entre les différents nœuds logiques sont transportables sans perte de signification. Une valeur entière, par exemple devra garder la même signification sémantique sur toutes les partitions.

L'annexe de systèmes répartis introduit la notion de partition comme étant un nœud logique, agrégat d'unités de bibliothèque. Elle distingue deux types de partitions :

- **Les partitions actives**, qui émettent et reçoivent des requêtes.
- **Les partitions passives**, qui servent d'espace de stockage utilisé par une ou plusieurs tâches situées dans des partitions actives.

Tous les appels entre partitions actives se font, en Ada, par le biais d'appels de sous-programmes ou de méthodes d'objets répartis. Le langage n'offre pas la possibilité d'utiliser, pour la synchronisation, des rendez-vous répartis dont l'un des participants se trouve sur une première partition et l'autre sur une deuxième. Cette possibilité a été écartée pour des raisons de compatibilité ascendante.

B.2.2 Catégorisation des paquetages

L'unité de base de répartition en Ada étant l'unité de bibliothèque, il n'est pas possible d'avoir un paquetage présent en partie sur une partition et en partie sur une autre. Les paquetages, dans le langage Ada, sont catégorisés à l'aide de directives de compilation, dont la hiérarchie respecte la règle suivante :

`Remote_Call_Interface > Remote_Types > Shared_Passive > Pure`

C1 > C2 signifie qu'un paquetage de catégorie C1 peut avoir une visibilité sur un paquetage de catégorie C2.

Un paquetage catégorisé **`Shared_Passive`** ne pourra donc avoir de visibilité que sur d'autres paquetages de même catégorisation ou de catégorisation plus restrictive (**`Pure`**). En ce qui concerne les catégorisations **`Remote_Call_Interface`** et **`Remote_Types`**, les limitations ne s'appliquent qu'à la déclaration de ces paquetages ; aucune limite de dépendance sémantique ne s'applique à leur corps.

Les différentes catégories de paquetages signifient respectivement :

- **`Remote_Call_Interface`** : le paquetage contient dans sa partie visible la déclaration des sous-programmes pouvant être appelés à distance (ou « Remote Procedure Call » en anglais) et de types transportables. Un tel paquetage définit l'interface d'un service localisé sur un nœud particulier d'une application répartie : **il ne peut être dupliqué**.
- **`Remote_Types`** : le paquetage ne contient, dans la partie visible de sa déclaration, que des types pouvant être transportés entre les différentes partitions. On pourra notamment définir des références sur sous-programmes ou sur objets distants. Les types transportables peuvent être utilisés par plusieurs partitions, et être transmis entre partitions en conservant leur sémantique. Un paquetage de ce type sera **dupliqué sur toutes les partitions qui le référencent**.

- **`Shared_Passive`** : le paquetage ne contient aucun flot de contrôle par lui-même. Les variables définies dans la partie visible d'un tel paquetage seront accessibles depuis plusieurs partitions du système réparti à travers un support partagé éventuellement réparti comme une mémoire, un système de fichiers ou une base de données. De plus, dans ces deux derniers cas, la persistance est obtenue automatiquement. Un tel paquetage définit un corpus de données passif partagé par l'ensemble des partitions qui prennent part à une application répartie : **il ne peut être dupliqué**.
- **`Pure`** : le paquetage est garanti sans effet de bord et ne conserve aucun état interne. Il contient typiquement des définitions de types simples et les opérations primitives applicables sur ces types. Un paquetage de ce type sera **dupliqué sur toutes les partitions qui le référencent**..

Toute unité de bibliothèque sans catégorisation est qualifiée de normale et sera dupliquée sur toutes les partitions sur lesquelles elle est référencée.

La directive de compilation **`Asynchronous`** peut également être appliquée à un sous-programme sans paramètre de sortie d'une unité de bibliothèque ayant la catégorie **`Remote_Call_Interface`**, ce qui a pour effet de rendre tout appel à ce sous-programme **unidirectionnel**. Toute levée d'exception est alors ignorée.

La directive de compilation **`All_Calls_Remote`** peut être appliquée à une unité de bibliothèque catégorisée **`Remote_Call_Interface`** de sorte que tout appel à un sous-programme distant de cette unité devra transiter par le sous-système de communication même si l'appel peut être résolu en local. Cette fonctionnalité peut se révéler fort utile lors de la mise au point de l'application alors qu'elle pas encore été répartie puisque les latences induites par la communication sont introduites.

B.2.3 Références sur entités distantes

Il existe deux types d'entités sur lesquels il est possible d'obtenir une référence ou pointeur distant en Ada : les types étiquetés limités privés, utilisés pour la programmation orientée objets, et les sous-programmes. Le premier cas est proche des objets répartis trouvés dans CORBA et RMI, alors que le second est plus spécifique à Ada.

B.2.3.1 Références sur objets distants

Un type pointeur défini dans la partie visible d'un paquetage portant une des catégorisations **`Remote_Call_Interface`** ou **`Remote_Types`** est un pointeur sur objets distants si les deux conditions suivantes sont vérifiées :

- il désigne une hiérarchie de types dont la racine est un type étiqueté limité privé (c'est-à-dire un type possédant les trois caractéristiques énoncées ci-dessus) ;
- c'est un pointeur général : les pointeurs généraux en Ada peuvent désigner aussi bien des objets locaux ou globaux que des objets alloués dynamiquement.

Une telle référence peut pointer aussi bien sur un objet local que sur un objet distant.

B.2.3.2 Références sur sous-programmes distants

Des pointeurs sur sous-programmes peuvent être déclarés dans les paquetages **`Remote_Call_Interface`** ou **`Remote_Types`**. Ces pointeurs deviennent automatiquement des pointeurs sur sous-programmes distants.

Comme les pointeurs sur objets répartis, les pointeurs sur sous-programmes distants connaissent quelques limitations :

- ils ne peuvent pointer que sur des sous-programmes appelables à distance, c'est-à-dire définis dans un paquetage catégorisé à l'aide de la directive **`Remote_Call_Interface`**;
- ils ne peuvent être convertis qu'en un autre type pointeur sur sous-programme distant.

B.2.3.3 Traitement des exceptions

Mis à part les changements introduits par l'utilisation de la directive de compilation **`Asynchronous`**, le traitement des exceptions dans une application répartie écrite en Ada 95 ne diffère aucunement de celui effectué dans une application monolithique.

Dans un programme monolithique, une exception inconnue pour des raisons de visibilité peut être rattrapée par un traite-exceptions grâce à une alternative **`when others`**.

Dans un programme réparti, il en est de même : une exception *E* levée sur une partition *P1* peut être inconnue sur une partition *P2* mais rattrapée à l'aide de cette même alternative.

Cependant, si cette exception est implicitement ou explicitement renvoyée sur *P1* ou sur toute autre partition sur laquelle *E* est connue, alors elle pourra être rattrapée par un traite-exception spécifique de type **`when`** *E* : le type de l'exception n'est pas perdu en cours de route, même si celui-ci est inconnu sur certaines partitions intermédiaires.

Enfin, le processus consistant à rassembler les différents paquetages en partitions est un processus **post-compilatoire** non normalisé. La norme laisse également une certaine place pour d'éventuelles extensions, l'équipe de normalisation étant consciente de ne pas explorer toutes les possibilités en la matière.

B.2.4 Exemple : « Hello World » en ADA 95 distribué :

Le serveur :

```
package Hello_Server is
   pragma Remote_Call_Interface;
   function Message return String;
end Hello_Server;

package body Hello_Server is
   function Message return String is
   begin
        return « Hello World! »;
   end Message;
end Hello_Server;
```

Le client :

```
with Ada.text_Io;
with Hello_server;

procedure Hello_client is
use Ada;
begin
   Text_Io.Put-Line (Hello_Server.Message);
end hello_Client;
```

La seule addition par rapport à la version monolithique est la directive de compilation « **Remote_Call_Interface** » (désignée ici en rouge) : **on peut difficilement faire plus simple** !

B.2.5 Résumé ADA

Nous l'avons vu, **ADA 95 possède de nombreux atouts**…

- ADA 95 permet de construire des systèmes répartis sans sortir du langage
- ADA 95 remplace certaines solutions propriétaires
- La migration d'une application monolithique en application distribuée est facile
- ADA 95 permet d'utiliser aussi bien les objets répartis que les RPC
- Il est possible de partager des objets (mémoire partagée virtuelle)

...mais **certaines fonctionalités manquent cruellement** :

- ADA 95 ne permet pas de coopérer avec d'autres systèmes

C La répartition avec CORBA

CORBA (*Common Object Request Broker Architecture*) est le projet **d'intergiciel** (« *middleware* » en anglais) le plus important et le plus ambitieux jamais entrepris par l'industrie informatique.

C'est le produit d'un consortium créé en 1989, l'OMG (*Object Management Group*) qui comprend plus de 800 entreprises informatiques (BEA, Oracle, Sun, IBM, etc.), à l'exception notable de Microsoft qui a développé un concurrent direct : DCOM (*Distributed Component Object Model*).

Qu'est-ce qu'un intergiciel ?

> Un intergiciel est une bibliothèque qui offre aux développeurs d'applications un **moyen de faire interagir des composants distants** sans avoir à prendre en charge certains aspects délicats de la répartition. En particulier, un intergiciel permet la communication entre composants sans que les développeurs aient à utiliser eux-mêmes les fonctionnalités de communication de l'environnement (par exemple les interfaces offertes par les réseaux).
>
> Un intergiciel peut prendre en charge d'autres aspects problématiques de la communication entre calculateurs, tels que **l'hétérogénéité.** À partir d'une description de haut niveau des données que doivent échanger deux composants, un intergiciel peut proposer une **représentation commune de ces données,** destinée à être communiquée entre calculateurs, et fournir les fonctions de conversion permettant de passer des représentations natives utilisées par les composants de l'application à la représentation commune utilisée pour les échanges.
>
> Enfin, un intergiciel peut offrir de nombreux services de haut niveau permettant de répondre à certains problèmes introduits spécifiquement par la répartition, tels que des **services d'annuaires de composants**.
>
> Les intergiciels offrent ces fonctionnalités aux applications par l'intermédiaire d'**interfaces de programmation**, qui peuvent être définies de diverses façons.
>
> De même, diverses conventions (ou protocoles) peuvent être utilisés par les intergiciels lorsqu'ils communiquent à travers un réseau pour s'informer mutuellement des interactions demandées par les composants applicatifs.

Les premières normes CORBA ont vu le jour en 1992 (version 1.1). En 1996, la version 2.0 apporte l'IDL et le protocole IIOP. En 1999 la version 2.3 rend interopérable CORBA et RMI. En 2002, la version 3 spécifie 16 nouveaux types de services mais tous ne sont pas mis en oeuvre dans les ORB du marché.

CORBA a la capacité d'**intégrer toutes les autres formes d'intergiciels client/serveur.**

CORBA peut être vu comme une sorte d'**enveloppe qui encapsule les objets** qui peuvent alors être utilisés dans un environnement réparti.

Pour spécifier les interfaces externes des objets, CORBA utilise un langage spécifique, l'**IDL** (*Interface Definition Language*). CORBA définit également les règles de correspondance entre cette interface IDL et de nombreux langages de programmation (ADA, C, C++, Smalltalk, Java, Cobol, PL/I, Python). Chaque interface est traduite dans le langage hôte pour le client (Stub), et pour le serveur (Skeleton).

CORBA permet également aux composants de se découvrir les uns les autres et d'interopérer sur un bus à objets : l'ORB.

C.1 IDL, un langage de définition des interfaces

IDL décrit l'interface, c'est à dire le contrat engageant clients et serveurs. Le langage est purement déclaratif : il ne fournit aucun détail sur l'implémentation.

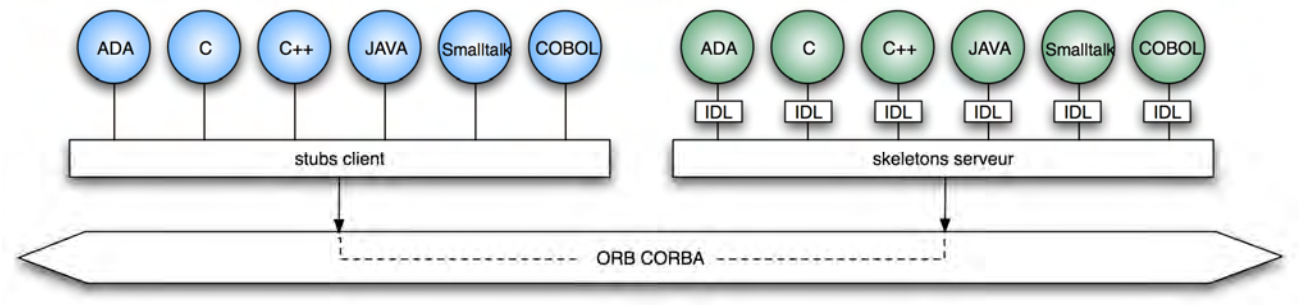

Figure 3 : Liens aux langages de l'IDL CORBA

La grammaire de l'IDL est un sous-ensemble de C++, avec des mots-clefs spécifiques supplémentaires. Elle supporte les instructions et directives de pré-traitement standard de C++.

Exemple d'IDL :

```
interface Echo {
   string echoString (in string mesg);
};
Interface Buffer {
   exception Empty;
   void put (in string content);
   string get() raises (Empty);
};
```

traduction ADA du service echo :

```
with Corba.Object;
package Echo is
   type Ref is new Corba.Object.Ref with null record;
   function To_Echo (Self : in Corba.Object.Ref'Class)
      return Ref'Class;
   function To_Ref (From : in Corba.Any) return Ref;
   function To_Any (From : in Ref) return Corba.Any;
   function echoString (Self : in Ref; msg  : in Corba.String)
      return Corba.String;
   Null_Ref : constant Ref := (Corba.Object.Null_Ref
                               with null record);
   Echo_R_Id : constant Corba.RepositoryId :=
      Corba.To_Unbounded_String («IDL:Echo:1.0»);
end Echo;
```

C.2 Architecture CORBA

L'architecture CORBA est appelée OMA (*Object Management Architecture*). Elle se compose de :

- l'**ORB** (bus à objet, voir plus loin)
- Les **services objets communs** qui fournissent les fonctionnalités de bas niveau
- Les **objets applicatifs** qui sont développés par les utilisateurs de l'architecture
- Les **utilitaires communs** qui sont des objets intermédiaires entre les services et les objets applicatifs.

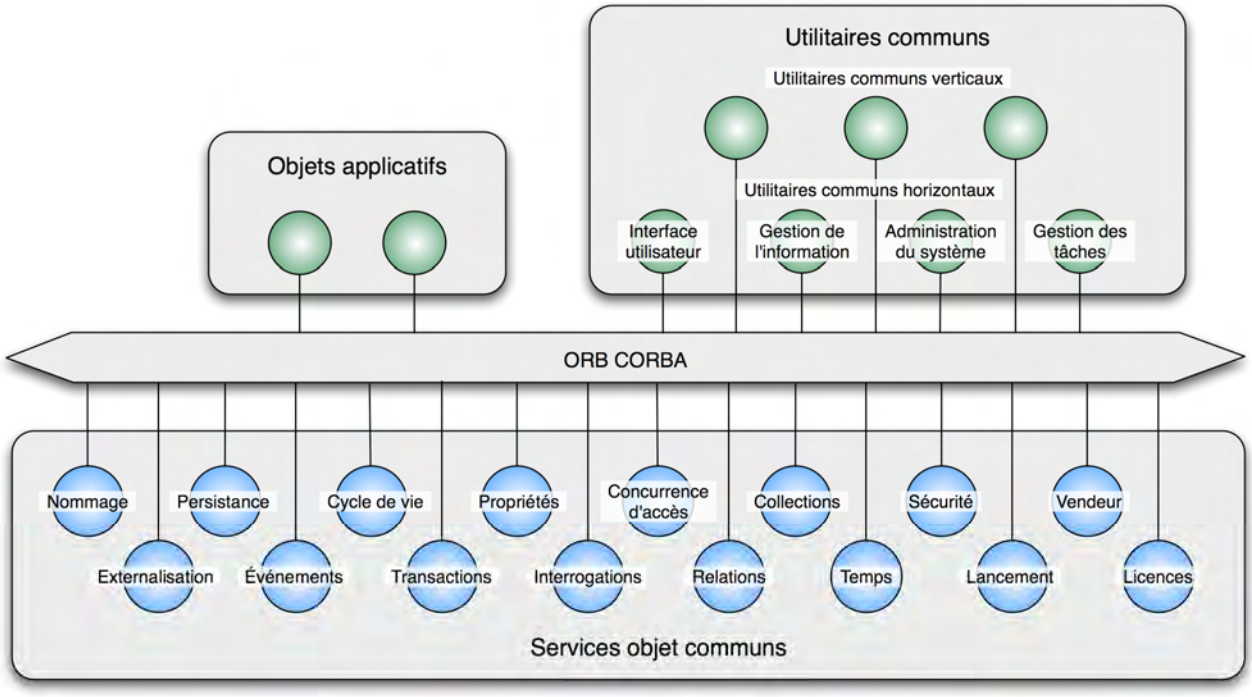

Figure 4 : OMA : Architecture de gestion des objets de l'OMG

C.3 ORB, le bus à objets

L'ORB (*Object Request Broker*) ou courtier d'objets est le bus à objets de CORBA. Il permet aux objets d'émettre des requêtes vers d'autres objets, locaux ou distants, et de recevoir des réponses. Le client n'a pas besoin de connaître les mécanismes utilisés pour communiquer avec le serveur.

En 1991, CORBA 1.1 ne mentionnait que l'IDL, les liens avec les langages et les API destinées à l'accès à l'ORB. Il était ainsi possible d'écrire des programmes portables capables de s'exécuter en utilisant une dizaine d'ORB conformes à CORBA commercialisés.

CORBA 2.0 permet l'interopérabilité entre les ORB de fournisseurs différents.

L'ORB permet aux objets de se découvrir les uns les autres au moment de l'exécution et d'invoquer leurs services respectifs.

Un ORB est beaucoup plus complexe que toutes les autres formes d'intergiciels client/serveur comme les appels de procédure distante (RPC), les intergiciels orientés messages (MOM), les procédures stockées dans les bases de données ou les services poste à poste.

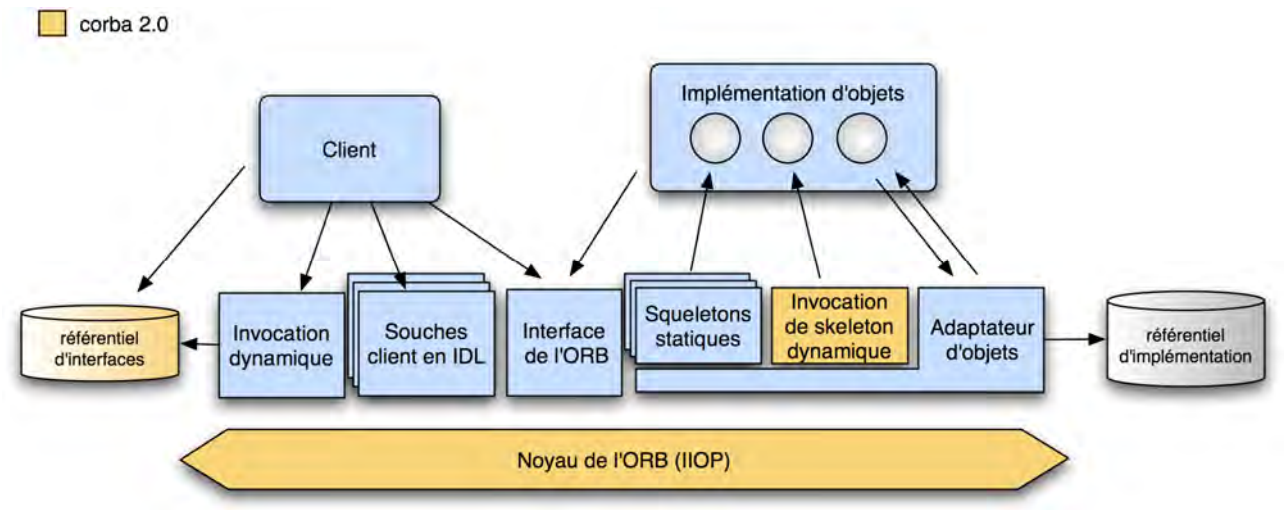

Figure 5 : Structure d'un ORB CORBA 2.0

Les avantages des ORB CORBA :

- **L'invocation de méthodes statiques et dynamiques**.
  L'ORB CORBA permet de définir des appels de méthode à la compilation, ou bien de les découvrir au moment de l'exécution. On peut donc, au choix :
  - obtenir un contrôle des types à la compilation
  - bénéficier de la souplesse associée à la liaison différée.
- **Des liaisons avec les langages de haut niveau.**
  On peut utiliser le langage de son choix pour invoquer des méthodes sur les objets serveur. CORBA sépare l'interface et l'implémentation, et fournit des **types de données indépendants du langage**, ce qui permet d'appeler les objets indépendamment du langage et du système d'exploitation.
- **L'auto-description.**
  Chaque ORB supporte un **référentiel d'interfaces** qui contient une description des fonctions fournies par un serveur et leurs paramètres. Grâce à ces meta-données, les clients découvrent à l'exécution comment ils doivent appeler ces services. Cela permet également aux outils de générer du code « à la volée ». Certains compilateurs sont ainsi capables de générer les stubs et skeletons à partir de bytecode java automatiquement.
- **L'indifférence à la distance.**
  Un ORB peut être local à une machine isolée ou s'interconnecter à un autre ORB via internet grâce à IIOP (*Internet Inter-Orb Protocol*) depuis CORBA 2.0. Un programmeur de système réparti basé sur ORB n'a plus à se préoccuper des piles de transport, de la localisation des serveurs, de l'activation des objets, de la représentation des données ou même du système d'exploitation cible.
- **La sécurité et les transactions sont intégrées**
- **L'envoi de messages ciblés.**
  En plus d'invoquer des fonctions à distance comme en RPC, CORBA les invoque sur des objets cible. Cela signifie que cette invocation aura des effets différents selon l'objet qui les reçoit.

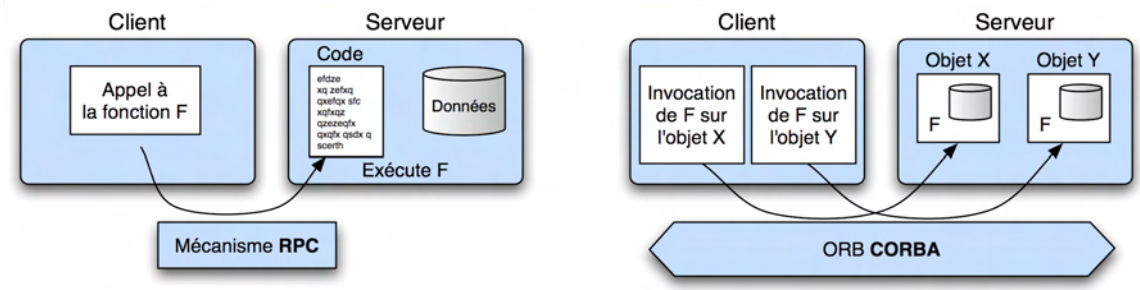

Figure 6 : comparaison des méthodes RPC et CORBA

- **Reprise de l'existant.**
  L'encapsulation permet de transformer un code existant en objet disponible sur l'ORB. On peut ainsi, par exemple, faire cohabiter des objets purs et du code Cobol encapsulé.

Architecture inter-ORB

Depuis CORBA 2.0, les différents ORB peuvent également communiquer entre eux.

Le protocole **GIOP** (*General Inter-Orb Protocol*) définit un ensemble de formats de messages et de représentations communes de données pour assurer la communication entre les ORB.

L'OMG en définit les trois parties :

- Les **formats de message** (au nombre de sept) qui couvrent toute la sémantique requête/réponse de l'ORB.
- Le **CRD** (*Common Data Representation*) convertit les types de données définies dans l'IDL de l'OMG en une représentation plate de type message de réseau. Le CDR se charge aussi des différences entre plates-formes, comme l'agencement des octets et les alignements en mémoire.
- L'**IOR** (*Interoperable Object Reference*) définit le format des références à un objet distant. L'IOR consiste en une série de profils marqués avec leurs composants qui peuvent contenir différentes informations. Un IOR contient généralement la version de protocole, l'adresse du serveur et une séquence d'octets identifiant l'objet distant (la « clef » de l'objet).

Le protocole **IIOP** (*Internet Inter-Orb Protocol*) définit la façon dont les messages GIOP circulent sur un réseau TCP/IP. IIOP permet donc d'utiliser l'Internet comme une dorsale sur laquelle les ORB peuvent établir des ponts. C'est la réalisation concrète des définitions abstraites de GIOP.

C.4 Composants côté client

- Les s**tubs** (*souches*) IDL client fournissent les interfaces statiques aux services objets. Ces souches pré-compilées définissent la façon dont les clients invoquent les services correspondant sur les serveurs.
- Les **interfaces à appel dynamiques** (DII) permettent de ne découvrir la méthode à appeler qu'au moment de l'exécution. CORBA définit des API standard pour rechercher les meta-données définissant l'interface du serveur. Il fournit aussi les API qui permettent de générer dynamiquement les paramètres, d'émettre l'appel à distance et de récupérer les résultats.
- Le **référentiel d'interfaces** (Interface Repository) est une base de données distribuée qui contient les versions exécutables des interfaces définies en IDL. Les API du Référentiel d'interfaces permettent d'obtenir et modifier les descriptions de toutes les interfaces de composant enregistrées, des méthodes qu'elles supportent et des paramètres qu'elles nécessitent. Ce sont des **signatures de méthodes**. On peut aussi le voir comme un référentiel dynamique de meta-données destinées aux ORB.

C.5 Composants côté serveur

- Les **souches IDL du serveur** (*Static Skeleton Interface*) fournissent les interfaces statiques à chaque service exporté par le serveur. Elles sont créées (comme pour les stubs client) en utilisant le compilateur IDL.
- **L'Interface de « Skeleton » Dynamique (***Dynamic Skeleton Interface***)** introduite avec CORBA 2.0 procure, à l'exécution, un mécanisme de liaison aux serveurs qui ont à traiter des appels de méthode entrants pour des composants ne possédant pas de skeleton compilé en IDL. Le skeleton dynamique recherche les valeurs des paramètres dans un message entrant pour déterminer son destinataire, c'est à dire la méthode et l'objet cible. Ils s'avèrent fort utiles pour mettre en place des ponts génériques entre ORB. Ils permettent également aux langages de script de générer dynamiquement des implémentations d'objets. C'est l'équivalent, côté serveur, du DII. Il peut recevoir des appels clients statiques ou dynamiques.
- L'**Adaptateur d'objets** (*Object Adapter*) se situe au dessus des services de communication du noyau de l'ORB : il accepte des requêtes de services à la demande des objets du serveur. Il instancie les objets serveur, leur communique des demandes et leur assigne des *identificateurs d'objets* (ID) aussi appelés références d'objet. Il enregistre les classes qu'il supporte et leurs instances d'exécution (c'est à dire les objets) dans le *référentiel d'implémentations*. Chaque ORB doit supporter un adaptateur standard nommé BOA (*Basic Object Adapter*). Depuis CORBA 3.0, il existe une version portable du BOA, le POA (*Portable Object Adapter*) qui instancie des objets qui ne sont pas actifs en mémoire.
- Le **Référentiel d'implémentations** (*Implementation Repository*) fournit des informations sur les classes que supporte un serveur, sur les objets instanciés et leurs ID, il est utilisé lors de l'exécution. C'est un socle commun qui stocke des informations supplémentaires associées à la mise en place de l'ORB : suivi, journaux d'audit, sécurité, données d'administration.
- L'**Interface de l'ORB** (*ORB Core*) est constituée de quelques API aux services locaux qui sont identiques à celles fournies côté client.

C.6 Les services objet

Les services objet (*Common Object Services*) sont des ensembles de services de niveau système, assemblés sous forme de composants dont les interfaces sont définies en IDL. Ils représentent des extensions et des enrichissements des fonctionnalités de l'ORB.

- Le service **Cycle de vie** définit les opérations de copie, déplacement et suppression des composants sur le bus.
- Le service **Persistance** stocke de manière pérenne les composants sur des supports variés comme des bases de données (SGBD) ou des fichiers simples.
- Le service de **Nommage** permet aux composants de localiser sur le bus d'autres composants par leur nom. Il permet de lier des objets aux annuaires de réseaux ou à des contextes de nommage existants : X500, DCE, NIS+, NDS, LDAP.
- Le service **Événements** permet aux composants du bus de faire enregistrer ou supprimer dynamiquement leur intérêt pour des événements spécifiques. Il définit un canal

d'événements qui collecte des événements et les distribue à des composants qui ne savent rien les uns des autres.

- Le service de **Contrôle de la concurrence d'accès** fournit un gestionnaire de verrouillage à la demande de transactions ou de threads. Il gère les accès concurrents aux ressources.
- Le service **Transactions** fournit une coordination du type « validation à deux phases » entre composants, en utilisant des transactions linéaires ou imbriquées. Il permet de s'assurer qu'une série d'actions sera bien exécutée de façon globale sans qu'aucune parmi l'ensemble n'échoue.
- Le service **Relations** offre le moyen de créer des liens dynamiques entre composants qui ignorent tout les uns des autres. Il permet de faire respecter les contraintes d'intégrité référentielle.
- Le service **Externalisation** permet d'insérer des données dans un composant ou de les en extraire.
- Le service **Interrogations** permet d'interroger les objets. C'est un sur-ensemble de SQL basé sur SQL3 et OQL (*Object Query Language*), le langage des bases de données objet.
- Le service **Contrôle des licences** permet de mesurer l'utilisation des composants afin de rémunérer les éditeurs. Il permet de facturer par session, par noeud, par création d'instance et par site.
- Le service **Propriétés** permet d'associer des valeurs nommées (ou propriétés) à n'importe quel composant. Certaines propriétés dynamiques (liées à un état) sont disponibles : titre, date).
- Le service **Temps** permet la synchronisation des horloges. Il définit et gère aussi les évènements liés au temps.
- Le service **Sécurité** offre un environnement de programmation complet (Framework) pour objets distribués. Il apporte l'authentification, les listes de contrôle d'accès (*Access Control Lists*), la confidentialité, la non répudiation, la délégation.
- Le service **Vendeur** (Trader) est une sorte de « pages jaunes » pour objets : il leur permet de faire connaître et d'offrir leurs services.
- Le service **Collections** fournit des interfaces CORBA pour créer et manipuler les collections classiques de façon générique.

Tous ces services enrichissent le comportement des composants distribués, ils leurs fournissent un environnement robuste et sécurisé. L'OMG offre régulièrement de nouveaux services en fonction des besoins, en utilisant des RFP (request For Proposal), basés sur le principe bien connu des RFC (Request For Comment) de l'internet.

C.7 Les utilitaires communs

Les utilitaires CORBA sont des ensembles d'environnements de programmation (Frameworks) définis en IDL qui fournissent des services utilisables par les objets applicatifs. Il existe des utilitaires horizontaux et verticaux, qui fixent les règles pour que les composants métier collaborent efficacement. Il s'agit, par exemple, des agents itinérants, de l'échange de données, de la gestion des tâches (workflow), des pare-feu (firewall), de l'internationalisation, etc.

C.8 Les objets métier

Les objets métier (Business Objects) permettent de décrire des concepts génériques réutilisables dans diverses applications (client, commande, facture, patient, paiement, etc.). On peut les utiliser selon des combinaisons flexibles. Par opposition aux composants système qui n'ont de sens que pour les systèmes d'information, les objets métier constituent des éléments qu'un utilisateur est capable de reconnaître.

Un objet métier est un élément auto-suffisant, qui possède une interface utilisateur, se trouve dans un certain état, et sait comment coopérer avec d'autres objets métier développés séparément pour exécuter une tâche déterminée.

Comme tous les autres objets CORBA, un objet métier offre ses interfaces à ses clients via l'IDL, et il communique avec les autres objets via l'ORB.

L'OMG, à travers la *Business Object Task Force* définit les règles nécessaires pour que ces composants puissent communiquer.

Les objets métiers sont développés selon le **schéma de conception MVC** (Modèle/Vue/Contrôleur) : Le *modèle* représente l'objet et ses données encapsulées, la *vue* représente l'objet à l'écran, le *contrôleur* définit les réactions de l'interface utilisateur aux évènements provoqués par celui-ci.

C.9 Les beans d'entreprise

Il s'agit ici de « marier » les modèles CORBA et EJB (Entreprise JavaBeans). Les EJB sont des modèles de composants liés au langage Java. Les composants CORBA sont plus ouverts : ils ne sont liés à aucun langage de programmation. Les conteneurs de beans CORBA peuvent ainsi accueillir des EJB et on peut exécuter un bean CORBA à l'intérieur d'un EJB. Les CORBA Beans sont apparus avec CORBA 3.0. À l'instar des composants Java, ils supportent plusieurs interfaces, l'héritage simple, des attributs (ou propriétés) et des « customisers ». Un bean CORBA peut être producteur ou consommateur d'évènements, il est conditionné sous la forme d'un fichier compressé (algorithme zip, extension .car), et il possède un descripteur de déploiement.

C.10 Remarques à propos de la sécurité

Les objets distribués CORBA font face à tous les problèmes classiques des sytèmes client-serveur : on ne peut se fier à aucun système d'exploitation des clients du réseau pour protéger les ressources du serveur ; le réseau étant accessible, on ne peut pas non plus se fier aux informations en cours de transit.

Les objets distribués doivent tenir compte des complications suivantes :

- **Les objets distribués peuvent jouer le rôle à la fois de client et de serveur.** Dans une architecture client/serveur classique, on peut faire confiance aux serveurs mais pas aux clients, ici il faut se méfier des deux.
- **Les objets distribués évoluent en permanence**. L'encapsulation cache aux autres objets l'implémentation des méthodes privées, et les objets peuvent déléguer à d'autres objets certaines parties de leur fonctionnement. Ces délégations peuvent également être constituées dynamiquement à l'exécution.

- **Les interactions entre objets sont parfois obscures**. l'encapsulation interdit de comprendre parfaitement toutes les interactions entre objets.
- **Les interactions sont difficilement prévisibles**. Les objets distribués sont plus souples que toutes les autres formes de systèmes clients-serveurs : ils peuvent interagir de manière très spécifique. C'est une force du modèle à objets distribués mais c'est un risque pour la sécurité.
- **Les objets distribués sont polymorphes**. Il est facile de remplacer un objet de l'ORB par un autre utilisant les mêmes interfaces. C'est ainsi une sorte de « paradis des chevaux de Troie »
- **Les objets distribués peuvent être redimensionnés sans limite**. Chaque objet pouvant être un serveur, l'ORB peut en contenir des millions : la gestion des autorisations d'accès peut alors devenir très délicate.
- **Les objets distribués sont très dynamiques**. Un environnement à objets distribués est par essence « anarchiste ». Les objets sont créés dynamiquement, ils se détruisent automatiquement lorsqu'ils ne sont plus utilisés. Cela peut rapidement transformer le système en un cauchemar en matière de sécurité.

Heureusement, la plupart de ces problèmes peuvent être résolus en gérant la sécurité dans l'ORB CORBA lui-même. L'ORB peut gérer la sécurité pour une gamme de systèmes s'étendant d'un processus ou une machine aux configurations inter-ORB mondiales. Placer la sécurité dans l'ORB permet de se décharger de ce travail dans chaque composant, rendant ceux-ci plus facilement portables. Le service sécurité est disponible pour cette tâche.

C.11 Résumé CORBA

Les composants distribués CORBA offrent des **solutions évolutives et souples** pour les environnements répartis sur les réseaux mondiaux aussi bien qu'en intranet. Les objets métier conditionnés sous forme de beans CORBA peuvent être décomposés et répartis sur plusieurs niveaux pour satisfaire les besoins d'une application. Ils sont auto-descriptifs et auto-administrés, on peut les déplacer et les exécuter à l'endroit le plus approprié. CORBA permet de **reprendre les applications existantes** sans avoir à repartir de zéro. On peut ainsi encapsuler les applications existantes et y ajouter de nouvelles fonctions petit à petit, composant par composant.

Forces

- **Interopérabilité** : le client peut être un objet java localisé sur une machine exécutant le système d'exploitation Windows, et le serveur une application écrite en Cobol sur un serveur AS 400
- **Localisation des objets intégrée**
- **Auto-description des composants**
- **Reprise de l'existant**

Faiblesses

- **L'IDL** est un sous-ensemble commun à tous les langages hôtes, il est donc **peu expressif** (par exemple, il est impossible d'utiliser des sous-types).
- Il est nécessaire pour le programmeur d'**apprendre un nouveau langage** qui est difficile d'accès.

D Comparaison des approches ADA 95 et CORBA

Il est difficile d'exercer une comparaison exhaustive entre deux modèles assez différents. En effet, les philosophies et les objectifs sont très différents :

- Le standard CORBA a été créé pour permettre la distribution dans des environnements hétérogènes. Les langages utilisables pour créer les composants sont nombreux mais il est nécessaire d'utiliser une couche d'interface commune.
- ADA 95 et son annexe DSA, au contraire, minimisent les différences entre les applications monolithiques et les applications distribuées, aucune interface n'est nécessaire, mais le langage utilisé est unique et moins répandu.

Nous allons néanmoins tenter de comparer ces deux approches.

Beaucoup de gens pensent que CORBA est LA solution pour programmer des applications distribuées. La force de l'OMG, la promotion de CORBA à travers livres, articles, conférences, son utilisation massive rendent par conséquent cette spécification incontournable.

D'un autre côté, l'annexe DSA de ADA 95 fut publiée après CORBA et ne reçut aucune publicité. Pire, la communauté ADA elle-même est restée expectative et l'annexe a failli disparaître du manuel de référence ! De plus, en 1999, bien peu d'applications l'utilisaient.

D.1 Premiers contacts

En informatique, le premier contact avec une nouvelle technologie est souvent prépondérant.

L'OMG clame régulièrement la simplicité de CORBA. C'est vrai si on se limite aux spécifications de base de CORBA (1.0). Cependant, comme CORBA est le fruit d'une collaboration de nombreux vendeurs de logiciels qui sont également concurrents, la spécification tend vers le plus petit dénominateur commun. Malgré cela, le dessein original a été régulièrement enrichi pour faire face à de plus en plus de fonctionnalités demandées par les différents acteurs de l'OMG. La possibilité d'interagir avec de nombreux langages de programmations à également augmenté la complexité du modèle. Le programmeur doit apprendre et assimiler un nouveau modèle et son langage associé (IDL) avant de commencer à écrire la moindre ligne de code de son programme dans son langage habituel.

L'annexe DSA, de son côté, semble complexe, le manuel de référence est assez abscons, et pourtant son utilisation est assez simple ! **L'annexe DSA a été conçue pour faire face à tous les types d'applications partagées** : des systèmes de calcul à processeurs multiples et mémoire partagée aux stations de travail connectées en réseau. L'appel à une structure distante s'apparente à celui d'un sous-programme, un objet distribué est un objet comme un autre et la syntaxe d'appel est très simple. De plus, la mise en place de la distribution sur une application ne requiert aucune connaissance au niveau système car l'environnement prend en charge les aspects réseau et les appels systèmes.

D.2 Gestion du système

ADA 95 ne propose aucun outil de gestion du système réparti.

CORBA doit prendre en compte nombre de situations et propose des solutions qui pourront satisfaire n'importe quel utilisateur dans n'importe quelle situation. Le coût de cette souplesse est assez élevé : le programmeur doit consulter une documentation assez volumineuse avant de pouvoir l'utiliser.

D.3 Interfaces

CORBA utilise un langage spécifique (l'IDL) pour que les différents systèmes communiquent. Les objets du système doivent mettre en place une couche intermédiaire entre eux-mêmes et le bus de communication.

ADA 95 n'a pas besoin de langage complémentaire ni d'aucune extension. Cela permet d'éviter les couches d'adaptation entre le langage utilisé pour confectionner l'objet et la définition de l'interface.

D.4 Concepts

Dans les systèmes distribués, les concepts à assimiler sont nombreux : enregistrer le serveur auprès du service d'annuaire, le rendre disponible aux requêtes des clients nécessitent de nombreux appels système dont la sémantique est loin d'être évidente.

Avec CORBA, les spécifications d'interfaces sont simples, mais la complexité réside dans leur mise en place et l'utilisation concrète de la plate forme choisie.

Avec ADA 95, nul besoin d'enrichir son application avec des appels systèmes compliqués : le programmeur peut se concentrer sur son application. Le passage au mode distribué ressemble à la programmation multi-threads. C'est à la compilation que tout se joue : il suffit d'indiquer les propriétés et comportements désirés lors de celle-ci pour transformer une application monolithique en une application répartie.

D.5 Portabilité

CORBA propose une interface commune basique : les éditeurs se sentent libres de mettre en place les spécifications de l'OMG à leur façon ou d'ajouter du code propriétaire qui leur simplifient le développement. L'OMG standardise les pratiques et la syntaxe mais répugne à spécifier les détails fins de chaque objet. Les bonnes pratiques sont ainsi définies de façon trop laxiste, il en résulte des difficultés à faire interopérer les objets. Les extensions spécifiques à une plate-forme pullulent. Cette situation entraîne de nombreuses incompatibilités et nécessite souvent un long travail d'adaptation pour que tous les composants communiquent efficacement.

En ADA 95, la portabilité est difficile à estimer. Il existe un compilateur open-source porté sur de nombreuses plate-formes (GLADE ajoute le support de l'annexe DSA à GNAT le compilateur historique) et le même code source peut être utilisé (avec quelques adaptations) sur nombre d'entre elles. Néanmoins, l'absence de concurrent à l'annexe DSA empêche de mesurer la portabilité entre fournisseurs. La configuration de la répartition se fait au moyen du partitionnement, après la compilation.

D.6 Passerelles entre les deux approches

- CIAO permet d'exporter des services ADA95 vers CORBA : un client CORBA émet des requêtes vers l'annexe répartie ADA95.
- DROOPI est un intergiciel générique qui supporte CORBA, ADA95, etc.)

D.7 Résumé (partiel) des caractéristiques ADA 95 et CORBA

	ADA 95	CORBA
Objets répartis	✓	✓
RPC	✓	
Support multi langages		✓
Mémoire virtuelle partagée	✓	
Richesse du langage de description	++	+
Sémantique déjà connue du programmeur ?	✓	
Règles de typage, visibilité, sécurité préservées ?	✓	
Localisation des services	automatique	largement assisté
Avortement d'appel distant	✓	
Sécurité	++	++
Services communs (cycle de vie, licences, etc.)		✓

Conclusion

Après six semaines d'études, nous avons acquis une vision plus précise de la répartition.

Les philosophies et les objectifs désignés de ADA 95 et de CORBA étant très différents, il nous a semblé difficile de désigner une solution universelle qui couvrira tous les besoins de la répartition, dans tous les cas de figure.

ADA 95 possède certaines particularités uniques qui peuvent être décisives (avortements d'appels, typage fort), notamment dans les systèmes sensibles et « temps réels ».

Le critère de l'interopérabilité nous a pourtant semblé fondamental dans les systèmes répartis : sur ce point, CORBA est incontestablement le mieux placé : il permet de faire communiquer des applications écrites dans de nombreux langages de programmations.

La possibilité d'intégrer des applications historiques (développées en langage objet ou non) dans un système réparti plaide également en faveur de la mise en place d'un intergiciel tel que CORBA.

De nombreux services disponibles dans l'environnement CORBA (annuaires, cycle de vie, contrôle des licences, transactions, intégration des EJB) font cruellement défaut en ADA.

Malgré une plus grande complexité, la nécessité pour les programmeurs d'acquérir un nouveau paradigme, la mise en place d'un intergiciel tel que CORBA nous semble malgré tout une solution plus flexible pour mettre en place la répartition.

Table des illustrations

Index

Quelques définitions

Un **type abstrait** est un type que l'on ne peut pas instancier (ou « virtuel » dans d'autres langages). Son rôle est de définir un gabarit : les types concrets dérivés d'un type abstrait devront obligatoirement redéfinir toutes les méthodes abstraites du type de base. Un type ou une méthode abstrait est identifié par la présence du mot-clé `abstract`.

Un **type étiqueté** (ou « *tagged type* » en anglais) est une classe dans la terminologie objet.

Un **type limité** (ou « *limited type* » en anglais) est un type dont il n'est pas possible de copier les instances, l'opération d'affectation n'étant pas définie.

Un **type privé** (ou « *private type* » en anglais) est un type dont la définition complète n'est pas accessible en dehors du paquetage le définissant et de ses paquetages enfants ; notamment dans le cas d'un type composite, structure ou tableau, il n'est pas possible d'accéder aux différents champs ou enregistrements d'une instance.

Bibliographie

Livres

[ORF 1999] ORFALI R., HARKEY D., EDWARDS J. « Client/Serveur guide de survie », *Vuibert*, Paris, 1999

[FAY 1996] FAYARD D., ROUSSEAU, M. « Vers ADA 95 par l'exemple », *Bibliothèque des universités*, Paris, 1996

Publications

[TAR 2004] TARDIEU, S., « Systèmes répartis », ENST, 2004

[PQT 2000] PAUTET L., QUINOT T., TARDIEU S., « Objets répartis avec ADA 95 », ENST, 2000

[KER 1999] KERMAREC Y., « Corba vs. ADA 95 DSA a programmer's view », *Proceedings of the SIGAda '99 conference*, New York, N.Y. USA, 1999, ACM Press, p. 39-46.

[PQT 1999] PAUTET L., QUINOT T., TARDIEU S., « Corba and Corba Services for DSA », *Proceedings of the SIGAda '99 conference*, New York, N.Y. USA, 1999, ACM Press, p. 31-38.

[ROU 2005] ROUSTANG., A., « RMI application distribuée », *Laboratoire supinfo des technologies Sun*, www.labo-sun.com, 2005.

Sites internet

OMG http://www.omg.org/

ADA France : http://www.ada-france.org/

ADA core : http://www.adacore.com/home/

FAQ ADA : http://fr.wikibooks.org/wiki/Programmation_Ada/FAQ/

Wikipedia : http://fr.wikipedia.org/